SAPIENTIA
ERDÉLYI MAGYAR
TUDOMÁNYEGYETEM
CSÍKSZEREDAI KAR

jegyzet



PÁL LÁSZLÓ

WEBPROGRAMOZÁS PHP-BEN

Scientia Kiadó | 2026

Pál László

Webprogramozás PHP-ben



SAPIENTIA ERDÉLYI MAGYAR TUDOMÁNYEGYETEM
CSÍKSZEREDAI KAR
GAZDASÁGTUDOMÁNYI TANSZÉK

Pál László

Webprogramozás PHP-ben

Scientia Kiadó
Kolozsvár · 2026

Készült
a Magyar kormány
támogatásával

Társadalmi Kapcsolatokért
és Kultúráért Felelős Minisztérium

Bethlen Gábor
Alap

DOI: 10.47745/SAPVOL.2026.34

Felelős kiadó:
Sorbán Angella

Lektor:
Szabó László Zsolt (Marosvásárhely)

Borítóterv:
Tipotéka Kft.

Kiadói koordinátor:
Szabó Beáta

A szakmai felelősséget teljes mértékben a szerkesztők, illetve a szerzők vállalják.

Első kiadás
© Scientia 2026
Minden jog fenntartva, beleértve a sokszorosítás, a nyilvános előadás, a rádió- és televízióadás, valamint a fordítás jogát, az egyes fejezeteket illetően is.

ISBN: 978-606-975-123-7

TARTALOMJEGYZÉK

Előszó 14

1. Webes alapok és PHP-bevezetés 16
1.1. A web működése, kliens–szerver modell 16
1.2. A PHP története, jellemzői 19
1.3. A fejlesztői környezet beállítása 22
1.4. Bemutató PHP-program 25
1.5. Kódolási konvenciók 27

2. A PHP-nyelv elemei 33
2.1. Szintaxis, változók, adattípusok 33
2.2. Karakterláncok kezelése 39
2.3. Operátorok 45
2.4. Vezérlési szerkezetek 51
2.5. Függvények 63
2.6. Tömbök 76
2.7. Dátum- és időkezelés 85
2.8. Gyakorlófeladatok 89

3. Objektumorientált programozás 94
3.1. Bevezetés 94
3.2. Osztályok és objektumok 94
3.3. Tulajdonságok, metódusok és láthatóság 98
3.4. Konstruktor és destruktor 108
3.5. Öröklődés 112
3.6. Kompozíció és aggregáció 117
3.7. Absztrakt osztályok és interfészek 119
3.8. Objektumok vizsgálata és összehasonlítása 126
3.9. Polimorfizmus 128
3.10. Metódustúlterhelés és magic metódusok 130
3.11. Traitek 138
3.12. Felsorolások (enum típusok) 144
3.13. Automatikus osztálybetöltés 148
3.14. Névterek 151
3.15. Hibakezelés és kivételkezelés 154
3.16. SOLID-elvek 160
3.17. Gyakorlófeladatok 160

4. Űrlapok kezelése 163
4.1. Űrlapok alapjai 163
4.2. GET- és POST-metódusok 166
4.3. Űrlapmezők feldolgozása PHP-ben 174
4.4. Kliensoldali és szerveroldali validáció 180

4.5. Adatok ellenőrzése és szűrése ... 182
4.6. Biztonsági megfontolások ... 189
4.7. Fájlfeltöltés kezelése ... 193
4.8. Gyakorlófeladatok ... 197

5. Sütik és munkamenetek kezelése ... 200
5.1. Állapottartási technikák ... 200
5.2. Sütik létrehozása és kezelése ... 203
5.3. Munkamenetek használata ... 213
5.4. Gyakorlófeladatok ... 221

6. Adatbázis-kezelés ... 223
6.1. Az adatbázisok szerepe és típusai ... 223
6.2. SQL- és MySQL-alapok ... 224
6.3. Adatbázis-kezelés PHP-ben ... 228
6.4. Előkészített utasítások és biztonság ... 231
6.5. CRUD-műveletek ... 234
6.6. Tranzakciókezelés ... 242
6.7. Teljesítményoptimalizálás ... 246
6.8. Gyakorlófeladatok ... 251

7. REST API-fejlesztés ... 254
7.1. Webszolgáltatások áttekintése ... 254
7.2. A REST alapjai ... 255
7.3. API-végpontok tervezése ... 259
7.4. HTTP-metódusok és státuszkódok ... 261
7.5. JSON-válaszok ... 266
7.6. REST API-implementáció PHP-ben ... 270
7.7. Biztonság REST API-kban ... 284
7.8. REST API-tesztelés ... 295
7.9. Gyakorlófeladatok ... 298

8. Composer és csomagkezelés ... 301
8.1. Composer-alapok ... 301
8.2. Projekt inicializálása ... 302
8.3. Csomagok kezelése ... 304
8.4. Automatikus osztálybetöltés ... 306
8.5. Composer-szkriptek ... 307
8.6. Projektstruktúra ... 307
8.7. Gyakran használt parancsok ... 310
8.8. Tipikus hibák ... 311
8.9. Gyakorlati példák ... 312
8.10. Gyakorlófeladatok ... 315

9. MVC típusú alkalmazások ... 317
9.1. Az MVC-architektúra alapelvei ... 317
9.2. Egyszerű MVC-keretrendszer készítése ... 320
9.3. Routing ... 326

9.4. Kontrollerek és modellek ... 330
9.5. Nézetek és sablonrendszerek ... 335
9.6. Dependency Injection alapok ... 339
9.7. Gyakorlófeladatok ... 343

10. Tesztelés és verziókövetés ... 345
10.1. Egységtesztelés PHPUnittal ... 345
10.2. Git-alapok ... 356
10.3. GitHub-munkafolyamat ... 363
10.4. Gyakorlófeladatok ... 368
Szakirodalom ... 371
A szerzőről ... 374

CUPRINS

Prefață 14

1. Noțiuni web de bază și introducere în PHP 16
1.1. Funcționarea web-ului: modelul client–server 16
1.2. Istoricul și caracteristicile PHP 19
1.3. Configurarea mediului de dezvoltare 22
1.4. Un prim program PHP 25
1.5. Convenții de codare 27

2. Elementele limbajului PHP 33
2.1. Sintaxă, variabile și tipuri de date 33
2.2. Gestionarea șirurilor de caractere 39
2.3. Operatori 45
2.4. Structuri de control 51
2.5. Funcții 63
2.6. Tablouri 76
2.7. Gestionarea datelor și a timpului 85
2.8. Exerciții practice 89

3. Programare orientată pe obiecte 94
3.1. Introducere 94
3.2. Clase și obiecte 94
3.3. Proprietăți, metode și vizibilitate 98
3.4. Constructor și destructor 108
3.5. Moștenire 112
3.6. Compoziție și agregare 117
3.7. Clase abstracte și interfețe 119
3.8. Examinarea și compararea obiectelor 126
3.9. Polimorfism 128
3.10. Supraîncărcarea metodelor și metode magice 130
3.11. Trait-uri 138
3.12. Enumerări 144
3.13. Autoîncărcarea claselor 148
3.14. Spații de nume 151
3.15. Gestionarea erorilor și a excepțiilor 154
3.16. Principiile SOLID 160
3.17. Exerciții practice 160

4. Gestionarea formularelor 163
4.1. Noțiuni de bază despre formulare 163
4.2. Metodele GET și POST 166
4.3. Prelucrarea câmpurilor de formular în PHP 174
4.4. Validare pe client și pe server 180
4.5. Validarea și filtrarea datelor 182

4.6. Considerații de securitate ... 189
4.7. Gestionarea încărcării fișierelor ... 193
4.8. Exerciții practice ... 197

5. Cookie-uri și sesiuni ... 200
5.1. Tehnici de gestionare a stării ... 200
5.2. Crearea și gestionarea cookie-urilor ... 203
5.3. Utilizarea sesiunilor ... 213
5.4. Exerciții practice ... 221

6. Gestionarea bazelor de date ... 223
6.1. Rolul și tipurile de baze de date ... 223
6.2. Noțiuni de bază despre SQL și MySQL ... 224
6.3. Gestionarea bazelor de date în PHP ... 228
6.4. Instrucțiuni pregătite și securitate ... 231
6.5. Operații CRUD ... 234
6.6. Gestionarea tranzacțiilor ... 242
6.7. Optimizarea performanței ... 246
6.8. Exerciții practice ... 251

7. Dezvoltarea REST API ... 254
7.1. Prezentare generală a serviciilor web ... 254
7.2. Principiile REST ... 255
7.3. Proiectarea endpoint-urilor API ... 259
7.4. Metode HTTP și coduri de stare ... 261
7.5. Răspunsuri JSON ... 266
7.6. Implementarea REST API în PHP ... 270
7.7. Securitate în REST API ... 284
7.8. Testarea REST API ... 295
7.9. Exerciții practice ... 298

8. Composer și gestionarea pachetelor ... 301
8.1. Noțiuni de bază despre Composer ... 301
8.2. Inițializarea unui proiect ... 302
8.3. Gestionarea pachetelor ... 304
8.4. Autoîncărcarea claselor ... 306
8.5. Scripturi Composer ... 307
8.6. Structura proiectului ... 307
8.7. Comenzi frecvent utilizate ... 310
8.8. Erori tipice ... 311
8.9. Exemple practice ... 312
8.10. Exerciții practice ... 315

9. Aplicații de tip MVC ... 317
9.1. Principiile arhitecturii MVC ... 317
9.2. Construirea unui cadru MVC simplu ... 320
9.3. Rutare ... 326
9.4. Controlere și modele ... 330

9.5. Vizualizări și sisteme de șabloane 335
9.6. Noțiuni de bază despre injectarea dependențelor 339
9.7. Exerciții practice 343

10. Testare și gestiunea versiunilor 345
10.1. Testare unitară cu PHPUnit 345
10.2. Noțiuni de bază despre Git 356
10.3. Fluxul de lucru GitHub 363
10.4. Exerciții practice 368
Bibliografie 371
Rezumat 372
Despre autor 374

TABLE OF CONTENTS

Preface 14

1. Web Basics and Introduction to PHP 16
1.1. How the Web Works: The Client–Server Model 16
1.2. History and Characteristics of PHP 19
1.3. Setting Up the Development Environment 22
1.4. A First PHP Program 25
1.5. Coding Conventions 27

2. PHP Language Fundamentals 33
2.1. Syntax, Variables, and Data Types 33
2.2. String Handling 39
2.3. Operators 45
2.4. Control Structures 51
2.5. Functions 63
2.6. Arrays 76
2.7. Date and Time Handling 85
2.8. Exercises 89

3. Object-Oriented Programming 94
3.1. Introduction 94
3.2. Classes and Objects 94
3.3. Properties, Methods, and Visibility 98
3.4. Constructor and Destructor 108
3.5. Inheritance 112
3.6. Composition and Aggregation 117
3.7. Abstract Classes and Interfaces 119
3.8. Inspecting and Comparing Objects 126
3.9. Polymorphism 128
3.10. Method Overloading and Magic Methods 130
3.11. Traits 138
3.12. Enumerations 144
3.13. Autoloading 148
3.14. Namespaces 151
3.15. Error Handling and Exceptions 154
3.16. SOLID Principles 160
3.17. Exercises 160

4. Form Handling 163
4.1. Form Basics 163
4.2. GET and POST Methods 166
4.3. Processing Form Fields in PHP 174
4.4. Client-Side and Server-Side Validation 180
4.5. Data Validation and Filtering 182

4.6. Security Considerations 189
4.7. File Upload Handling 193
4.8. Exercises 197

5. Cookies and Sessions 200
5.1. State Management Techniques 200
5.2. Creating and Managing Cookies 203
5.3. Working with Sessions 213
5.4. Exercises 221

6. Database Management 223
6.1. The Role and Types of Databases 223
6.2. SQL and MySQL Basics 224
6.3. Database Management in PHP 228
6.4. Prepared Statements and Security 231
6.5. CRUD Operations 234
6.6. Transaction Management 242
6.7. Performance Optimization 246
6.8. Exercises 251

7. REST API Development 254
7.1. Overview of Web Services 254
7.2. REST Fundamentals 255
7.3. Designing API Endpoints 259
7.4. HTTP Methods and Status Codes 261
7.5. JSON Responses 266
7.6. REST API Implementation in PHP 270
7.7. Security in REST APIs 284
7.8. REST API Testing 295
7.9. Exercises 298

8. Composer and Package Management 301
8.1. Composer Basics 301
8.2. Initializing a Project 302
8.3. Managing Packages 304
8.4. Autoloading 306
8.5. Composer Scripts 307
8.6. Project Structure 307
8.7. Frequently Used Commands 310
8.8. Common Errors 311
8.9. Practical Examples 312
8.10. Exercises 315

9. MVC Applications 317
9.1. MVC Architecture Principles 317
9.2. Building a Simple MVC Framework 320
9.3. Routing 326
9.4. Controllers and Models 330

9.5. Views and Template Systems 335
9.6. Dependency Injection Basics 339
9.7. Exercises 343

10. Testing and Version Control 345
10.1. Unit Testing with PHPUnit 345
10.2. Git Basics 356
10.3. GitHub Workflow 363
10.4. Exercises 368
Bibliography 371
Abstract 373
About the Author 374

ELŐSZÓ

Ez a kötet a szerveroldali webprogramozást mutatja be PHP-nyelven, az alapoktól a modern webalkalmazások felépítéséig. A PHP napjainkban is az egyik legelterjedtebb szerveroldali programozási nyelv, amelyet számos webalkalmazás és tartalomkezelő rendszer használ. Rugalmassága, gazdag dokumentációja és aktív közössége miatt jó kiindulópont a webfejlesztés tanulásához.

A tananyag gyakorlatorientált szemléletet követ: a nyelvi alapok mellett hangsúlyt kapnak azok az eszközök és fejlesztési minták is, amelyek a modern PHP-alapú webalkalmazások fejlesztése során gyakran előfordulnak. A példák és feladatok célja nem csupán az egyes nyelvi elemek bemutatása, hanem azok alkalmazása valós webes környezetben.

A könyv nem feltételez előzetes PHP-ismeretet, azonban az algoritmikus gondolkodás és egy programozási nyelv alapjainak ismerete ajánlott. A HTML és CSS alapszintű ismerete szintén hasznos, mivel a szerveroldali kód kimenete jellemzően ezekre épül.

A fejezetek felépítése

A tíz fejezet egymásra épülve vezeti be a szerveroldali webfejlesztés főbb fogalmait. Az első fejezetek a web működését és a PHP alapjait ismertetik, ezt követi az objektumorientált programozás, majd az összetettebb témák: űrlapkezelés, munkamenetek, adatbázis-kezelés, REST API-fejlesztés, Composer-alapú projektkezelés, MVC-architektúra, automatizált tesztelés és verziókövetés.

A fejezetek elméleti áttekintéssel és kódpéldákkal kezdődnek, és a bevezető fejezet kivételével gyakorlófeladatokkal zárulnak.

A kötet a natív PHP használatára épít, ezért nem egy konkrét keretrendszerre koncentrál. Célja az alapvető webfejlesztési fogalmak, a HTTP-alapú kommunikáció és a szerveroldali alkalmazások felépítésének megismertetése. Az itt megszerzett ismeretek jó alapot nyújtanak modern PHP-keretrendszerek tanulásához is.

Kódpéldák és megoldások

A fejezetekben szereplő példák önállóan is kipróbálhatók és továbbfejleszthetők. A könyv kódpéldái a GitHub-tárolóban futtatható formában, a könyvbeli sorszámukkal megtalálhatók, így a teljes, működő kód is áttekint-

hető. A fájlok fejezetenként, az alfejezetek sorrendjét követő számozással vannak elrendezve. A tároló néhány helyen bővebb példákat is tartalmaz. A gyakorlófeladatok az ismeretek elmélyítését szolgálják.

A kötetben szereplő kódpéldák és a gyakorlófeladatok megoldásai az alábbi GitHub-tárolóból tölthetők le:

`https://github.com/pallaszlo/webprogramozas-php`

Csíkszereda, 2026. november *Pál László*

1. FEJEZET

WEBES ALAPOK ÉS PHP-BEVEZETÉS

1.1. A web működése, kliens–szerver modell

A világháló (World Wide Web) olyan rendszer, amely lehetővé teszi weboldalak és más erőforrások elérését az interneten keresztül. A web működésének alapja a *kliens–szerver modell*, amely szétválasztja a kéréseket küldő klienseket és a szolgáltatást nyújtó szervereket.

1.1.1. Kliens

A kliens általában egy webböngésző (pl. Google Chrome, Mozilla Firefox, Safari, Microsoft Edge), de lehet bármilyen más program is, amely képes HTTP-kérések küldésére és a válaszok értelmezésére. A kliens fő feladatai:

- a felhasználói interakciók kezelése (pl. URL beírása, linkre kattintás);
- a HTTP-kérések összeállítása és elküldése;
- a szervertől kapott válasz feldolgozása és megjelenítése;
- kliensoldali szkriptek futtatása (pl. JavaScript).

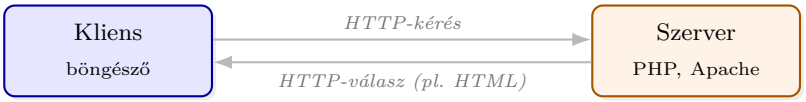
1.1.2. Szerver

A webfejlesztésben a *szerver* kifejezés általában a kiszolgáló szoftvert jelenti (nem magát a hardvert). A webszerver fő feladatai:

- HTTP-kérések fogadása és feldolgozása;
- megfelelő erőforrások (pl. HTML-fájlok, képek) vagy dinamikusan generált tartalom szolgáltatása;
- adatbázis-műveletek kezelése;
- szerveroldali szkriptek futtatása (pl. PHP);
- biztonsági és jogosultságkezelési feladatok ellátása.

Népszerű webszerverek: Apache HTTP Server, Nginx, Microsoft Internet Information Services (IIS).

A kliens és a szerver közötti alapvető kommunikációt az 1.1. ábra szemlélteti.



1.1. ábra. *A kliens–szerver kommunikáció alapja*

1.1.3. HTTP-protokoll

A Hypertext Transfer Protocol (HTTP)[1] egy olyan protokoll, amely lehetővé teszi a kommunikációt a kliens és a szerver között. A protokoll főbb jellemzői:

- **Állapotmentes:** Minden kérés független az előzőektől.
- **Kérés–válasz alapú:** A kliens kérést küld, amelyre a szerver választ ad.
- **Szöveges:** A kérések és a válaszok ember által is olvasható formátumúak.

Egy tipikus HTTP-kérés felépítése:

```
GET /index.html HTTP/1.1
Host: www.example.com
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv
    :88.0) Gecko/20100101 Firefox/88.0
Accept: text/html,application/xhtml+xml,application/xml;q
    =0.9,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate, br
Connection: keep-alive
```

1.1. kódpélda. *Példa HTTP GET-kérés*

A szerver válasza hasonló felépítésű:

```
HTTP/1.1 200 OK
Date: Mon, 23 Mar 2026 10:15:00 GMT
Server: Apache/2.4.62 (Unix) OpenSSL/3.3.1 PHP/8.4.1
Last-Modified: Mon, 02 Mar 2026 09:00:00 GMT
```

[1] https://www.rfc-editor.org/rfc/rfc9110

```
Content-Type: text/html; charset=UTF-8

<!DOCTYPE html>
<html lang="hu">
<head>
    <title>Példaoldal</title>
</head>
<body>
    <h1>Helló, Világ!</h1>
</body>
</html>
```

1.2. kódpélda. *Példa HTTP-válasz*

1.1.4. HTTPS

A HTTP-protokoll titkosítás nélkül továbbítja az adatokat, ezért az érzékeny információk, például jelszavak vagy személyes adatok, a hálózaton lehallgathatók. A HTTPS (Hypertext Transfer Protocol Secure) a HTTP-protokoll TLS (Transport Layer Security) titkosítással kiegészített változata. A TLS biztosítja, hogy

- az adatforgalom titkosított legyen, így illetéktelenek ne tudják értelmezni az átvitt adatokat;
- a kliens ellenőrizni tudja a szerver hitelességét, vagyis megbizonyosodhasson arról, hogy valóban a kívánt szerverrel kommunikál;
- az adatok az átvitel során ne módosulhassanak.

HTTPS használata esetén a kommunikáció titkosított kapcsolaton keresztül történik. Éles webalkalmazásokban kizárólag HTTPS használata ajánlott, a HTTP-kérések automatikus átirányítása HTTPS-re pedig általános bevett gyakorlat.

1.1.5. A weboldal betöltésének folyamata

A weboldal betöltése több lépésből álló folyamat (lásd az 1.2. ábrát):

1. **DNS-lekérdezés:** A böngésző a DNS-szerver (Domain Name System) segítségével meghatározza a domainnévhez tartozó IP-címet.
2. **TCP-kapcsolat:** A böngésző hálózati kapcsolatot épít fel a szerverrel a TCP (Transmission Control Protocol) protokoll segítségével.
3. **HTTP-kérés:** A böngésző elküldi a kérést a kívánt erőforrás eléréséhez.

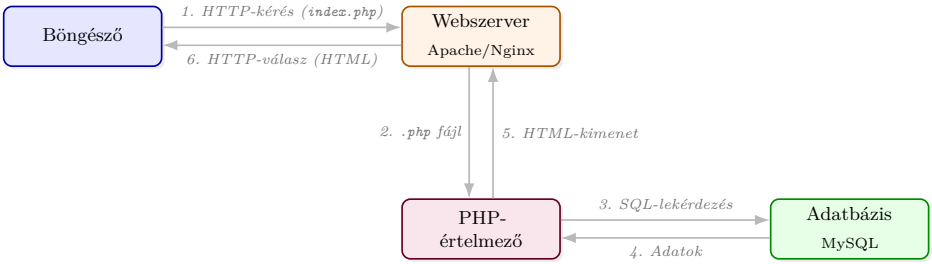
4. **Szerveroldali feldolgozás (PHP):** A webszerver átadja a kérést a PHP-értelmezőnek, amely lefuttatja a szkriptet, elvégzi az esetleges adatbázis-lekérdezéseket, és előállítja a dinamikus HTML-t.
5. **Renderelés (böngésző):** A böngésző feldolgozza a HTML-t, letölti a szükséges erőforrásokat (CSS, JavaScript, képek), majd megjeleníti az oldalt.



1.2. ábra. *A weboldal betöltésének folyamata*

1.1.6. A PHP-feldolgozás folyamata

Amikor a böngésző egy `.php` kiterjesztésű oldalt kér le, a webszerver nem egyszerűen elküldi a fájlt, hanem átadja azt a PHP-értelmezőnek feldolgozásra. Az értelmező lefuttatja a kódot, elvégzi az esetleges adatbázis-lekérdezéseket, majd az eredményből HTML-t állít elő. Ezt a folyamatot az 1.3. ábra szemlélteti.



1.3. ábra. *A PHP-kérés feldolgozásának folyamata*

A böngésző soha nem látja a PHP-forráskódot, csak a már legenerált HTML-t kapja meg. Az adatbázis-lépés opcionális: egyszerűbb oldalak esetén el is maradhat.

1.2. A PHP története, jellemzői

1.2.1. A PHP története

A PHP (PHP: Hypertext Preprocessor, eredetileg Personal Home Page Tools) (Zandstra 2021) egy szerveroldali szkriptnyelv, amelyet webalkalma-

zások fejlesztésére terveztek. A nyelv története 1994-ben kezdődött, amikor Rasmus Lerdorf létrehozta az első verziót.

Főbb mérföldkövek:

- **1994:** Rasmus Lerdorf létrehozza az első verziót személyes weboldala látogatottságának nyomon követésére;
- **1995:** PHP/FI – az első nyilvánosan elérhető verzió (Personal Home Page / Forms Interpreter);
- **1997:** PHP/FI 2.0 – becslések szerint már több tízezer weboldal használja;
- **1998:** PHP 3.0 – teljes újraírás, a név ekkor változik „PHP: Hypertext Preprocessor”-ra;
- **2000:** PHP 4.0 – a Zend Engine bevezetése, jelentős teljesítményjavulás;
- **2004:** PHP 5.0 – az objektumorientált programozás támogatásának megerősítése;
- **2015:** PHP 7.0 – jelentős teljesítményjavulás, típusdeklarációk bevezetése;
- **2020:** PHP 8.0 – JIT-fordító, név szerinti argumentumok, union típusok, match kifejezés;
- **2021:** PHP 8.1 – felsorolások (enum), csak olvasható tulajdonságok, fibers;
- **2022:** PHP 8.2 – csak olvasható osztályok, diszjunktív normálforma típusok;
- **2023:** PHP 8.3 – típusos osztálykonstansok, `Override` attribútum;
- **2024:** PHP 8.4 – property hookok, aszimmetrikus láthatóság.

A PHP 8.x sorozat számos újítást hozott. Ezek jelentősen javították a teljesítményt, a kód olvashatóságát és a típusbiztonságot. A legfontosabb 8.x újdonságok:

- **JIT-fordító (8.0):** jelentősen gyorsabb kódvégrehajtás;
- **Név szerinti argumentumok (named arguments, 8.0):** a függvényparaméterek név szerinti megadása;
- **Match kifejezés (8.0):** a `switch` modern alternatívája;

- **Felsorolások (enum) (8.1):** előre definiált értékkészletek kezelése;
- **Csak olvasható (`readonly`) tulajdonságok (8.1):** olyan objektumtulajdonságok, amelyek az első értékadás után módosíthatatlanná válnak, így biztosítva az objektum állapotának integritását;
- **Típusos osztálykonstansok (8.3):** az osztálykonstansokhoz explicit típus rendelhető, erősítve a típusbiztonságot.

Ezek a fejlesztések jól mutatják, hogy a PHP folyamatosan fejlődik és alkalmazkodik a modern webfejlesztés kihívásaihoz. A nyelv egyre inkább a típusbiztonságra, a teljesítményre és a fejlesztői élmény javítására összpontosít, miközben megőrzi egyik fő erősségét: a könnyű tanulhatóságot és használhatóságot.

1.2.2. A PHP főbb jellemzői

- **Szerveroldali végrehajtás:** A PHP-kód a szerveren fut, a kliens csak az eredményt kapja meg.
- **Beágyazhatóság HTML-be:** A PHP-kód közvetlenül integrálható HTML-dokumentumokba.
- **Platformfüggetlenség:** A PHP több operációs rendszeren és webszerveren fut.
- **Széles körű adatbázis-támogatás:** Támogatja a népszerű adatbáziskezelő rendszereket (MySQL, PostgreSQL, SQLite, Oracle stb.).
- **Gazdag beépített függvénykészlet:** Számos függvény áll rendelkezésre (pl. sztringkezelés, fájlműveletek, hálózatkezelés).
- **Dinamikus típuskezelés:** A változók típusát nem kell előre megadni, a PHP automatikusan kezeli a típuskonverziót.
- **Több paradigmát támogató nyelv:** Támogatja a procedurális, objektumorientált és funkcionális programozást is.
- **Nagy és aktív közösség:** Rengeteg könyvtár, keretrendszer és erőforrás érhető el.
- **Nyílt forráskód:** A PHP ingyenesen használható és fejleszthető.

1.2.3. A PHP összehasonlítása más szerveroldali technológiákkal

A szerveroldali webfejlesztésre több nyelv és platform is alkalmas. Az alábbiakban a PHP-t a leggyakoribb alternatívákkal hasonlítjuk össze. A cél nem a „győztes" kihirdetése, hanem annak bemutatása, hogy az egyes technológiák milyen területeken erősek.

PHP és Node.js. A Node.js JavaScript-alapú, eseményvezérelt platform, amely különösen alkalmas valós idejű alkalmazásokhoz (pl. csevegőrendszerek, valós idejű értesítések). A PHP ezzel szemben elsősorban a hagyományos kérés–válasz alapú webalkalmazások fejlesztésében terjedt el, és egyszerűen integrálható HTML-lel és adatbázis-kezeléssel. Mindkét platform modern, aktívan fejlesztett ökoszisztémával rendelkezik.

PHP és Python. A Python általános célú programozási nyelv, amelyet adattudományban, gépi tanulásban és automatizálásban is széles körben alkalmaznak. Webfejlesztésre a Django és a Flask keretrendszerek révén szintén alkalmas, az utóbbi években pedig a FastAPI keretrendszer is jelentős teret nyert. A PHP fő felhasználási területe a szerveroldali webfejlesztés, ezért a HTTP-kérések kezelése, az űrlapfeldolgozás és az adatbázis-műveletek közvetlenül kapcsolódnak a nyelv tipikus használatához.

PHP és Java. A Java nagyvállalati rendszerekben elterjedt, erősen típusos nyelv, amely komplex architektúrák megvalósítására is alkalmas. A PHP-hez képest a Java-alapú webfejlesztés általában összetettebb projektstruktúrát és konfigurációt használ. PHP-ben ugyanakkor már néhány sornyi kóddal készíthető egyszerű dinamikus weboldal, ami oktatási szempontból előnyös lehet.

A PHP jó választás a webprogramozás tanulásához, mert egyszerűen szemlélteti a webes alapfogalmak működését, például a kérések és válaszok kezelését vagy az adatbázisok használatát. Emellett számos népszerű rendszer és keretrendszer, például a WordPress és a Laravel (Stauffer 2023) is PHP-re épül.

1.3. A fejlesztői környezet beállítása

A PHP-fejlesztéshez szükség van egy webszerverre, PHP-értelmezőre, valamint általában egy adatbázis-kezelő rendszerre is. Bár ezek külön-külön is telepíthetők, léteznek kényelmes, előre összeállított csomagok, amelyek minden szükséges komponenst tartalmaznak a fejlesztéshez.

A könyv kódpéldái a PHP 8.x sorozathoz készültek. Futtatásukhoz a PHP 8.2 vagy újabb verzió szükséges, ajánlott a 8.3 használata. Néhány bemutatott nyelvi elem csak a 8.3 vagy a 8.4 verziótól érhető el. Ezt a szöveg az adott helyen külön jelzi.

1.3.1. XAMPP

A XAMPP egy ingyenes, nyílt forráskódú, platformfüggetlen webszervercsomag. Kezdők számára ez az egyik legkönnyebben telepíthető megoldás. A név egy betűszó: **X** (cross-platform) + **A**pache + **M**ariaDB + **P**HP + **P**erl.

A XAMPP telepítése és használata:

1. Töltsük le a XAMPP-ot a hivatalos weboldalról: https://www.apachefriends.org/.
2. Futtassuk a telepítőt, és kövessük az utasításokat. Válasszuk ki a kívánt komponenseket (általában az alapértelmezett beállítások megfelelőek).
3. A telepítés után indítsuk el a XAMPP Control Panelt.
4. Indítsuk el az Apache- és a MySQL-szolgáltatásokat a Control Panelben.
5. Nyissuk meg a böngészőben a `http://localhost` címet a telepítés teszteléséhez. Ha minden rendben van, egy üdvözlő oldalt látunk.
6. A PHP-fájlokat a „htdocs” mappába kell helyezni (általában `C:\xampp\htdocs` Windowson, illetve `/opt/lampp/htdocs` Linuxon).

1.3.2. PHP beépített fejlesztői szerver

A PHP 5.4 óta elérhető egy beépített fejlesztői webszerver, amely XAMPP telepítése nélkül, azonnal használható fejlesztési célokra:

```
php -S localhost:8000
```

A parancs futtatása után a projekt a `http://localhost:8000` címen érhető el. A szerver a terminálban a `Ctrl+C` billentyűkombinációval állítható le.

Megjegyzés

A beépített PHP-szerver fejlesztési célokra alkalmas, éles (production) környezetben azonban nem használható. Egyszerre egy kérés kiszolgálására képes.

1.3.3. Docker

Haladó fejlesztők számára a *Docker* konténerizációs platform lehetővé teszi, hogy a teljes fejlesztői környezet (PHP, adatbázis, webszerver) hordozható és reprodukálható módon legyen definiálva. Ez egy alternatíva azoknak,

akik már ismerik a Docker alapjait. Windows-rendszeren a Docker Desktop[2] telepítése szükséges, Linux- és macOS-rendszeren a Docker Engine közvetlenül telepíthető a csomagkezelőn keresztül. A Docker részletes tárgyalása meghaladja e jegyzet kereteit.

1.3.4. Kódszerkesztő választása

A PHP-kód írásához megfelelő szerkesztőre van szükség. Az egyszerű szövegszerkesztőktől az összetett fejlesztői környezetekig számos lehetőség áll rendelkezésre. Kezdők számára három kategóriát mutatunk be:

Visual Studio Code

- ingyenes és könnyen használható;
- kiváló PHP-támogatás bővítményekkel;
- automatikus kódkiegészítés és hibakeresési lehetőségek;
- integrált terminál és Git-támogatás;
- nagy bővítmény-ökoszisztéma.

Gyors beállítás:

1. Töltsük le a telepítőt a `https://code.visualstudio.com/` címről.
2. Telepítsük a „PHP IntelliSense” bővítményt.
3. (Opcionálisan) telepítsük a „PHP Debug” bővítményt.
4. Hozzunk létre egy `.php` fájlt, és elkezdhetünk kódolni.

Egyszerű szövegszerkesztők

Azok számára, akik minimális környezetet preferálnak:

- **Notepad++ (Windows):** könnyű, gyors, szintaxiskiemeléssel;
- **Sublime Text:** elegáns, testreszabható, többplatformos.

Professzionális IDE-k

Haladó fejlesztőknek vagy nagyobb projektekhez:

- **PhpStorm:**[3] kifejezetten PHP-fejlesztésre tervezett IDE, fejlett hibakereséssel; hallgatók és oktatók számára ingyenes licenccel elérhető;
- **NetBeans:**[4] ingyenes, nyílt forráskódú IDE, jó PHP-támogatással.

[2] `https://www.docker.com/products/docker-desktop/`
[3] `https://www.jetbrains.com/phpstorm/`
[4] `https://netbeans.apache.org/`

1.3.5. Fejlesztői környezet tesztelése

A fejlesztői környezet beállítása után érdemes ellenőrizni, hogy minden megfelelően működik-e. Ehhez hozzunk létre egy egyszerű PHP-fájlt:

```php
<?php
    phpinfo();
?>
```

1.3. kódpélda. *phpinfo.php*

XAMPP tesztelése

Másoljuk a `phpinfo.php` fájlt a `C:\xampp\htdocs` mappába, majd nyissuk meg a böngészőben a `http://localhost/phpinfo.php` címet.

PHP beépített szerver tesztelése

Navigáljunk a projekt mappájába a terminálban, indítsuk el a szervert a `php -S localhost:8000` paranccsal, majd nyissuk meg a `http://localhost:8000/phpinfo.php` címet.

Sikeres teszt

Ha minden megfelelően működik, egy részletes információs oldalt láthatunk a PHP verziójáról, a telepített modulokról, a konfigurációs beállításokról és a szerver adatairól.

1.4. Bemutató PHP-program

Ez a program bemutatja a PHP néhány alapvető funkcióját. Hozzunk létre egy `bemutato.php` fájlt a `htdocs` mappában (XAMPP esetén), vagy a projekt könyvtárában, ha a beépített szervert használjuk.

```php
<!DOCTYPE html>
<html lang="hu">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width,
        initial-scale=1.0">
    <title>Bemutató PHP program</title>
</head>
<body>
    <h1>Üdvözöl a PHP!</h1>

    <?php
        // Ez egy egysoros megjegyzés
```

```
        /*
        Ez egy többsoros
        megjegyzés blokk
        */

        // Változók és echo használata
        $name = "Felhasználó";
        $age = 25;
        echo "<p>Helló, $name! {$age} éves vagy.</p>";

        // Feltételes utasítás
        if ($age >= 18) {
            echo "<p>Felnőtt vagy.</p>";
        } else {
            echo "<p>Kiskorú vagy.</p>";
        }

        // Tömbök
        $fruits = ["alma", "körte", "banán", "narancs"];
        echo "<p>A kedvenc gyümölcsöm: " . $fruits[2] .
            "</p>";

        // Ciklus
        echo "<ul>";
        foreach ($fruits as $fruit) {
            echo "<li>$fruit</li>";
        }
        echo "</ul>";

        // Függvény definiálása és használata
        function add($a, $b) {
            return $a + $b;
        }

        $result = add(5, 3);
        echo "<p>5 + 3 = $result</p>";

        // Dátum és idő
        echo "<p>Mai dátum: " . date("Y-m-d") . "</p>";
        echo "<p>Jelenlegi idő: " . date("H:i:s") . "</p>";
    ?>
</body>
</html>
```

1.4. kódpélda. *bemutato.php*

Ebben a példában a PHP számos alapvető elemét láthatjuk:

- **PHP-címkék (tagek):** a PHP-kód a `<?php` nyitó és a `?>` záró címke közé kerül;
- **Megjegyzések:** egysoros (`//`) és többsoros (`/* */`) megjegyzések használata;
- **Változók:** a `$` jellel jelöljük őket, és dinamikus típusúak;
- **Kiírás (echo):** adatok megjelenítése a kimeneten;
- **Sztringek:** idézőjelek között helyezkednek el, és a változók közvetlenül beágyazhatók;
- **Feltételes utasítások:** `if-else` szerkezet használata;
- **Tömbök:** a `[]` szintaxissal hozhatók létre (korábban: `array()`);
- **Ciklusok:** a `foreach` ciklus tömbök bejárására;
- **Függvények:** saját függvények definiálása és használata;
- **Beépített függvények:** például a `date()` függvény használata.

1.5. Kódolási konvenciók

A kódolási konvenciók egységes szabályrendszert biztosítanak a kód írásához. Betartásuk javítja a kód olvashatóságát, karbantarthatóságát, és megkönnyíti a csapatmunkát. A PHP-közösség a PSR-ajánlásokat (PHP Standards Recommendations) tekinti irányadónak.[5]

1.5.1. PSR-1: alapvető kódolási szabvány

A PSR-1 az alapvető kódolási szabályokat határozza meg:

- **PHP-címkék:** csak a `<?php` vagy a rövid `<?=` nyitó címke használható;
- **Fájlkódolás:** UTF-8, BOM nélkül;
- **Osztálynevek:** `PascalCase` (StudlyCaps) formátum;
- **Konstansok:** nagybetűs, aláhúzással elválasztott formátum (`UPPER_CASE`);
- **Metódusnevek:** `camelCase` formátum.

[5] https://www.php-fig.org/psr/

```php
<?php

// Osztálynév: PascalCase
class UserRepository
{
    // Konstans: UPPER_CASE
    public const MAX_LOGIN_ATTEMPTS = 5;

    // Metódusnév: camelCase
    public function findById(int $id): ?User
    {
        // ...
    }

    public function getUserByEmail(string $email): ?User
    {
        // ...
    }
}
```

1.5. kódpélda. *PSR-1 példák*

1.5.2. PSR-12: kiterjesztett kódolási stílus

A PSR-12 a PSR-1-re épül, és részletesebb formázási szabályokat határoz meg.

Behúzás és sorhossz

- 4 szóközös behúzás (tabulátor használata nem javasolt);
- ajánlott maximális sorhossz: 80 karakter, felső korlát: 120 karakter;
- a fájl végén egy üres sor legyen.

Kapcsos zárójelek elhelyezése

```php
<?php

// Osztályoknál és metódusoknál: új sorban
class UserService
{
    public function createUser(string $name, string $email):
        User
    {
        // Vezérlési szerkezeteknél: azonos sorban
```

```
        if ($this->userExists($email)) {
            throw new UserExistsException();
        }

        foreach ($this->validators as $validator) {
            $validator->validate($name);
        }

        return new User($name, $email);
    }
}
```

1.6. kódpélda. *Kapcsos zárójelek elhelyezése*

Szóközök használata

```
<?php

// Vezérlési szerkezetek után szóköz
if ($condition) {
    // ...
}

foreach ($items as $key => $value) {
    // ...
}

// Függvényhívásnál nincs szóköz
$result = calculateTotal($price, $quantity);

// Operátorok körül szóköz
$total = $price * $quantity;
$name = $firstName . ' ' . $lastName;
```

1.7. kódpélda. *Szóközök használata*

1.5.3. Elnevezési konvenciók

Az egységes elnevezés kulcsfontosságú az olvasható kód szempontjából. A legfontosabb konvenciókat az 1.1. táblázat foglalja össze:

1.1. táblázat. *Elnevezési konvenciók PHP-ben*

Elem	Konvenció	Példa
Osztály	PascalCase	`UserController`
Metódus	camelCase	`getUserById()`
Változó	camelCase	`$userName`
Konstans	UPPER_CASE	`MAX_SIZE`
Privát tulajdonság	camelCase	`$connectionPool`

Megjegyzés

Beszédes nevek. Használjunk leíró neveket, amelyekből egyértelműen kiderül a változó vagy függvény célja:

- nem ajánlott: `$d`, `$tmp`, `process()`;
- ajánlott: `$age`, `$userName`, `calcTotal()`.

1.5.4. Fájlszervezési konvenciók

A PSR-4-szabvány szerint minden osztályt külön fájlban kell elhelyezni, és a fájl nevének meg kell egyeznie az osztály nevével:

- `UserService.php` → egy `UserService` osztályt tartalmaz;
- `ProductRepository.php` → egy `ProductRepository` osztályt tartalmaz.

A PSR-4-autoloading lehetővé teszi, hogy az osztályok automatikusan betöltődjenek `require` vagy `include` használata nélkül. Ennek működését a Composer csomagkezelővel együtt részletesen tárgyaljuk.

1.5.5. Típusdeklarációk

A PHP 7-től kezdődően elérhetők a típusdeklarációk, amelyek jelentősen javítják a kód megbízhatóságát:

```
<?php

declare(strict_types=1);

// Paraméter és visszatérési típus
function add(int $a, int $b): int
{
    return $a + $b;
```

```
}

// Nullable típus: null is visszaadható
function findUser(int $id): ?string
{
    if ($id === 1) {
        return "Admin";
    }
    return null;
}

// Union type (PHP 8.0+)
function formatPrice(int|float $price): string
{
    return number_format($price, 2) . " RON";
}
```

1.8. kódpélda. *Típusdeklarációk használata*

Megjegyzés

A `declare(strict_types=1)` direktíva szigorú típusellenőrzést alkalmaz a függvények skaláris típusokkal deklarált bemeneti paramétereire és visszatérési értékeire. Nem kapcsolja ki azonban a PHP automatikus típuskonverzióját más műveletek, például az operátorok használata során. Használata ajánlott minden PHP-fájl elején.

1.5.6. Dokumentációs megjegyzések (PHPDoc)

A PHPDoc megjegyzések segítik a kód dokumentálását, valamint az IDE-k intelligens kódkiegészítését:

```
<?php

/**
 * Felhasználói fiókokat kezelő szolgáltatás.
 *
 * Ez az osztály felelős a felhasználói fiókok
    létrehozásáért,
 * módosításáért és törléséért.
 */
class UserService
{
    /**
     * Felhasználó keresése e-mail-cím alapján.
```

```
     *
     * @param string $email A keresett e-mail-cím
     * @return User|null A megtalált felhasználó vagy null
         érték
     * @throws InvalidEmailException Hibás e-mail-formátum
         esetén
     */
    public function findByEmail(string $email): ?User
    {
        // ...
    }

    /**
     * Új felhasználó létrehozása.
     *
     * @param array<string, mixed> $data Felhasználói adatok
     * @return User A létrehozott felhasználó
     */
    public function create(array $data): User
    {
        // ...
    }
}
```

1.9. kódpélda. *PHPDoc megjegyzések*

2. FEJEZET

A PHP-NYELV ELEMEI

2.1. Szintaxis, változók, adattípusok

2.1.1. PHP-kód szintaxisa és beágyazása HTML-be

A PHP-kód (PHP Group 2026) gyakran HTML-dokumentumokba ágyazva jelenik meg, amely lehetővé teszi dinamikus tartalom előállítását weboldalakon. A PHP-kód a `<?php` nyitó és a `?>` záró címkék között helyezkedik el.

- **Standard PHP-címke (ajánlott):** `<?php ... ?>`, amely minden PHP-környezetben támogatott.
- **Rövid echo címke:** `<?= ... ?>`, amely PHP 5.4 verziótól kezdve alapértelmezetten elérhető, és gyakran használják HTML-sablonokban rövid kimenetek kiírására.

```
<?php
// Ide kerül a PHP kód
?>
```

2.1. kódpélda. *PHP-kód-határolók*

Az alábbi példa bemutatja a PHP-kód HTML-be ágyazásának egy egyszerű módját:

```
<!DOCTYPE html>
<html>
<head>
    <title>Üdvözlő oldal</title>
</head>
<body>
    <h1>Üdvözöljük!</h1>
    <?php
    $name = "Kovács Anna";
    $age = 25;
    ?>
```

```
    <!-- Standard echo -->
    <?php echo "<p>Helló, $name!</p>"; ?>

    <!-- Rövid echo tag (ajánlott HTML-ben) -->
    <p>Kor: <?= $age ?> év</p>
</body>
</html>
```

2.2. kódpélda. *PHP-kód beágyazása HTML-be*

A PHP-utasításokat pontosvesszővel (`;`) kell lezárni. Egy sorban több utasítás is szerepelhet, azonban az olvashatóság érdekében ajánlott az egyes utasításokat külön sorba írni.

2.1.2. Megjegyzések

A megjegyzések a kód magyarázatára és dokumentálására szolgálnak. A PHP-értelmező figyelmen kívül hagyja őket. Három szintaxis áll rendelkezésre:

- `//` – egysoros megjegyzés, a sor végéig tart;
- `#` – egysoros megjegyzés, Perl-stílusú alak;
- `/* ... */` – többsoros megjegyzés.

```
// Ez egy egysoros megjegyzés
# Ez szintén egysoros megjegyzés

/*
 * Ez egy többsoros megjegyzés.
 * Több sorban is folytatható.
 */

$x = 5; // Megjegyzés sor végén is elhelyezhető
```

2.3. kódpélda. *Megjegyzések PHP-ben*

2.1.3. Változók

A PHP-ben a változókat a $ (dollárjel) előzi meg. A változónevek kis- és nagybetű-érzékenyek, és csak betűvel vagy alulvonással kezdődhetnek, amelyet betűk, számok és alulvonások követhetnek. A következő példa bemutatja a változók deklarálását és használatát:

```php
$name = "Kovács Anna";
$age = 30;
$isActive = false;

echo $name;  // Kimenet: Kovács Anna
echo $Name;  // Figyelmeztetés: Undefined variable $Name
```

2.4. kódpélda. *Változók deklarálása és használata*

A PHP dinamikus és gyengén típusos nyelv, ezért egy változó típusa a program futása során megváltozhat. A nyelv szükség esetén automatikusan típuskonverziót végez. Ezt szemlélteti a következő példa:

```php
$var = "42";
echo $var + 8;  // Kimenet: 50 (a sztring automatikusan int
    típussá konvertálódik)

$var = "Helló";
// $var = $var + 1;  // PHP 8-ban TypeError kivételt dob:
// a nem numerikus sztring nem alakítható számmá
```

2.5. kódpélda. *Dinamikus típuskezelés és automatikus konverzió*

Ez a viselkedés növeli a rugalmasságot, ugyanakkor potenciális hibaforrás is lehet, ezért fontos ismerni a típuskonverzió szabályait.

A PHP-ben a kis- és nagybetű-érzékenység az egyes nyelvi elemek esetében eltérő:

- **Változónevek:** Kis- és nagybetű-érzékenyek.
- **Függvénynevek:** Nem kis- és nagybetű-érzékenyek.
- **Konstansok:** Kis- és nagybetű-érzékenyek.
- **Kulcsszavak** (pl. `if`, `else`, `while`): Nem kis- és nagybetű-érzékenyek.

```php
$name = "Kovács Anna";
$Name = "Nagy Béla";

echo $name; // Kimenet: Kovács Anna
echo $Name; // Kimenet: Nagy Béla

function greet() {
    echo "Helló!";
}
```

```php
GREET(); // Működik, mert a függvénynevek nem kis- és
    nagybetű-érzékenyek

define("MAX_AGE", 120);

echo MAX_AGE; // Működik
// echo max_age; // PHP 8-ban Error kivételt eredményez
```

2.6. kódpélda. *Kis- és nagybetű-érzékenység példája*

A következetes névkonvenciók alkalmazása segít elkerülni az ebből adódó hibákat.

2.1.4. PHP-adattípusok

A PHP dinamikus és gyengén típusos nyelv. Ez azt jelenti, hogy egy változó típusa a hozzárendelt érték alapján automatikusan meghatározódik, és a típusok között szükség esetén automatikus konverzió történhet. A PHP-ben a következő alaptípusok léteznek:

1. **Integer** (**egész szám**): Előjeles egész számok; értéktartományuk platformfüggő, 32 bites rendszereken általában −2 147 483 648 és 2 147 483 647 közé esik.
2. **Float** (**lebegőpontos szám**): Tizedes számok; pontosságuk platformfüggő, általában körülbelül 14 tizedesjegyig tekinthetők megbízhatónak.
3. **String** (**karakterlánc**): Szöveges adatok; hosszát alapvetően csak a rendelkezésre álló memória korlátozza.
4. **Boolean** (**logikai érték**): Igaz (`true`) vagy hamis (`false`) értéket vehet fel.
5. **Array** (**tömb**): Rendezett kulcs–érték párok gyűjteménye.
6. **Object** (**objektum**): Osztályok példányai.
7. **NULL**: Speciális érték, amely a „nincs érték” állapotot jelöli.
8. **Resource** (**erőforrás**): Külső erőforrásokra (pl. fájlokra, adatbázis-kapcsolatokra) mutató speciális típus.

Az alábbi példa bemutatja a felsorolt adattípusok használatát:

```php
$age = 30;                          // Integer
$height = 1.78;                     // Float
$name = "Kovács Anna";              // Sztring
```

```php
$isActive = true;                       // Boolean
$roles = ["admin", "user"];             // Array
$deletedAt = null;                      // NULL

class User {
    public string $name;
}

$user = new User();                     // Object

$file = fopen(__FILE__, "r");           // Resource
```

2.7. kódpélda. *PHP-alaptípusok*

2.1.5. Konstansok

A konstansok olyan azonosítók, amelyek értéke a program futása során nem változtatható meg. Konstansokat kétféleképpen lehet definiálni:

1. a `define()` függvénnyel;
2. a `const` kulcsszóval.

Az alábbi példa bemutatja mindkét módszer használatát:

```php
define("PI", 3.14);
const MAX_VALUE = 100;

echo PI;         // Kimenet: 3.14
echo MAX_VALUE; // Kimenet: 100
```

2.8. kódpélda. *Konstansok definiálása*

A `define()` függvény és a `const` kulcsszó közötti főbb különbségek:

- A `define()` futás közben értékelődik ki, míg a `const` a kód feldolgozásának korábbi szakaszában, ezért csak konstans kifejezésekkel használható.
- A `define()` a kód bármely részén használható, míg a `const` csak a legfelső szinten vagy osztályokon belül.
- A `define()` dinamikus névvel és értékkel is használható, míg a `const` esetében ez nem lehetséges.

2.1.6. Kiíratás a kimenetre

A PHP több módszert kínál adatok kiíratására. A leggyakrabban használt módszerek a következők:

- **echo**: Több érték kiírására is használható.
- **print**: Egyetlen érték kiírására szolgál.
- **printf**: Formázott szöveg kiírására alkalmas.
- **print_r**: Strukturált kiíratásra használható (pl. tömbök esetén).
- **var_dump**: Részletes típus- és értékinformációt ad.

Az alábbi példa bemutatja ezeket a kiíratási módszereket:

```php
$name = "Kovács Anna";
$age = 30;

// echo - több értéket is kiírhatunk vesszővel elválasztva
echo "Név: ", $name, ", Kor: ", $age, "<br>";

// print - csak egy értéket ír ki
print "Helló, $name!<br>";

// printf - formázott kiírás
printf("Név: %s, Kor: %d<br>", $name, $age);

// print_r - strukturált kiíratás (hasznos tömbök és
    objektumok esetén)
$roles = ["admin", "editor", "viewer"];
print_r($roles);

// var_dump - részletes információt ad a változóról
var_dump($name);
```

2.9. kódpélda. *Különböző kiíratási módszerek*

Az `echo` és a `print` közötti főbb különbségek:

- Az `echo` nem ad vissza értéket, míg a `print` mindig 1-et ad vissza.
- Az `echo` képes több paraméter fogadására, míg a `print` csak egyet fogad.
- Az `echo` általában gyorsabb, különösen több érték kiírásakor.

2.1.7. Típusokkal kapcsolatos függvények

A PHP számos beépített függvényt kínál a változók típusának ellenőrzésére és kezelésére. A legfontosabb függvénycsoportok a következők:

- **Típusellenőrzés:** `is_int()`, `is_float()`, `is_string()`, `is_array()`, `is_object()`, `is_null()`, `is_resource()`;
- **Típuskonverzió:** `intval()`, `floatval()`, `strval()`;
- **Típus lekérdezése:** `gettype()`;
- **Típus beállítása:** `settype()`.

Az alábbi példa bemutatja ezen függvények használatát:

```php
$value = "42";

// Típus ellenőrzése
var_dump(is_string($value));  // bool(true)
var_dump(is_int($value));     // bool(false)
var_dump(is_numeric($value)); // bool(true)

// Típuskonverzió
$intValue = intval($value);
var_dump($intValue); // int(42)

// Típus lekérdezése
echo gettype($value); // string

// Változó típusának beállítása
settype($value, "integer");
var_dump($value); // int(42)
```

2.10. kódpélda. *Típusellenőrző és típuskezelő függvények*

A PHP-ben ezek a függvények különösen hasznosak hibakeresés során, illetve olyan esetekben, amikor egy művelet végrehajtása előtt meg kell győződni egy változó típusáról.

2.2. Karakterláncok kezelése

A karakterlánc (*string*, magyar szakszövegben *sztring* alakban is) a PHP egyik leggyakrabban használt adattípusa. A PHP rugalmas és sokoldalú eszközkészletet biztosít a karakterláncok kezelésére, amely lehetővé teszi a szöveges adatok hatékony feldolgozását.

2.2.1. Karakterláncok megadási módjai

PHP-ben négyféle módon adhatunk meg karakterláncokat. Az egyes formátumok eltérő viselkedéssel rendelkeznek, különösen a változók és az escape karakterek feldolgozása szempontjából:

1. **Aposztrófok (`'...'`)**: Egyszerű karakterláncok; a tartalom literálként értelmeződik, a változók és az escape szekvenciák nem kerülnek feldolgozásra.
2. **Idézőjelek (`"..."`)**: A PHP értelmezi a változókat és az escape karaktereket; dinamikus szövegek létrehozására alkalmas.
3. **`Heredoc`**: Többsoros karakterláncok megadására szolgál; az idézőjeles formához hasonlóan értelmezi a változókat és az escape karaktereket.
4. **`Nowdoc`**: Többsoros karakterláncok megadására szolgál; literálként kezeli a tartalmat változó- és escape-karakterfeldolgozás nélkül.

Alapvető példák

```php
$name = "Kovács Anna";
$age = 30;

// Aposztrófok - változók NEM értékelődnek ki
echo 'Helló $name, $age éves vagy.';
// Kimenet: Helló $name, $age éves vagy.

// Idézőjelek - változók kiértékelődnek
echo "Helló $name, $age éves vagy.";
// Kimenet: Helló Kovács Anna, 30 éves vagy.

// Heredoc - többsoros, változók kiértékelődnek
echo <<<EOT
Helló $name,
$age éves vagy.
Ez egy többsoros karakterlánc.
EOT;
// Kimenet:
// Helló Kovács Anna,
// 30 éves vagy.
// Ez egy többsoros karakterlánc.

// Nowdoc - többsoros literál
echo <<<'EOT'
Helló $name,
$age éves vagy.
A változók itt nem értékelődnek ki.
EOT;
```

```
// Kimenet:
// Helló $name,
// $age éves vagy.
// A változók itt nem értékelődnek ki.
```

2.11. kódpélda. *Karakterláncok megadási módjai*

Fontos megjegyezni, hogy az aposztrófos és a `nowdoc` szintaxis általában gyorsabb feldolgozást eredményez, mivel a PHP ilyenkor nem végez változó- és escape-karakterkiértékelést. Ugyanakkor ez a teljesítménybeli különbség a gyakorlatban általában csak nagy mennyiségű szöveg feldolgozása esetén válik mérhetővé.

Változók behelyettesítése összetett kifejezésekben

Tömbök és objektumok elemei kapcsos zárójelek segítségével illeszthetők karakterláncba, így a PHP-értelmező egyértelműen felismeri a változó határait:

```
$fruit = "alma";
$count = 5;

// Egyszerű változók (kapcsos zárójelek nélkül is működik)
echo "$count darab $fruit van a kosárban.";
// Kimenet: 5 darab alma van a kosárban.

// Tömbelem behelyettesítése kapcsos zárójelekkel
$fruits = ['alma', 'körte', 'szilva'];
echo "A kedvenc gyümölcsöm: {$fruits[0]}";
// Kimenet: A kedvenc gyümölcsöm: alma

// Objektumtulajdonság behelyettesítése
$obj = new stdClass();
$obj->name = "Kovács Anna";

echo "A nevem: {$obj->name}";
// Kimenet: A nevem: Kovács Anna

// Kifejezések esetén összefűzést használunk
echo "Gyümölcsök száma: " . ($count + 1);
// Kimenet: Gyümölcsök száma: 6
```

2.12. kódpélda. *Összetett változóbehelyettesítés*

2.2.2. Escape karakterek

Az escape karakterek speciális jelentéssel rendelkező karakterek a karakterláncokban, amelyek lehetővé teszik olyan karakterek megadását, amelyeket

közvetlenül nem vagy csak nehezen lehetne beírni. PHP-ben az escape karakter a backslash (\).

Gyakori escape szekvenciák

- `\n` – új sor;
- `\r` – kocsivissza (carriage return);
- `\t` – tabulátor;
- `\$` – dollárjel (a változóbehelyettesítés elkerülésére);
- `\"` – idézőjel idézőjeles karakterláncban;
- `\'` – aposztróf aposztrófos karakterláncban;
- `\\` – backslash.

Fontos megjegyezni, hogy aposztrófos karakterláncokban (`'...'`) csak a `\'` és a `\\` escape szekvenciák kerülnek értelmezésre. Ezzel szemben idézőjeles karakterláncokban (`"..."`) a PHP a legtöbb escape szekvenciát feldolgozza.

Példák escape karakterekre

```php
$name = "Kovács Anna";

// Idézőjeles karakterláncban az escape szekvenciák
    értelmeződnek
echo "Új sor:\nTabulátor:\tBackslash:\\";
// Kimenet: Új sor:(új sor)Tabulátor:(tab)Backslash:\

// Aposztrófos karakterláncban az escape szekvenciák
    többsége nem értelmeződik
echo 'Új sor:\nTabulátor:\tBackslash:\\';
// Kimenet: Új sor:\nTabulátor:\tBackslash:\

// Változóbehelyettesítés elkerülése
echo "A nevem: \$name";  // Kimenet: A nevem: $name
echo "A nevem: $name";   // Kimenet: A nevem: Kovács Anna

// Idézőjelek használata karakterláncban
echo "Azt mondta: \"Helló!\"";  // Kimenet: Azt mondta:
    "Helló!"
echo 'Azt mondta: "Helló!"';    // Kimenet: Azt mondta:
    "Helló!"
```

2.13. kódpélda. *Escape karakterek használata*

Az escape karakterek helyes használata fontos a karakterláncok pontos formázásához és a nem kívánt értelmezések elkerüléséhez, különösen változóbehelyettesítés és speciális karakterek használata esetén.

2.2.3. Karakterlánc-műveletek és hasznos függvények

A PHP gazdag eszközkészletet biztosít a karakterláncok kezelésére. A leggyakrabban használt műveletek és függvények a következők:

- **Összefűzés és módosítás:** összefűzés-operátor (`.`), hozzáfűzés-operátor (`.=`), `str_replace()`, `substr()`;
- **Hossz és indexelés:** `strlen()`, indexelés (`[]`);
- **Keresés és összehasonlítás:** `strpos()`, `strcmp()`;
- **Formázás és átalakítás:** `strtolower()`, `strtoupper()`, `ucfirst()`, `ucwords()`, `trim()`, `sprintf()`;
- **Szétbontás és összefűzés:** `explode()`, `implode()`;
- **Reguláris kifejezések:** `preg_match()`.

Az alábbi példák bemutatják a fenti műveletek és függvények használatát:

```
// Összefűzés és módosítás
$str1 = "Helló";
$str2 = "világ";
$result = $str1 . " " . $str2;              // Helló világ

$str = "Helló";
$str .= " világ";                           // Helló világ

$newStr = str_replace("világ", "PHP", $str); // Helló PHP
$sub = substr($str, 0, 4);                  // "Hell"
$mbSub = mb_substr($str, 0, 5);             // "Helló"

// Hossz és indexelés
$char = $str[0];                            // 'H'
$length = strlen($str);                     // 13 (bájtok
    száma)
$mbLength = mb_strlen($str);                // 11 (karakterek
    száma)

// Keresés és összehasonlítás
$position = strpos($str, "világ");          // 7 (bájtpozíció)
$comparison = strcmp("alma", "körte");      // negatív érték
```

```php
// Formázás és átalakítás
echo strtolower("HELLO");                    // hello
echo strtoupper("hello");                    // HELLO

$trimmed = trim("  Helló  ");                // "Helló"

$formatted = sprintf("Név: %s, Kor: %d", "Kovács Anna", 30);
// Név: Kovács Anna, Kor: 30

// Szétbontás és egyesítés
$parts = explode(" ", $str);                 // ["Helló",
    "világ"]
$joined = implode(", ", $parts);             // "Helló, világ"

// Reguláris kifejezés
$pattern =
    "/^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\\.[a-zA-Z]{2,}$/";
$email = "anna@example.ro";

if (preg_match($pattern, $email)) {
    echo "Érvényes e-mail cím";
}
```

2.14. kódpélda. *Karakterlánc-műveletek és függvények példái*

Megjegyzés

A `strtolower()` és a `strtoupper()` függvények csak az ASCII-karaktereket kezelik helyesen. Ékezetes karakterek (pl. á, é, ő) nagybetűsítéséhez vagy kisbetűsítéséhez az `mb_strtolower()` és az `mb_strtoupper()` függvényeket kell használni, amelyek figyelembe veszik a karakterkódolást (pl. UTF-8). Ugyanez igaz a `strlen()`, a `substr()` és a `strpos()` függvényre is. Ezek bájtokkal számolnak, egy ékezetes betű pedig UTF-8-ban két bájt. Ékezetes szövegnél ezért az `mb_strlen()`, az `mb_substr()` és az `mb_strpos()` adja a karakterek szerinti eredményt, ahogy a példában is látható. A példa végén szereplő `preg_match()` függvény reguláris kifejezés alapján keres mintát a sztringben, részletesen a 4.5.4. alfejezetben mutatjuk be.

Formázott karakterláncok: `sprintf()`

A `sprintf()` függvény formázott karakterláncot hoz létre, és visszaadja azt. Ezzel szemben a `printf()` közvetlenül kiírja az eredményt. Webes alkalmazásokban a `sprintf()` gyakran hasznosabb, mivel a visszaadott érték

tovább feldolgozható vagy eltárolható.

Szintaxis

```
sprintf(string $format, mixed ...$values): string
```

Paraméterek:

- `$format`: a formátumsztring, amely szöveget és formátumjelölőket (`%s`, `%d`, `%f` stb.) tartalmazhat;
- `$values`: a formátumjelölők helyére kerülő értékek, vesszővel elválasztva.

Visszatérési érték: a formázott karakterlánc, amelyet a függvény visszaad (nem írja ki).

A formátumsztring leggyakoribb jelölői:

```php
$price = 1499.9;
echo sprintf("Ár: %.2f RON", $price);         // Ár: 1499.90 RON

$name = "Kovács Anna";
$age  = 30;

echo sprintf("%s (%d éves)", $name, $age);  // Kovács Anna (30 éves)
echo sprintf("%05d", 42);                   // 00042
```

2.15. kódpélda. *`sprintf()` használata*

A leggyakoribb formátumjelölők: `%s` (szöveg), `%d` (egész szám), `%f` (lebegőpontos szám), `%.2f` (két tizedesjegy).

A `printf()` ugyanezt a formátumszintaxist használja, azzal a különbséggel, hogy az eredményt közvetlenül a kimenetre írja ki ahelyett, hogy visszaadná azt.

2.3. Operátorok

Az operátorok kulcsfontosságú elemei a PHP-nek, amelyek lehetővé teszik különböző műveletek végrehajtását változókon és értékeken, például számítások, összehasonlítások és logikai műveletek elvégzését. A PHP számos operátort kínál, amelyek funkciójuk alapján csoportosíthatók.

2.3.1. Aritmetikai operátorok

Az aritmetikai operátorok alapvető matematikai műveletek végrehajtására szolgálnak. Ezek az operátorok számokkal dolgoznak, és a matematikában megszokott módon működnek (2.1. táblázat).

2.1. táblázat. *Aritmetikai operátorok*

Operátor	Művelet	Példa
`+`	összeadás	`10 + 3 = 13`
`-`	kivonás	`10 - 3 = 7`
`*`	szorzás	`10 * 3 = 30`
`/`	osztás	`10 / 4 = 2.5`
`%`	maradékos osztás	`10 % 3 = 1`
`**`	hatványozás	`2 ** 8 = 256`

2.3.2. Hozzárendelő operátorok

A hozzárendelő operátorok értékeket rendelnek változókhoz. Az egyszerű értékadáson túl léteznek összetett hozzárendelő operátorok is, amelyek egy műveletet és egy hozzárendelést kombinálnak (2.2. táblázat).

2.2. táblázat. *Hozzárendelő operátorok*

Operátor	Leírás
`=`	egyszerű hozzárendelés
`+=`, `-=`, `*=`, `/=`, `%=`	aritmetikai művelet és hozzárendelés
`.=`	összefűzés és hozzárendelés

```php
$a = 5;
$a += 3; // Ekvivalens: $a = $a + 3;
echo $a; // Kimenet: 8

$str = "Helló";
$str .= " világ"; // karakterlánc összefűzés és hozzárendelés
echo $str; // Kimenet: "Helló világ"
```

2.16. kódpélda. *Hozzárendelő operátorok használata*

2.3.3. Logikai operátorok

A logikai operátorok logikai (*boolean*) értékekkel dolgoznak, és gyakran használjuk őket feltételes szerkezetekben. Ezek az operátorok lehetővé teszik összetett feltételek kialakítását (2.3. táblázat).

2.3. táblázat. *Logikai operátorok*

Operátor	Jelentés
`&&`	és (logikai AND)
`||`	vagy (logikai OR)
`!`	nem (logikai NOT)

Fontos megjegyezni, hogy a `&&` és a `||` operátorok rövidzáras (*short-circuit*) kiértékelést alkalmaznak. Ez azt jelenti, hogy amint az eredmény eldönthető, a további kiértékelés leáll.

```php
$a = true;
$b = false;

var_dump($a && $b); // Kimenet: bool(false)
var_dump($a || $b); // Kimenet: bool(true)
var_dump(!$a);      // Kimenet: bool(false)
```

2.17. kódpélda. *Logikai operátorok használata*

2.3.4. Összehasonlító operátorok

Az összehasonlító operátorok két érték összehasonlítására szolgálnak, és logikai értéket adnak vissza, amely jelzi, hogy az összehasonlítás igaz vagy hamis (2.4. táblázat).

2.4. táblázat. *Összehasonlító operátorok*

Operátor	Leírás
`==`	egyenlő
`===`	azonos
`!=`	nem egyenlő
`!==`	nem azonos
`<`	kisebb
`>`	nagyobb
`<=`	kisebb vagy egyenlő
`>=`	nagyobb vagy egyenlő

Fontos kiemelni az == és az === operátorok közötti különbséget:

- Az == (egyenlőség) operátor az értékeket hasonlítja össze, a típust figyelmen kívül hagyva.
- Az === (azonosság) operátor az értékeket és a típust is összehasonlítja.

```php
$a = 5;
$b = "5";

var_dump($a == $b);  // Kimenet: bool(true)  - az érték
    egyezik
var_dump($a === $b); // Kimenet: bool(false) - a típus
    különbözik
var_dump($a != "6"); // Kimenet: bool(true)
var_dump($a !== 5);  // Kimenet: bool(false)
```

2.18. kódpélda. *Összehasonlító operátorok használata*

Megjegyzés

Részesítsük előnyben az === operátort: A szigorú összehasonlítás megakadályozza a nem várt típuskonverziókat. Általában az === és az !== használata ajánlott az == és az != operátorokkal szemben, kivéve, ha szándékosan szeretnénk típuskonverziót alkalmazni.

2.3.5. Változók ellenőrzése

A PHP több beépített függvényt kínál változók értékének és állapotának ellenőrzésére. Ezek a függvények különösen hasznosak feltételes szerkezetekben és adatok ellenőrzése során.

- `empty()`: ellenőrzi, hogy egy változó üres-e.
- `is_null()`: ellenőrzi, hogy egy változó `null` értékű-e.
- `isset()`: ellenőrzi, hogy egy változó létezik-e és nem `null` értékű-e.

```php
$emptyString = "";
$nullValue = null;
$zeroInt = 0;
$falseBool = false;
$nonEmptyString = "Helló";
```

```php
// empty() függvény
var_dump(empty($emptyString));     // bool(true)
var_dump(empty($nullValue));       // bool(true)
var_dump(empty($zeroInt));         // bool(true)
var_dump(empty($falseBool));       // bool(true)
var_dump(empty($nonEmptyString)); // bool(false)

// is_null() függvény
var_dump(is_null($emptyString));  // bool(false)
var_dump(is_null($nullValue));    // bool(true)
var_dump(is_null($zeroInt));      // bool(false)

// isset() függvény
var_dump(isset($emptyString));    // bool(true)
var_dump(isset($nullValue));      // bool(false)
var_dump(isset($undefinedVar));   // bool(false)
```

2.19. kódpélda. *Változók ellenőrzésének használata*

Megjegyzés

Fontos különbségek:

- Az `empty()` `true` értéket ad üres karakterláncra, a `"0"` karakterláncra, `0`-ra, `false`-ra, `null`-ra és üres tömbre.
- Az `is_null()` csak akkor ad `true` értéket, ha a változó értéke pontosan `null`.
- Az `isset()` `false` értéket ad `null` érték és nem definiált változók esetén, de `true` értéket ad üres karakterlánc és `0` esetén.

2.3.6. Feltételes operátor

A feltételes (ternáris) operátor az `if-else` szerkezet tömörebb alternatívája. Ez az operátor különösen hasznos egyszerű feltételes értékadások esetén.

Szintaxis

```
feltétel ? érték_ha_igaz : érték_ha_hamis
```

Bár a feltételes operátor tömör és hatékony, túlzott használata ronthatja a kód olvashatóságát, ezért érdemes mértékkel alkalmazni.

```php
$age = 20;
$status = ($age >= 18) ? "felnőtt" : "kiskorú";
```

```php
echo $status; // Kimenet: "felnőtt"

// Egymásba ágyazott használat (kerülendő)
$result = $a > $b ? "a nagyobb" : ($a < $b ? "b nagyobb" :
    "egyenlőek");
```

2.20. kódpélda. *Feltételes operátor használata*

2.3.7. Null coalescing operátor

A *null coalescing* operátor (`??`) a PHP 7-ben került bevezetésre, és jelentősen leegyszerűsíti a nullérték-ellenőrzéseket. Ez az operátor különösen hasznos, amikor alapértelmezett értéket szeretnénk adni, ha egy változó nem létezik vagy `null` értékű.

```php
$name = $firstName ?? "Névtelen";

echo $name;
// Ha $firstName nem létezik vagy null, "Névtelen" lesz
    kiírva

// Ekvivalens megoldás:
$name = isset($firstName) ? $firstName : "Névtelen";

// Láncolható: több forrásból az első nem null értéket
    választja
$username = $_GET['user'] ?? $_POST['user'] ?? 'vendeg';

$data = $result['data'] ?? [];
```

2.21. kódpélda. *Null coalescing operátor használata*

A `??=` operátor csak akkor ad értéket a változónak, ha az még nem létezik vagy `null`.

```php
$config['debug'] ??= false;
// Ekvivalens: $config['debug'] = $config['debug'] ?? false;
```

2.22. kódpélda. *A `??=` operátor használata*

2.3.8. Spaceship operátor

A *spaceship* operátor (`<=>`) egy összehasonlító operátor, amely a PHP 7-ben került bevezetésre. Tipikus alkalmazási területe a rendezési függvények és az összetettebb összehasonlítások.

Az operátor összehasonlítja a bal és jobb oldali értékeket, majd a következő értékek egyikét adja vissza:

- -1, ha a bal oldali érték kisebb;
- 0, ha a két érték egyenlő;
- 1, ha a bal oldali érték nagyobb.

Példa:

```php
echo 1 <=> 1; // Kimenet: 0  (egyenlő)
echo 1 <=> 2; // Kimenet: -1 (bal oldal kisebb)
echo 2 <=> 1; // Kimenet: 1  (bal oldal nagyobb)

// Használat rendezésnél
$fruits = ['alma', 'körte', 'szilva'];

usort($fruits, function ($a, $b) {
    return $a <=> $b;
});

print_r($fruits);
// Kimenet: Array ( [0] => alma [1] => körte [2] => szilva )
```

2.23. kódpélda. *Spaceship operátor használata*

2.4. Vezérlési szerkezetek

A vezérlési szerkezetek a programozás alapvető elemei, amelyek lehetővé teszik a program végrehajtásának irányítását. PHP-ben számos vezérlési szerkezet áll rendelkezésre, amelyek segítségével feltételes végrehajtást, ciklusokat és összetettebb programlogikát valósíthatunk meg.

2.4.1. Utasítás és utasításblokk

A PHP-programokban az utasítások az alapvető végrehajtási egységek. Egy utasítás általában egy műveletet hajt végre, és pontosvesszővel zárul. Az utasításblokk több utasítás összefogására szolgál, és kapcsos zárójelek közé írjuk.

Szintaxis

```
utasítás;

{
    utasítás1;
    utasítás2;
    // ...
}
```

Példa:

```php
$total = 0; // Egyetlen utasítás

{
    $price = 100;
    $tax = 0.2;
    $total = $price * (1 + $tax);
}

echo "Összesen: $total";
```

2.24. kódpélda. *Utasítás és utasításblokk*

2.4.2. Az if utasítás

Az `if` utasítás feltételes végrehajtásra szolgál. A program a megadott feltétel értékétől függően hajtja végre az adott kódrészletet.

Szintaxis

```
// Egyszerű if utasítás
if (feltétel) {
    // kód, ha a feltétel igaz
}
```

```php
$age = 18;

if ($age >= 18) {
    echo "Felnőtt.";
}
```

2.25. kódpélda. *Egyszerű if utasítás*

Szintaxis

```
// Kétágú if utasítás
if (feltétel) {
    // kód, ha a feltétel igaz
} else {
    // kód, ha a feltétel hamis
}
```

```php
$balance = 1000;
$withdrawal = 1200;

if ($balance >= $withdrawal) {
    echo "Sikeres tranzakció.";
    $balance -= $withdrawal;
} else {
    echo "Nincs elég fedezet.";
}
```

2.26. kódpélda. *Kétágú if utasítás*

Szintaxis

```
// Többágú if utasítás
if (feltétel1) {
    // kód, ha feltétel1 igaz
} elseif (feltétel2) {
    // kód, ha feltétel1 hamis, de feltétel2 igaz
} else {
    // kód, ha egyik feltétel sem igaz
}
```

```php
$grade = 75;

if ($grade >= 90) {
    echo "Jeles";
} elseif ($grade >= 80) {
    echo "Jó";
} elseif ($grade >= 70) {
    echo "Közepes";
} elseif ($grade >= 60) {
    echo "Elégséges";
} else {
    echo "Elégtelen";
}
```

2.27. kódpélda. *Többágú if utasítás*

Az egymásba ágyazott `if` szerkezetek lehetővé teszik összetett döntési fák kialakítását, azonban túlzott használatuk ronthatja a kód olvashatóságát. Az alábbi példa ezt szemlélteti:

```php
$username = "kovacs.anna";
$password = "titok123";
$isAdmin = false;

if ($username === "kovacs.anna") {
    if ($password === "titok123") {
        echo "Sikeres bejelentkezés. ";

        if ($isAdmin) {
            echo "Üdvözöljük az admin felületen!";
        } else {
            echo "Üdvözöljük a felhasználói felületen!";
        }
    } else {
        echo "Hibás jelszó.";
    }
} else {
    echo "A felhasználó nem található.";
}
```

2.28. kódpélda. *Egymásba ágyazott if utasítások*

2.4.3. A switch utasítás

A `switch` utasítás lehetővé teszi egy kifejezés értékének több lehetséges esettel történő összehasonlítását. Ez különösen hasznos, amikor sok elágazást kell kezelni.

Szintaxis

```
switch (kifejezés) {
    case érték1:
        // kód, ha kifejezés == érték1
        break;

    case érték2:
        // kód, ha kifejezés == érték2
        break;

    // ...

    default:
        // kód, ha egyik eset sem teljesül
}
```

Lássunk egy egyszerű példát:

```php
$color = "piros";

switch ($color) {
    case "piros":
        echo "A szín piros.";
        break;

    case "kék":
        echo "A szín kék.";
        break;

    default:
        echo "A szín nem piros vagy kék.";
}
// Kimenet: A szín piros.
```

2.29. kódpélda. *Egyszerű switch utasítás*

Fontos megjegyezni, hogy a `break` utasítás hiányában a végrehajtás a következő `case` ágon folytatódik. Ez lehetővé teszi több eset összevonását, ahogy az alábbi példában is látható:

```php
$day = "Hétfő";

switch ($day) {
    case "Hétfő":
        echo "A munkahét első napja.";
        break;

    case "Kedd":
    case "Szerda":
    case "Csütörtök":
        echo "Hétköznap.";
        break;

    case "Péntek":
        echo "Közeleg a hétvége!";
        break;

    case "Szombat":
    case "Vasárnap":
        echo "Hétvége van!";
        break;

    default:
        echo "Érvénytelen nap.";
}
```

2.30. kódpélda. *Switch utasítás több eset összevonásával*

2.4.4. A match kifejezés

A PHP 8.0-tól elérhető `match` kifejezés a `switch` modernebb alternatívája. Főbb jellemzői:

- szigorú összehasonlítást (`===`) használ;
- értéket ad vissza, így közvetlenül hozzárendelhető változóhoz;
- nincs szükség `break` utasításra, minden ág automatikusan lezárul;
- ha nincs egyezés és nincs `default` ág, `UnhandledMatchError` kivételt dob.

Szintaxis

```
$eredmény = match (kifejezés) {
    érték1 => visszatérési_érték1,
    érték2, érték3 => visszatérési_érték2,
    default => alapértelmezett_érték
};
```

Egyszerű példa:

```
$role = 'szerkesztő';

$label = match ($role) {
    'admin'      => 'Rendszergazda',
    'szerkesztő' => 'Szerkesztő',
    'olvasó'     => 'Olvasó',
    default      => 'Ismeretlen szerepkör'
};

echo $label; // Kimenet: Szerkesztő
```

2.31. kódpélda. *Egyszerű match kifejezés*

A `match` kifejezés több érték kezelésére is alkalmas:

```
$day = 'Szombat';

$dayType = match ($day) {
    'Hétfő', 'Kedd', 'Szerda', 'Csütörtök', 'Péntek' =>
        'hétköznap',
    'Szombat', 'Vasárnap' => 'hétvége',
};

echo $dayType; // Kimenet: hétvége
```

2.32. kódpélda. *Match kifejezés több értékkel*

Feltételek használata:

```php
$age = 25;

$category = match (true) {
    $age < 18                => 'kiskorú',
    $age >= 18 && $age < 65 => 'felnőtt',
    $age >= 65               => 'idős',
};

echo $category; // Kimenet: felnőtt
```

2.33. kódpélda. *Match kifejezés feltételekkel*

Főbb különbségek a `switch` és a `match` között:

- **Összehasonlítás:** a `switch` laza (==), míg a `match` szigorú (===) összehasonlítást használ.
- **Visszatérési érték:** a `match` kifejezés értéket ad vissza, míg a `switch` utasítás nem.
- A `switch` esetén explicit `break` utasítás szükséges, míg a `match` automatikusan lezárja az ágakat.

```php
$value = "1";

// Switch - laza összehasonlítás
switch ($value) {
    case 1:
        echo "Szám\n";
        break;

    case "1":
        echo "String\n";
        break;
}

// Kimenet: Szám

// Match - szigorú összehasonlítás
$result = match ($value) {
    1   => "Szám",
    "1" => "String",
};

echo $result; // Kimenet: String
```

2.34. kódpélda. *A switch és match összehasonlítása*

A `match` kifejezés különösen hasznos egyszerű értékleképezésekhez és konfigurációs beállítások kezeléséhez. Modern PHP-kódban gyakran előnyösebb a `switch` használatánál.

2.4.5. A while ciklus

A `while` ciklus lehetővé teszi, hogy egy utasításblokkot addig ismételjünk, amíg egy feltétel teljesül.

Szintaxis

```
while (feltétel) {
    // ismétlendő kód
}
```

Egyszerű példa:

```php
$i = 1;

while ($i <= 5) {
    echo $i . " ";
    $i++;
}
// Kimenet: 1 2 3 4 5
```

2.35. kódpélda. *Egyszerű while ciklus*

Összetettebb példa:

```php
$attempts = 0;
$maxAttempts = 5;

while ($attempts < $maxAttempts) {
    echo "Próbálkozás: " . ($attempts + 1) . "\n";

    $success = (rand(0, 1) == 1);

    if ($success) {
        echo "Sikeres művelet!\n";
        break;
    }

    $attempts++;
}

if ($attempts == $maxAttempts) {
    echo "Elérte a maximális próbálkozások számát.";
}
// Kimenet: a véletlen függvényében változó
```

2.36. kódpélda. *Összetettebb while ciklus*

2.4.6. A do...while ciklus

A `do...while` ciklus hasonló a `while` ciklushoz, azzal a különbséggel, hogy a feltétel ellenőrzése a ciklusmag végrehajtása után történik, így a ciklusmag legalább egyszer biztosan lefut.

Szintaxis

```
do {
    // ismétlendő kód
} while (feltétel);
```

Példa:

```php
$pin = "";

do {
    $pin = readline("Adja meg a PIN kódot: ");

    if ($pin != "1234") {
        echo "Helytelen PIN. Próbálja újra.\n";
    }

} while ($pin != "1234");

echo "Sikeres bejelentkezés!";
// Kimenet: a felhasználói bevitel függvényében változó
```

2.37. kódpélda. *A do...while ciklus használata*

2.4.7. A for ciklus

A `for` ciklus kompakt módot biztosít ismétlések végrehajtására, ahol az inicializálás, a feltétel és a léptetés egyetlen szerkezetben szerepel.

Szintaxis

```
for (inicializálás; feltétel; léptetés) {
    // ismétlendő kód
}
```

A `for` ciklus három fő részből áll:

1. **Inicializálás:** a ciklusváltozó kezdőértékének megadása; csak egyszer fut le, a ciklus indulásakor.

2. **Feltétel:** meghatározza, hogy a ciklus folytatódjon-e; minden iteráció előtt kiértékelődik.

3. **Léptetés:** minden iteráció végén hajtódik végre; általában itt növeljük vagy csökkentjük a ciklusváltozót.

Egyszerű példa: számok kiírása növekvő sorrendben

```php
for ($i = 1; $i <= 5; $i++) {
    echo $i . " ";
}
// Kimenet: 1 2 3 4 5
```

2.38. kódpélda. *Egyszerű for ciklus*

Csökkenő sorrendben (visszafelé léptetés):

```php
for ($i = 5; $i >= 1; $i--) {
    echo $i . " ";
}
// Kimenet: 5 4 3 2 1
```

2.39. kódpélda. *For ciklus visszafelé léptetéssel*

Összetettebb példa:

```php
$total = 0;
$items = [10, 20, 30, 40, 50];

for ($i = 0; $i < count($items); $i++) {
    $total += $items[$i];
    echo "Részösszeg a " . ($i + 1) . ". tétel után:
        $total\n";
}

echo "Végösszeg: $total";
```

2.40. kódpélda. *Összetettebb for ciklus*

2.4.8. A foreach ciklus

A `foreach` ciklus kifejezetten tömbök bejárására szolgál. Automatikusan végigiterál a tömb elemein.

Szintaxis

```
foreach ($tömb as $érték) {
    // kód minden elemre
}

foreach ($tömb as $kulcs => $érték) {
    // kód minden kulcs-érték párra
}
```

A `foreach` ciklusnak két változata van:

1. Az első változat csak az értékeket adja vissza.
2. A második változat a kulcsokat és az értékeket is visszaadja.

Egyszerű példa:

```php
$colors = ["piros", "kék", "zöld"];

foreach ($colors as $color) {
    echo $color . " ";
}
// Kimenet: piros kék zöld
```

2.41. kódpélda. *Egyszerű foreach ciklus*

Összetettebb példa:

```php
$prices = [1200, 350, 89, 4500, 275];
$total = 0;

foreach ($prices as $index => $price) {
    echo ($index + 1) . ". termék ára: $price RON\n";
    $total += $price;
}

echo "Összesen: $total RON";
```

2.42. kódpélda. *Összetettebb foreach ciklus*

2.4.9. Ciklusvezérlő utasítások

A PHP két fontos ciklusvezérlő utasítást kínál: `break` és `continue`. Ezek a ciklusok végrehajtásának finomhangolására szolgálnak.

- `break`: azonnal kilép a ciklusból.

- `continue`: a ciklus következő iterációjára ugrik.

Példa páros számok összegzésére:

```php
$numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
$sum = 0;

foreach ($numbers as $number) {
    if ($number % 2 != 0) {
        continue; // a páratlan számokat kihagyjuk
    }

    if ($sum + $number > 20) {
        break; // megállunk, ha az összeg meghaladná a 20-at
    }

    $sum += $number;
}

echo "Páros számok összege (max. 20): $sum";  // Kimenet: 20
```

2.43. kódpélda. *Ciklusvezérlő utasítások használata*

2.4.10. Egymásba ágyazott ciklusok

A PHP lehetővé teszi ciklusok egymásba ágyazását, ami összetett iterációs struktúrák kialakítását teszi lehetővé.

Egyszerű példa egymásba ágyazott ciklusokra:

```php
for ($i = 1; $i <= 3; $i++) {
    for ($j = 1; $j <= 3; $j++) {
        echo "($i, $j) ";
    }

    echo "<br>";
}
// Kimenet:
// (1, 1) (1, 2) (1, 3)
// (2, 1) (2, 2) (2, 3)
// (3, 1) (3, 2) (3, 3)
```

2.44. kódpélda. *Egyszerű egymásba ágyazott ciklusok*

Összetettebb példa:

```php
$students = ["Anna", "Béla", "Csilla"];
$subjects = ["Webprogramozás", "Adatbázisok",
    "Algoritmusok"];
```

```php
foreach ($students as $student) {
    foreach ($subjects as $subject) {
        echo "$student - $subject\n";
    }
}
```

2.45. kódpélda. *Egymásba ágyazott ciklusok tömbbel*

Egymásba ágyazott ciklusok esetén a `break` és a `continue` utasítás után egy szám is megadható. Ez a szám azt jelöli, hogy hány egymásba ágyazott ciklusra vonatkozzon az utasítás. Például a `break 2` a belső és a külső ciklusból is kilép:

```php
foreach ($students as $student) {
    foreach ($subjects as $subject) {
        if ($subject === "Adatbázisok") {
            break 2; // mindkét ciklusból kilép
        }
        echo "$student - $subject\n";
    }
}
// Kimenet: Anna - Webprogramozás
```

2.46. kódpélda. *Kilépés több ciklusból*

2.5. Függvények

A függvények a PHP-programozás alapvető építőkövei. Lehetővé teszik a kód újrafelhasználását, a program strukturálását és a komplexitás csökkentését. Ebben az alfejezetben részletesen áttekintjük a PHP-függvények különböző aspektusait, a szintaxistól kezdve a speciális függvénytípusokig.

2.5.1. Függvények alapjai

Függvények szintaxisa

A PHP-ben a függvényeket a `function` kulcsszóval definiáljuk. A függvénydefiníció tartalmazza a függvény nevét, a paraméterlistát (ha van), valamint a függvény törzsét.

Szintaxis

```
function függvényNév(paraméter1, paraméter2, ...)
{
    // függvény törzse
    return visszatérési_érték;
}
```

Példa:

```php
function greeting($name)
{
    return "Szia, $name!";
}

echo greeting("János");  // Kimenet: Szia, János!
```

2.47. kódpélda. *Egyszerű függvénydefiníció*

Ebben a példában a `greeting()` függvény egy `$name` paramétert fogad, és egy üdvözlő üzenetet ad vissza. A függvény meghívásakor a „János” értéket argumentumként adjuk át.

Függvényekkel kapcsolatos fontosabb szabályok

A PHP-függvények használatakor néhány alapvető szabályt érdemes figyelembe venni:

- A függvénynevek nem lehetnek foglalt kulcsszavak (pl. `if`, `else`, `while`).
- A függvénynevek technikailag nem kis- és nagybetű-érzékenyek, azonban a konzisztens kódstílus érdekében ajánlott mindig az eredeti írásmódot használni.
- A feltétel nélkül definiált függvények a definíciójuk előtt is meghívhatók. A feltételesen (például egy `if` blokkban) definiált függvény csak a definíció végrehajtása után hívható.
- A túlterhelés (*overloading*) nem támogatott: nem definiálható azonos nevű függvény eltérő paraméterlistával.

Eljárásszerű és függvényszerű működés

A PHP-ben nincs külön szintaxis az eljárások és a függvények számára. Azok a függvények, amelyek nem adnak vissza értéket, eljárásszerűen viselkednek.

```php
function procedureExample($msg)
{
    echo $msg;
}

function functionExample($a, $b)
{
    return $a + $b;
}

procedureExample("Szia!");  // Kimenet: Szia!
$result = functionExample(3, 4);  // $result értéke: 7
```

2.48. kódpélda. *Eljárásszerű és függvényszerű működés*

Ebben a példában a `procedureExample()` függvény csak kiír egy üzenetet, míg a `functionExample()` visszatérési értéket is ad, amely eltárolható egy változóban.

2.5.2. Paraméterek és visszatérési értékek

Paraméterek használata

A függvények paramétereket fogadhatnak, amelyek lehetnek kötelezőek vagy opcionálisak. A paraméterek lehetővé teszik adatok átadását a függvény számára feldolgozásra.

```php
function greet($name, $greeting = "Szia")
{
    return "$greeting, $name!";
}

echo greet("Anna");                  // Kimenet: Szia, Anna!
echo greet("Péter", "Jó reggelt");   // Kimenet: Jó reggelt,
    Péter!
```

2.49. kódpélda. *Paraméterek használata*

Ebben a példában a `greet()` függvény két paramétert fogad: egy kötelező `$name` paramétert és egy opcionális `$greeting` paramétert, amelynek alapértelmezett értéke „Szia”.

Alapértelmezett paraméterérték

Az opcionális paraméterekhez alapértelmezett érték adható. Ezeknek a paramétereknek a függvénydefinícióban mindig a kötelező paraméterek után kell szerepelniük.

```php
function power($base, $exponent = 2)
{
    return $base ** $exponent;
}

echo power(3);      // Kimenet: 9 (3^2)
echo power(2, 3);   // Kimenet: 8 (2^3)
```

2.50. kódpélda. *Alapértelmezett paraméterérték*

Itt a `power()` függvény egy kötelező `$base` paramétert és egy opcionális `$exponent` paramétert fogad, amelynek alapértelmezett értéke 2.

Paramétertípusok megadása (type hinting)

PHP 7.0-tól kezdve lehetőség van a paraméterek típusának explicit megadására. Ez javítja a kód olvashatóságát és segíti a hibák korai felismerését.

```php
declare(strict_types=1);

function add(int $a, int $b): int
{
    return $a + $b;
}

echo add(5, 3);  // Kimenet: 8
// echo add("5", "3");  // TypeError kivétel
```

2.51. kódpélda. *Paramétertípusok megadása*

Ebben a példában az `add()` függvény két egész számot (`int`) vár paraméterként, és garantálja, hogy a visszatérési érték is egész szám lesz. A `strict_types=1` beállítás hatására a PHP a paraméterek átadásakor és a visszatérési értéknél nem végez automatikus típuskonverziót, ezért az `add("5", "3")` hívás `TypeError` kivételt okoz. A függvény törzsében használt operátorokra ez nem vonatkozik, ott a numerikus sztringek továbbra is automatikusan számmá alakulnak.

Union típusok (egyesített típusok)

A PHP 8.0-tól lehetőség van több típus megadására egy paraméternél vagy visszatérési értéknél a `|` (*pipe*) operátor használatával.

```php
function processInput(int|float $number): int|float
{
    return $number * 2;
}
```

```php
echo processInput(5);       // 10
echo processInput(3.5);     // 7

// Több típus kombinálása
function getValue(string $key): int|string|bool|null
{
    return match ($key) {
        'count'  => 42,
        'name'   => 'PHP',
        'active' => true,
        default  => null
    };
}

// Az array típus is használható union típusban
function getData(): array|false
{
    // Sikertelen művelet esetén false értéket ad vissza
    return false;
}
```

2.52. kódpélda. *Union típusok használata*

Fontos szabályok:

- A `void` típus nem szerepelhet union típusban.
- Duplikált típusok nem engedélyezettek.
- A típusok sorrendje nem számít.
- A `?Type` rövidítés (pl. `?string`) ekvivalens a `Type|null` union típussal.

Mixed és never típusok

Mixed típus

A `mixed` típus bármilyen értéket elfogad, mivel lefedi az összes PHP beépített típust. Akkor érdemes használni, ha egy függvény valóban bármilyen típusú értékkel képes dolgozni, azonban általában célszerű kerülni, és inkább pontosabb union típusokat alkalmazni.

```php
function processValue(mixed $value): mixed
{
    // Bármilyen típusú értéket elfogad és visszaadhat
    if (is_string($value)) {
        return strtoupper($value);
    }
```

```
    if (is_numeric($value)) {
        return $value * 2;
    }

    return $value;
}

echo processValue("szia");    // SZIA
echo processValue(5);         // 10
print_r(processValue([1, 2]));
```

2.53. kódpélda. *Mixed típus használata*

Never típus

A `never` visszatérési típus azt jelzi, hogy a függvény soha nem tér vissza normál módon: kivételt dob, végtelen ciklusba kerül, vagy megszakítja a futást (`exit()` vagy `die()`). Ez segíti a statikus elemzőket és fejlesztői eszközöket az elérhetetlen kód felismerésében.

```
function redirect(string $url): never
{
    header("Location: $url");
    exit();
}

function throwError(string $message): never
{
    throw new Exception($message);
}

function infiniteLoop(): never
{
    while (true) {
        // végtelen ciklus
    }
}

// A never típusú függvény után a kód nem fut tovább
redirect('/home');
echo "Ez nem fog lefutni"; // elérhetetlen kód
```

2.54. kódpélda. *Never típus használata*

Visszatérési érték típusa

A függvény visszatérési értékének típusát is megadhatjuk, ami tovább növeli a kód biztonságát és olvashatóságát.

```php
function getName(): string
{
    return "János";
}
$name = getName();
var_dump($name);  // string(6) "János"
```

2.55. kódpélda. *Visszatérési érték típusa*

A `getName()` függvény garantálja, hogy mindig `string` típusú értéket ad vissza.

Szigorú mód (strict mode)

A *szigorú mód* bekapcsolásával a PHP szigorúbban ellenőrzi a típusokat, ami segít a típushibák korai felismerésében.

```php
declare(strict_types=1);
function add(int $a, int $b): int
{
    return $a + $b;
}
echo add(5, 3);  // Működik
// echo add("5", "3");  // TypeError kivétel
```

2.56. kódpélda. *A szigorú mód használata*

Szigorú módban a PHP a függvényhívásoknál nem végez automatikus típuskonverziót, ezért az „5” és a „3” sztringértékek átadása `TypeError` kivételt vált ki.

Referencia szerinti paraméterátadás

Alapértelmezetten a PHP érték szerint adja át a paramétereket. Referencia szerinti átadáshoz a `&` referenciaoperátort használjuk, amely lehetővé teszi, hogy a függvény módosítsa az eredeti változó értékét.

```php
function increment(&$number)
{
    $number++;
}
$a = 5;
increment($a);
echo $a;  // Kimenet: 6
```

2.57. kódpélda. *Referencia szerinti paraméterátadás*

Ebben a példában az `increment()` függvény referencia szerinti paraméterátadás miatt módosítja az `$a` változó értékét.

Név szerinti argumentumok (named arguments)

A PHP 8.0-tól lehetőség van a függvényparaméterek név szerinti megadására, amely jelentősen növeli a kód olvashatóságát, különösen sok paraméterrel rendelkező függvények esetén.

```
function createUser(
    string $name,
    string $email,
    int $age = 18,
    bool $active = true
) {
    // ...
}

// Hagyományos hívás
createUser("Kovács János", "kovacs.janos@example.com", 25,
    false);

// Named arguments használata
createUser(
    name: "Kovács János",
    email: "kovacs.janos@example.com",
    age: 25,
    active: false
);

// Csak a kötelező paraméterek megadása
createUser(
    name: "Nagy Péter",
    email: "nagy.peter@example.com"
);
```

2.58. kódpélda. *Név szerinti argumentumok használata*

Előnyök:

- növeli a kód olvashatóságát;
- az alapértelmezett értékkel rendelkező paraméterek kihagyhatók;
- a név szerinti argumentumok sorrendje tetszőleges;
- csökkenti a hibalehetőséget hosszú paraméterlisták esetén.

2.5.3. Változók hatóköre

A változó hatóköre megadja, hogy a program mely részéből érhető el a változó. PHP-ben minden függvénynek saját hatóköre van.

Lokális változók

A függvényen belül definiált változók lokálisak, csak a függvényen belül érhetők el.

```php
function showMessage()
{
    $message = "Helló a függvényből!";  // lokális változó
    echo $message;
}

showMessage();   // Kimenet: Helló a függvényből!
echo $message;   // Warning: Undefined variable $message
```

2.59. kódpélda. *Lokális változók*

A függvényen kívül definiált (globális) változók a függvény törzsében nem érhetők el. A PHP a `global` kulcsszóval lehetővé teszi az elérésüket, de ez rejtett függőséget teremt, ezért a szükséges értékeket inkább paraméterként adjuk át.

Statikus változók

A `static` kulcsszóval definiált lokális változók megtartják értéküket a függvényhívások között.

```php
function counter()
{
    static $count = 0;
    return ++$count;
}

echo counter();  // Kimenet: 1
echo counter();  // Kimenet: 2
echo counter();  // Kimenet: 3
```

2.60. kódpélda. *Statikus változók*

Statikus változót olyankor használunk, amikor egy függvénynek a hívások között meg kell őriznie valamilyen állapotot, például egy számlálót vagy egy egyszer kiszámolt, később újra felhasználható értéket. Objektumorientált kódban erre a célra inkább osztálytulajdonságokat használunk (lásd a 3. fejezetet).

2.5.4. Speciális függvénytípusok

Névtelen függvények

A névtelen függvények (anonim függvények) olyan függvények, amelyeknek nincs nevük. Változóhoz rendelhetők, függvényparaméterként átadhatók, vagy visszatérési értékként visszaadhatók.

```php
$greet = function ($name) {
    return "Helló, $name!";
};

echo $greet("Anna");  // Kimenet: Helló, Anna!
```

2.61. kódpélda. *Névtelen függvény változóhoz rendelve*

A névtelen függvényeket leggyakrabban *callback* (visszahívó) függvényként használjuk. Ez olyan függvény, amelyet paraméterként adunk át egy másik függvénynek, és az hívja meg. Tipikus példa a tömbfüggvények paramétere. Ezeket a függvényeket a 2.6.7. alfejezetben mutatjuk be részletesen:

```php
$numbers = [1, 2, 3, 4, 5];
$evens = array_filter($numbers, function ($n) {
    return $n % 2 == 0;
});
print_r($evens);  // Array ( [1] => 2 [3] => 4 )
```

2.62. kódpélda. *Névtelen függvény callbackként*

Closure

A *closure* olyan névtelen függvény, amely a `use` kulcsszóval hozzáfér a definiálásakor érvényes külső hatókör változóihoz. A névtelen függvény ugyanis alapértelmezetten nem látja a körülvevő hatókör változóit.

Szintaxis

```
$closure = function(paraméterek) use (külső változók) {
    // függvény törzse
};
```

Ahol:

- `function`: a névtelen függvény deklarálása;
- **paraméterek**: a függvény által fogadott paraméterek (opcionálisak);
- `use`: a kulcsszó, amely után a felhasználni kívánt külső változókat soroljuk fel;

- **függvény törzse**: itt hajtódik végre a closure logikája.

```php
$message = 'Helló';

$greet = function () use ($message) {
    echo $message;
};

$greet(); // Kimenet: Helló
```

2.63. kódpélda. *Closure példa*

A fenti példában a `use` kulcsszóval tesszük elérhetővé a külső hatókörben lévő `$message` változót a closure számára. A külső változók érték szerint adódnak át, vagyis a closure a definiálás pillanatában érvényes értéküket látja. Ha a closure módosítani is szeretné a külső változót, azt referenciaként kell átadni: `use (&$message)`.

A closure paramétereket is fogadhat:

```php
$multiplier = 2;

$multiply = function ($value) use ($multiplier) {
    return $value * $multiplier;
};

echo $multiply(5); // Kimenet: 10
```

2.64. kódpélda. *Closure paraméterekkel*

Nyílfüggvény (arrow function)

A *nyílfüggvény* (arrow function) a *closure* rövidebb, tömörebb szintaxisa, amely PHP 7.4-től érhető el. A nyílfüggvény automatikusan hozzáfér a definiálásakor érvényes külső hatókör változóihoz, `use` kulcsszó nélkül.

Szintaxis

```
$fn = fn(paraméterek) => kifejezés;
```

Ahol:

- `fn`: a nyílfüggvény kulcsszava;
- **paraméterek**: a függvény által fogadott paraméterek (opcionálisak);
- `=>`: a nyíl operátor, amely elválasztja a paramétereket a kifejezéstől;

- **kifejezés**: egyetlen kifejezés, amely a függvény visszatérési értéke lesz.

A nyílfüggvény használatának főbb tulajdonságai:

- Mindig egyetlen kifejezést tartalmaz, amely automatikusan a visszatérési érték lesz.
- Nincs szükség explicit `return` kulcsszóra.
- Automatikusan örökli a külső hatókör változóit érték szerint, ezért nincs szükség `use` kulcsszóra.
- A külső változók értékét nem módosíthatja.
- Nem tartalmazhat több utasítást.
- Osztálymetódusokon belül is használható, és hozzáfér a `$this` változóhoz.

```php
$message = 'Szia';

$greet = fn($name) => $message . ', ' . $name;

echo $greet('világ'); // Kimenet: Szia, világ
```

2.65. kódpélda. *Nyílfüggvény példa*

A fenti példában a `$message` változó automatikusan elérhető a nyílfüggvény számára, anélkül hogy azt a `use` kulcsszóval expliciten át kellene adni.

A következő példában egy tipikus felhasználási mód látható callback függvényként, ahol egy tömb minden elemét megszorozzuk egy értékkel:

```php
$numbers = [1, 2, 3, 4, 5];
$multiplier = 2;

$doubled = array_map(fn($n) => $n * $multiplier, $numbers);

print_r($doubled);
// Array ( [0] => 2 [1] => 4 [2] => 6 [3] => 8 [4] => 10 )

// Összehasonlítás hagyományos closure szintaxissal:
$doubledClosure = array_map(function($n) use ($multiplier)
{
    return $n * $multiplier;
}, $numbers);
```

2.66. kódpélda. *Nyílfüggvény callback függvényként*

A nyílfüggvény különösen hasznos rövid callback függvények esetén, például tömbműveleteknél.

Változó számú paramétert fogadó (variadic) függvények

A *variadic* függvények olyan függvények, amelyek változó számú argumentumot képesek fogadni. PHP 5.6-tól kezdve a „három pont” (`...`) operátorral definiálhatók.

Szintaxis

```
function függvényNév(...paraméterek)
{
    // függvény törzse
}
```

Ahol:

- **...**: a variadic operátor, amely jelzi, hogy a függvény változó számú argumentumot fogad;
- **paraméterek**: egy tömb, amely az összes átadott argumentumot tartalmazza.

```php
function sum(...$numbers)
{
    return array_sum($numbers);
}

echo sum(1, 2, 3, 4);  // Kimenet: 10
echo sum(5, 10, 15);   // Kimenet: 30
```

2.67. kódpélda. *Egyszerű variadic függvény*

A variadic paraméter a függvényen belül tömbként érhető el, így az összes beépített tömbfüggvény használható rajta. Az alábbi példában sztringek összefűzésére alkalmazzuk:

```php
function concatenate(...$strings)
{
    return implode(', ', $strings);
}
echo concatenate('alma', 'banán', 'citrom');
// Kimenet: alma, banán, citrom
echo concatenate('egy', 'kettő');
// Kimenet: egy, kettő
```

2.68. kódpélda. *Variadic függvény sztringek összefűzésére*

A variadic függvények előnyei:

- rugalmasabb függvénydefiníciók;
- tisztább kód, az argumentumok közvetlenül tömbként érhetők el;
- jobb olvashatóság és könnyebben értelmezhető kód;
- egyszerűbb paramétertovábbítás más függvények felé.

Fontos megjegyezni, hogy a variadic paraméternek mindig a paraméterlista végén kell szerepelnie. Emellett típusdeklaráció használata esetén az minden átadott argumentumra vonatkozik.

```php
declare(strict_types=1);

function sumIntegers(int ...$numbers)
{
    return array_sum($numbers);
}

echo sumIntegers(1, 2, 3);  // Működik
// echo sumIntegers(1, 2, '3');
// TypeError: Argument 3 must be of type int, string given
```

2.69. kódpélda. *Típusdeklaráció variadic függvénynél*

2.6. Tömbök

A tömbök a PHP egyik legalapvetőbb és legsokoldalúbb adatstruktúrái. Lehetővé teszik több érték egyetlen változóban történő tárolását és kezelését. Használatuk rendkívül elterjedt a PHP-programozásban, mivel rugalmas módot biztosítanak az adatok szervezésére és kezelésére.

2.6.1. A tömbök fajtái

PHP-ben két fő tömbtípus létezik:

- **Indexelt tömbök**: automatikusan hozzárendelt egész kulcsokkal rendelkeznek.
- **Asszociatív tömbök**: kulcs–érték párokat tárolnak, ahol a kulcsok sztringek vagy egész számok lehetnek, az értékek pedig tetszőleges típusúak.

Fontos megjegyezni, hogy a PHP-ben minden tömb asszociatív módon működik a háttérben. Az indexelt tömbök ennek speciális esetei, ahol a kulcsok egész számok.

2.6.2. Indexelt tömbök

Szintaxis és létrehozás

Indexelt tömbök létrehozására a `[]` tömbképző operátort használjuk. A régebbi `array()` forma továbbra is érvényes, de modern PHP-kódban általában kerülendő.

Szintaxis

```
$tömb = [érték1, érték2, ...];
```

Példa:

```php
$fruits = ["alma", "banán", "cseresznye"];
// Régebbi szintaxis (kerülendő)
$vegetables = array("sárgarépa", "brokkoli", "spenót");
// Üres tömb létrehozása és elemek hozzáadása
$colors = [];
$colors[] = "piros";
$colors[] = "zöld";
$colors[] = "kék";
```

2.70. kódpélda. *Indexelt tömb létrehozása*

A `$colors[] = "piros";` alak a tömb végére fűzi az új elemet. Az elem kulcsa az eddigi legnagyobb egész kulcsnál eggyel nagyobb lesz, üres tömb esetén 0.

Elemek elérése

Az indexelt tömbök elemei index alapján érhetők el. Az indexelés 0-tól kezdődik:

```php
echo $fruits[0];       // Kimenet: alma
echo $vegetables[1];   // Kimenet: brokkoli
```

2.71. kódpélda. *Indexelt tömb elemeinek elérése*

Fontos megjegyezni, hogy nem létező index elérése figyelmeztetést eredményezhet. Emiatt gyakran használjuk az `isset()` függvényt annak ellenőrzésére, hogy az adott index létezik-e:

```php
if (isset($fruits[3])) {
    echo $fruits[3];
} else {
    echo "A negyedik elem nem létezik.";
}
```

2.72. kódpélda. *Biztonságos elemhozzáférés*

Tömb elemeinek törlése

Az `unset()` függvény egy vagy több tömbelem törlésére használható. Fontos, hogy az `unset()` nem indexeli újra automatikusan a tömböt.

```php
$fruits = ["alma", "banán", "cseresznye", "datolya"];
unset($fruits[1]);  // törli a "banán" elemet

print_r($fruits);
// Kimenet:
// Array
// (
//     [0] => alma
//     [2] => cseresznye
//     [3] => datolya
// )
```

2.73. kódpélda. *Tömb elemeinek törlése `unset()` függvénnyel*

Ha szükséges a tömb újraindexelése törlés után, az `array_values()` függvény használható:

```php
$fruits = ["alma", "banán", "cseresznye", "datolya"];
unset($fruits[1]);

$fruits = array_values($fruits);  // újraindexelés

print_r($fruits);
// Kimenet:
// Array
// (
//     [0] => alma
//     [1] => cseresznye
//     [2] => datolya
// )
```

2.74. kódpélda. *Tömb újraindexelése törlés után*

2.6.3. Asszociatív tömbök

Az asszociatív tömbök kulcs–érték párok tárolására szolgálnak. Különösen hasznosak strukturált adatok kezelésére, ahol az elemeket beszédes kulcsnevekkel szeretnénk elérni.

Szintaxis

```
$tömb = [
    "kulcs1" => "érték1",
    "kulcs2" => "érték2",
    // további kulcs-érték párok
];
```

Ahol:

- `[]`: az ajánlott szintaxis (PHP 5.4+); a régebbi `array()` forma továbbra is használható, de általában kerülendő.
- **Kulcsok**: sztring vagy egész szám típusú értékek, amelyeknek egyedinek kell lenniük a tömbön belül.
- **Értékek**: tetszőleges típusú adatok, például `string`, `integer`, `float`, `boolean`, tömb vagy objektum.

```
$person = [
    "name" => "Kovács János",
    "age" => 30,
    "city" => "Budapest"
];

$car = [
    "brand" => "Toyota",
    "model" => "Corolla",
    "year" => 2020
];
```

2.75. kódpélda. *Asszociatív tömbök létrehozása*

Az asszociatív tömbök különösen alkalmasak összetett adatok reprezentálására, például személyek, termékek vagy konfigurációs beállítások tárolására.

Elemek elérése

Az asszociatív tömbök elemei a kulcsuk alapján érhetők el:

```
echo $person["name"];   // Kimenet: Kovács János
echo $car["brand"];     // Kimenet: Toyota
```

2.76. kódpélda. *Asszociatív tömb elemeinek elérése*

Akárcsak indexelt tömbök esetén, itt is érdemes ellenőrizni a kulcs létezését az `isset()` vagy az `array_key_exists()` függvényekkel:

```php
if (array_key_exists("age", $person)) {
    echo "Kor: " . $person["age"];
} else {
    echo "Kor nincs megadva.";
}
```

2.77. kódpélda. *Biztonságos elemhozzáférés asszociatív tömbben*

Az `array_key_exists()` függvény akkor is `true` értéket ad vissza, ha a kulcshoz tartozó érték `null`.

Megjegyzés

`array_key_exists()` és `isset()`: az `isset()` `false` értéket ad vissza akkor is, ha a kulcs létezik, de az értéke `null`. Az `array_key_exists()` ezzel szemben kizárólag a kulcs meglétét vizsgálja.

2.6.4. Tömbök bejárása

A tömbök bejárása alapvető művelet, amelyre PHP-ben több lehetőség is rendelkezésre áll.

Indexelt tömbök bejárása

Indexelt tömböket általában `for` vagy `foreach` ciklussal járunk be:

```php
// for ciklussal
for ($i = 0; $i < count($fruits); $i++) {
    echo $fruits[$i] . "<br>";
}

// foreach ciklussal
foreach ($fruits as $fruit) {
    echo $fruit . "<br>";
}
```

2.78. kódpélda. *Indexelt tömb bejárása*

Megjegyzés

Részesítsük előnyben a `foreach` ciklust: tömbök bejárásához általában a `foreach` az olvashatóbb és biztonságosabb választás. Nincs szükség manuális indexkezelésre, és nem fordulhat elő indexhatár-túllépés.

Asszociatív tömbök bejárása

Asszociatív tömböket általában `foreach` ciklussal járunk be, mivel így a kulcs és az érték is egyszerűen elérhető:

```
foreach ($person as $key => $value) {
    echo "$key: $value<br>";
}
```

2.79. kódpélda. *Asszociatív tömb bejárása*

Ez különösen hasznos, mert nem szükséges előre ismerni a tömb kulcsait.

2.6.5. Hasznos tömbfüggvények

A PHP számos beépített függvényt kínál a tömbök kezelésére. Az alábbiakban néhány gyakran használt függvény látható:

- `count()`: a tömb elemeinek számát adja vissza.
- `array_push()`: elemet ad hozzá a tömb végéhez.
- `array_pop()`: eltávolítja és visszaadja a tömb utolsó elemét.
- `array_shift()`: eltávolítja és visszaadja a tömb első elemét.
- `array_unshift()`: elemet szúr be a tömb elejére; a numerikus kulcsok újraszámozódnak.
- `array_merge()`: két vagy több tömböt egyesít.
- `in_array()`: ellenőrzi, hogy egy érték szerepel-e a tömbben.
- `array_key_exists()`: ellenőrzi, hogy egy kulcs létezik-e a tömbben; `true` értéket ad vissza akkor is, ha az érték `null`.

2.6.6. Tömbök rendezése

A tömbök rendezése gyakori művelet a programozásban, és a PHP több beépített függvényt is kínál erre a célra. Ezek a függvények különböző módon rendezik a tömböket, attól függően, hogy milyen típusú a tömb, és milyen rendezési feltételt szeretnénk alkalmazni:

- `sort()`: érték szerint növekvő sorrendbe rendezi a tömböt, és újraindexeli.
- `rsort()`: érték szerint csökkenő sorrendbe rendezi a tömböt, és újraindexeli.

- `asort()`: érték szerint növekvő sorrendbe rendezi a tömböt úgy, hogy megőrzi a kulcs–érték kapcsolatokat.
- `arsort()`: érték szerint csökkenő sorrendbe rendezi a tömböt úgy, hogy megőrzi a kulcs–érték kapcsolatokat.
- `ksort()`: kulcs szerint növekvő sorrendbe rendezi a tömböt.
- `krsort()`: kulcs szerint csökkenő sorrendbe rendezi a tömböt.

```php
$numbers = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5];
sort($numbers);

print_r($numbers);
// Kimenet:
// Array ( [0] => 1 [1] => 1 [2] => 2 [3] => 3 [4] => 3 [5]
    => 4 [6] => 5 [7] => 5 [8] => 5 [9] => 6 [10] => 9 )

$fruits = ["banán" => "sárga", "alma" => "piros",
    "cseresznye" => "piros"];
asort($fruits);

print_r($fruits);
// Kimenet:
// Array ( [alma] => piros [cseresznye] => piros [banán] =>
    sárga )
```

2.80. kódpélda. *Tömbök rendezése*

Fontos megjegyezni, hogy ezek a függvények közvetlenül módosítják az eredeti tömböt. Ha az eredeti tömböt is meg szeretnénk tartani, előtte készítsünk róla másolatot egy egyszerű értékadással (`$sorted = $numbers;`), és a másolatot rendezzük.

2.6.7. `array_map()`, `array_walk()` és `array_filter()` függvények

Ezek a függvények lehetővé teszik, hogy műveleteket végezzünk a tömb elemein, és különösen hasznosak funkcionális programozási stílusú kód írásakor.

`array_map()`

Az `array_map()` függvény egy megadott függvényt alkalmaz a tömb minden elemére, majd az eredményt egy új tömbben adja vissza. Az eredeti tömböt nem módosítja.

```php
$numbers = [1, 2, 3, 4, 5];

$squared = array_map(function($n) {
    return $n * $n;
}, $numbers);

print_r($squared);
// Kimenet:
// Array ( [0] => 1 [1] => 4 [2] => 9 [3] => 16 [4] => 25 )
```

2.81. kódpélda. *`array_map()` használata*

Az `array_map()` különösen hasznos, amikor minden tömbelemre ugyanazt az átalakítást szeretnénk alkalmazni. Gyakran használják adatok formázására vagy transzformálására.

`array_walk()`

Az `array_walk()` függvény egy felhasználó által megadott függvényt alkalmaz a tömb minden elemére, és közvetlenül módosítja az eredeti tömböt.

```php
$fruits = ["a" => "alma", "b" => "banán"];

array_walk($fruits, function(&$value, $key) {
    $value = mb_strtoupper($value);
});

print_r($fruits);
// Kimenet:
// Array ( [a] => ALMA [b] => BANÁN )
```

2.82. kódpélda. *`array_walk()` használata*

Az `array_walk()` különösen hasznos, ha az eredeti tömböt kell módosítani, vagy ha a kulcsokra is szükség van a feldolgozás során. A visszatérési értéke mindig `true`, ezért az eredményt nem egy új tömbben kapjuk vissza.

`array_filter()`

Az `array_filter()` függvény a tömb elemeit egy megadott feltétel alapján szűri. Csak azokat az elemeket tartja meg az eredménytömbben, amelyekre a szűrőfüggvény `true` értéket ad vissza.

```php
$numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

$even = array_filter($numbers, function($n) {
    return $n % 2 == 0;
});
```

```php
print_r($even);
// Kimenet:
// Array ( [1] => 2 [3] => 4 [5] => 6 [7] => 8 [9] => 10 )
```

2.83. kódpélda. *`array_filter()` használata*

Az `array_filter()` megőrzi az eredeti kulcsokat, ezért az eredménytömb indexei nem feltétlenül lesznek folytonosak.

2.6.8. Többdimenziós tömbök

A PHP támogatja a többdimenziós tömböket, amelyek tulajdonképpen „tömbök a tömbben”. Ezek lehetővé teszik összetettebb adatstruktúrák létrehozását és kezelését.

Példa:

```php
$matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
];
echo $matrix[1][2];  // Kimenet: 6
$users = [
    ["name" => "János", "age" => 30],
    ["name" => "Anna",  "age" => 25]
];
echo $users[1]["name"];  // Kimenet: Anna
```

2.84. kódpélda. *Többdimenziós tömb*

A többdimenziós tömbök különösen hasznosak olyan esetekben, amikor táblázatos adatokat vagy hierarchikus struktúrákat kell kezelni. Például egy sakktábla reprezentálása vagy egy összetett JSON-struktúra PHP-beli megfelelője is többdimenziós tömbként ábrázolható.

A többdimenziós tömbök bejárásához általában egymásba ágyazott ciklusokat használunk:

```php
foreach ($users as $user) {
    foreach ($user as $key => $value) {
        echo "$key: $value<br>";
    }
    echo "<br>";
}
```

2.85. kódpélda. *Többdimenziós tömb bejárása*

2.7. Dátum- és időkezelés

A dátum- és időkezelés számos alkalmazásban kiemelten fontos, a felhasználói felületektől kezdve az adatbázis-műveleteken át az ütemezett feladatokig. A PHP többféle eszközt biztosít ezek kezelésére, beleértve a procedurális függvényeket és az objektumorientált megközelítést is.

2.7.1. Aktuális dátum és idő

Az aktuális dátum és idő lekérdezése az egyik legalapvetőbb művelet:

```php
echo date("Y-m-d H:i:s");  // pl. 2025-05-15 14:30:00
echo time();               // UNIX-időbélyeg, pl. 1747308600
```

2.86. kódpélda. *Aktuális dátum és idő lekérdezése*

A `date()` függvény formázott karakterláncot ad vissza, míg a `time()` az aktuális UNIX-időbélyeget (*timestamp*) adja vissza. Ez az 1970. január 1. éjfél (00:00:00, UTC) óta eltelt másodpercek száma. Az időbélyeg széles körben használt, mert könnyen összehasonlítható és feldolgozható.

Az időbélyeg független az időzónától, ugyanabban a pillanatban a világ minden pontján ugyanaz az értéke. A `date()` függvény viszont a beállított időzóna szerint formázza a helyi időt. Az időzónát a `php.ini` `date.timezone` beállítása határozza meg, de a kódból is megadható:

```php
date_default_timezone_set('Europe/Bucharest');
```

2.87. kódpélda. *Időzóna beállítása*

2.7.2. Dátum formázása

A `date()` függvény lehetővé teszi a dátum különböző formátumok szerinti megjelenítését:

```php
$timestamp = time();

echo date("Y. F j., l", $timestamp);  // pl. 2025. May 15.,
    Thursday
echo date("Y-m-d", $timestamp);       // pl. 2025-05-15
echo date("d/m/Y H:i", $timestamp);   // pl. 15/05/2025 14:30
echo date("j. n. Y", $timestamp);     // pl. 15. 5. 2025
```

2.88. kódpélda. *Dátum formázása különböző módokon*

Néhány gyakran használt formátumkarakter (a `date()` függvény a hónap- és napneveket mindig angolul adja vissza):

- **y**: kétjegyű évszám;
- **F**: a hónap teljes neve (pl. January);
- **M**: a hónap rövid neve (pl. Jan);
- **j**: a hónap napja vezető nulla nélkül (1–31);
- **D**: a hét napjának rövid neve (pl. Mon);
- **l**: a hét napjának teljes neve (pl. Monday);
- **g**: 12 órás formátumú óra vezető nulla nélkül (1–12);
- **a**: ante meridiem / post meridiem (am vagy pm).

Magyar nyelvű hónap- és napnevek megjelenítéséhez az `intl` bővítmény `IntlDateFormatter` osztálya használható, amely a megadott nyelvi beállítás szerint formáz:

```php
$formatter = new IntlDateFormatter(
    'hu_HU',
    IntlDateFormatter::FULL,
    IntlDateFormatter::NONE
);
echo $formatter->format(time());  // pl. 2025. május 15.,
    csütörtök
```

2.89. kódpélda. *Dátum formázása magyar nyelven*

2.7.3. Dátum létrehozása és módosítása

Konkrét dátumot létrehozhatunk a `strtotime()` függvénnyel:

```php
$date = strtotime("2025-05-15 14:30:00");
echo date("Y-m-d H:i:s", $date);  // 2025-05-15 14:30:00

$nextWeek = strtotime("+1 week", $date);
echo date("Y-m-d H:i:s", $nextWeek);  // 2025-05-22 14:30:00

$lastDayOfMonth = strtotime("last day of May 2025");
echo date("Y-m-d", $lastDayOfMonth);  // 2025-05-31

$nextTuesday = strtotime("next Tuesday");
echo date("Y-m-d", $nextTuesday);  // következő kedd dátuma
```

2.90. kódpélda. *Dátum létrehozása és módosítása*

A `strtotime()` függvény rugalmas, és természetes nyelvi kifejezéseket is képes értelmezni. Ugyanakkor érdemes körültekintően használni, mert bizonyos kifejezések félreérthetők lehetnek. Például a „next Monday” mindig a következő hétfőt jelenti, még akkor is, ha az aktuális nap hétfő.

2.7.4. `DateTime` osztály

PHP 5.2-től elérhető a `DateTime` osztály, amely objektumorientált megközelítést biztosít a dátum- és időkezeléshez:

```php
$date = new DateTime("2025-05-15 14:30:00");
echo $date->format("Y-m-d H:i:s");  // 2025-05-15 14:30:00

$date->modify("+1 month");
echo $date->format("Y-m-d H:i:s");  // 2025-06-15 14:30:00

$date->add(new DateInterval("P2D"));  // 2 nap hozzáadása
echo $date->format("Y-m-d H:i:s");    // 2025-06-17 14:30:00

$date->sub(new DateInterval("PT3H")); // 3 óra kivonása
echo $date->format("Y-m-d H:i:s");    // 2025-06-17 11:30:00
```

2.91. kódpélda. *DateTime osztály használata*

A `DateTime` osztály előnyei:

- objektumorientált megközelítést biztosít, amely jól illeszkedik összetettebb alkalmazásokhoz;
- beépített időzóna-támogatással rendelkezik;
- egyszerűvé teszi a dátumok összehasonlítását és a különbségek számítását;
- elérhető nem módosítható (immutable) változata is (`DateTime Immutable`), amely csökkenti a mellékhatások lehetőségét.

2.7.5. `DateInterval` és `DatePeriod` osztályok

A `DateInterval` időtartamok kezelésére szolgál, míg a `DatePeriod` ismétlődő időintervallumok kezelésére használható:

```php
$interval = new DateInterval("P1Y2M3DT4H5M6S");
// 1 év, 2 hónap, 3 nap, 4 óra, 5 perc, 6 másodperc

$date = new DateTime("2025-01-01");
$date->add($interval);
```

```php
echo $date->format("Y-m-d H:i:s");
// 2026-03-04 04:05:06

$start = new DateTime("2025-01-01");
$interval = new DateInterval("P1D");
$end = new DateTime("2025-01-10");

$period = new DatePeriod($start, $interval, $end);

foreach ($period as $date) {
    echo $date->format("Y-m-d") . "\n";
}
```

2.92. kódpélda. *DateInterval és DatePeriod használata*

2.7.6. Időzónák kezelése

Az időzónák kezelése különösen fontos nemzetközi alkalmazások esetén. A PHP a `DateTimeZone` osztályt biztosítja erre a célra:

```php
$date = new DateTime("now", new
    DateTimeZone("Europe/Budapest"));
echo $date->format("Y-m-d H:i:s");

$date->setTimezone(new DateTimeZone("America/New_York"));
echo $date->format("Y-m-d H:i:s");
```

2.93. kódpélda. *Időzónák kezelése*

A `DateTime` osztály automatikusan kezeli a nyári időszámítást is.

2.7.7. Dátumok összehasonlítása és különbségek számítása

Dátumok összehasonlítására és a köztük lévő különbség meghatározására több lehetőség is rendelkezésre áll:

```php
$date1 = new DateTime("2025-05-15");
$date2 = new DateTime("2025-06-15");

if ($date1 < $date2) {
    echo "date1 korábbi, mint date2\n";
}

$diff = $date1->diff($date2);

echo $diff->format("%R%a nap\n");  // +31 nap
```

```php
echo "Év különbség: " . $diff->y . "\n";
echo "Hónap különbség: " . $diff->m . "\n";
echo "Nap különbség: " . $diff->d . "\n";

echo "Összes nap különbség: " . $diff->days . "\n";
```

2.94. kódpélda. *Dátumok összehasonlítása és különbségek számítása*

A `diff()` metódus egy `DateInterval` objektumot ad vissza, amely részletes információt tartalmaz a két dátum közötti különbségről.

2.8. Gyakorlófeladatok

2.8.1. Szintaxis, változók, adattípusok

1. Tároljuk egy film adatait változókban (cím, rendező, megjelenési év, értékelés 1–10-es skálán). Írjuk ki az adatokat a `.` operátorral összefűzve, változóbehelyettesítéssel és heredoc szintaxissal is.

2. Egy futárszolgálat díjait rögzítsük konstansokban.

   - A súlyhatárt (30 kg), az alapdíjat (15 RON) és a kilogrammonkénti pótdíjat (2 RON) a `const` kulcsszóval, a cég nevét a `define()` függvénnyel definiáljuk.
   - Számítsuk ki néhány csomag szállítási díját.
   - Próbáljuk meg felülírni az egyik konstanst, és figyeljük meg, mi történik.

3. Írjunk egy számlaösszesítő szkriptet. Tároljuk egy termék adatait (név, egységár, mennyiség, akciós státusz) változókban. Akció esetén alkalmazzunk 15%-os kedvezményt, majd írjuk ki az eredményt formázott szövegként. Ellenőrizzük a változók típusát a `gettype()` függvénnyel.

4. PHP-ben az `==` és az `===` operátor eltérő eredményt adhat. Írjunk egy szkriptet, amely összehasonlítja az alábbi párokat mindkét operátorral: `0` és `false`, `"1"` és `true`, `null` és `0`, `"42"` és `42`, `""` és `false`. Írjuk ki minden összehasonlítás eredményét mindkét operátorral.

5. Kérjünk be a felhasználótól egy mondatot. Írjuk ki a karakterek számát szóközök nélkül, alakítsuk a szöveget csupa kisbetűssé, cseréljük le benne az összes számjegyet `'#'` karakterre, és ellenőrizzük, hogy tartalmaz-e `'@'` karaktert. Csak beépített sztringfüggvényeket használjunk.

6. Készítsünk mozijegy-kalkulátort.
   - Rögzítsük konstansokban az alapjegyárat (35 RON) és a hétvégi felárat (8 RON).
   - Definiáljuk konstansokban a kedvezményeket: gyerek (14 év alatt, 40%), nyugdíjas (65 év felett, 30%), diák (20%).
   - Változókban tároljuk a néző korát, hogy hétvége van-e, és hogy rendelkezik-e diákigazolvánnyal.
   - Ha több kedvezményre is jogosult, csak a legnagyobb érvényesüljön.
   - Számítsuk ki és írjuk ki a fizetendő összeget.

2.8.2. Operátorok, vezérlési szerkezetek

1. Készítsünk egy szkriptet, amely bekér két számot, majd elvégzi rajtuk az alapvető aritmetikai műveleteket (összeadás, kivonás, szorzás, osztás). Használjunk megfelelő operátorokat, és kezeljük a nullával való osztás esetét.

2. Valósítsunk meg egy szkriptet, amely egy vizsgapontszámból (0–100) `match` kifejezéssel meghatározza az érdemjegyet: 90–100 jeles, 75–89 jó, 60–74 közepes, 50–59 elégséges, 0–49 elégtelen. Írjuk ki a pontszámot és a hozzá tartozó érdemjegyet.

3. Döntsük el egy szkript segítségével, hogy egy adott év szökőév-e. A szabályok sorrendben:
   - ha az év osztható 400-zal, szökőév;
   - egyébként ha osztható 100-zal, nem szökőév;
   - egyébként ha osztható 4-gyel, szökőév;
   - minden más esetben nem szökőév.

   Írjuk ki az eredményt és a döntés indoklását.

4. Oldjuk meg a „FizzBuzz” feladatot. Írjuk ki a számokat 1-től 20-ig úgy, hogy a 3-mal osztható számok helyett a „Fizz”, az 5-tel oszthatók helyett a „Buzz”, a mindkettővel oszthatók helyett pedig a „FizzBuzz” szó jelenjen meg.

5. Írjunk egy szkriptet, amely egy adott pozitív egész számra kiírja az összes osztóját, megszámolja azokat, és megállapítja, hogy prímszám-e.

6. Készítsünk egy szkriptet, amely egymásba ágyazott ciklusokkal devizaátváltási táblázatot készít. Soronként egy összeg szerepeljen 100-tól 1000-ig (100-as lépésekben), mellette az átváltott értékek RON→EUR (5,0), RON→USD (4,5) és RON→GBP (5,8) árfolyamon. Használjunk `printf`-et a formázáshoz.

7. Valósítsunk meg egy egyszerű bankszámla-szimulációt. Kérjünk be egy kezdeti egyenleget, majd egy cikluson belül tranzakciókat (befizetés vagy kifizetés). Minden tranzakció után írjuk ki az aktuális egyenleget, és álljunk le, ha az egyenleg 0 alá csökkenne.

2.8.3. Függvények

1. Írjunk függvényeket a hőmérsékleti skálák közötti átváltáshoz: Celsius → Fahrenheit, Celsius → Kelvin és Fahrenheit → Celsius. Minden függvény kapja paraméterként a hőmérsékletet, és adja vissza az átváltott értéket. Mutassuk be a működésüket néhány jellegzetes értékkel (fagypont, emberi testhőmérséklet, forráspont).

2. Készítsünk egy függvényt, amely egy sztringet és egy maximális hosszt kap paraméterként, majd visszaadja a sztring rövidített változatát, ha az hosszabb a megadott maximumnál. A rövidített szöveg végére a „…” jel kerüljön úgy, hogy a visszaadott sztring hossza ne haladja meg a megadott maximumot.

3. Valósítsunk meg egy függvényt, amely egy sztringet kap paraméterként, és visszaadja, hogy az palindrom-e (visszafelé olvasva is ugyanaz, figyelmen kívül hagyva a szóközöket és az írásjeleket).

4. Készítsünk egy függvényt, amely egy sztringet kap paraméterként, és visszaadja a benne leggyakrabban előforduló karaktert, valamint annak előfordulási számát.

5. Írjunk egy `formatPrice()` függvényt, amely az összeget, a pénznemet és a tizedesjegyek számát kapja paraméterként, az utóbbi kettőnek alapértelmezett értékkel (`'RON'` és `2`).

   - Mutassuk be a működését különböző paraméterekkel.
   - Készítsünk mellé egy `applyDiscount()` függvényt is, amely alapértelmezetten 10%-os kedvezményt alkalmaz.
   - Egészítsük ki a szkriptet a `declare(strict_types=1);` deklarációval, és figyeljük meg, mi történik, ha hibás típusú argumentumot adunk át.

6. Írjunk egy `applyToEach(array $items, callable $fn): array` függvényt, amely egy tömb minden elemére alkalmaz egy átadott függvényt, majd visszaadja az eredménytömböt. Mutassuk be a működését az alábbi esetekben:

   - névtelen függvénnyel, amely megduplázza a számokat;
   - nyílfüggvény segítségével, amely egy külső változóban tárolt értéket ad hozzá minden elemhez;
   - a beépített `strtoupper()` függvénnyel sztringek nagybetűsítéséhez.

2.8.4. Tömbök

1. Adott egy tömb tanulók vizsgajegyeivel (1–10 között). Számítsuk ki az átlagot, keressük meg a legjobb és a leggyengébb jegyet, és írjuk ki, hány tanuló teljesített az átlag felett. Ne használjunk beépített rendezőfüggvényt.

2. Egy webáruház konfigurációját asszociatív tömbben tárolják, de nem minden kulcs van mindig megadva. A `??` operátorral állítsuk be a hiányzó értékeket:

   - ha nincs `'timeout'` kulcs, legyen `30`;
   - ha nincs `'currency'` kulcs, legyen `'RON'`;
   - ha nincs `'max_items'` kulcs, legyen `100`.

   Ezután a `??=` operátorral adjunk hozzá egy `'debug'` kulcsot `false` alapértékkel, ha még nem létezik.

3. Készítsünk egy asszociatív tömböt, amely országokat és azok fővárosait tartalmazza. Írjunk egy függvényt, amely egy országnevet kap paraméterként, és visszaadja a hozzá tartozó fővárost. Ha az ország nem szerepel a tömbben, jelenítsünk meg megfelelő hibaüzenetet.

4. Írjunk egy függvényt, amely egy tömböt kap paraméterként, és visszaadja a tömb második legnagyobb elemét. Ne használjunk beépített rendezőfüggvényeket.

5. Adott egy termékeket tartalmazó tömb, ahol minden elem egy asszociatív tömb (név, ár, raktáron lévő darabszám). Szűrjük ki az `array_filter()` segítségével a raktáron lévő termékeket, majd az `array_map()` függvénnyel számítsuk ki mindegyik 21%-os áfával növelt árát. Írjuk ki az eredményt.

6. Készítsünk egy szkriptet, amely egy egyszerű TODO-listát szimulál. A feladatok tárolására használjunk tömböt, és valósítsuk meg a következő műveleteket: új feladat hozzáadása, feladat törlése, feladat késznek jelölése, valamint a feladatok listázása állapot szerint (kész / folyamatban).

2.8.5. Dátum- és időkezelés

1. Írjunk egy szkriptet, amely kiírja az aktuális dátumot és időt különböző formátumokban (pl. „2026-05-15”, „15/05/2026”, „May 15, 2026”, „14:30:00”).

2. Készítsünk egy függvényt, amely kiszámolja és visszaadja egy személy életkorát években. A függvény paraméterként a születési dátumot kapja meg.

3. Írjunk egy függvényt, amely kiszámolja és visszaadja két dátum között eltelt napok számát. A dátumokat sztring formátumban kapja paraméterként.

4. Írjunk egy függvényt, amely meghatározza, hogy egy adott évben hány péntek 13-a van.

5. Készítsünk egy függvényt, amely kiszámítja két időpont között az eltelt időt órákban, percekben és másodpercekben. Az időpontokat sztring formátumban kapja paraméterként (pl. „14:30:00”).

6. Írjunk egy függvényt, amely megkapja a felhasználó születési dátumát, és visszaadja, hány nap van még a következő születésnapjáig. Ha a mai nap a születésnap, jelenítsünk meg gratuláló üzenetet. Használjuk a `DateTime` osztályt.

3. FEJEZET

OBJEKTUMORIENTÁLT PROGRAMOZÁS

3.1. Bevezetés

Az objektumorientált programozás (OOP) egy olyan programozási paradigma, amely a valós világ objektumainak modellezésére épül (Freeman–Robson 2021; Zandstra 2021). A PHP-ben az OOP támogatása az évek során jelentős fejlődésen ment keresztül, és mára a nyelv egyik központi elemévé vált.

Az OOP négy alapelvre épül:

- **Absztrakció** (abstraction): A lényeges tulajdonságok és viselkedések kiemelése, miközben a megvalósítás részletei rejtve maradnak.
- **Egységbezárás** (encapsulation): Az adatok és az azokon végzett műveletek egy egységbe szervezése, miközben a belső állapot közvetlen elérése korlátozott.
- **Öröklődés** (inheritance): Egy osztály átveheti és kiterjesztheti egy másik osztály tulajdonságait és metódusait, elősegítve a kód újrafelhasználását.
- **Polimorfizmus** (polymorphism): Különböző osztályok példányai egységes interfészen keresztül kezelhetők, miközben a konkrét viselkedés osztályonként eltérhet.

Ezek az alapelvek együttesen teszik az objektumorientált megközelítést hatékony programozási eszközzé. Az OOP segítségével a kód újrafelhasználhatóbbá, karbantarthatóbbá és könnyebben bővíthetővé válik, miközben a nagyobb rendszerek szervezettebb és egységesebb struktúrában valósíthatók meg.

3.2. Osztályok és objektumok

3.2.1. Osztály definiálása, objektumok létrehozása

Az osztályok a PHP-ben az objektumorientált programozás alapvető építőelemei. Egy osztály tulajdonságok és metódusok gyűjteménye, amely egy

fogalom (entitás) adatait és viselkedését írja le.

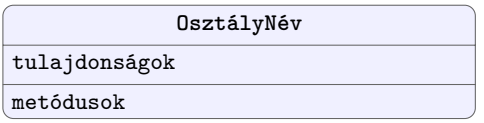
Szintaxis

```
class OsztályNév
{
    tulajdonságok
    metódusok
}
```

Ahol:

- `OsztályNév`: az osztály neve;
- `tulajdonságok`: az osztály adattagjai, amelyek az objektum állapotát tárolják;
- `metódusok`: az osztályhoz tartozó függvények, amelyek az objektum viselkedését definiálják.

A tulajdonságok és metódusok előtt láthatósági módosítókat (`public`, `protected`, `private`) használhatunk, amelyek segítségével szabályozható a hozzáférés. A fenti szintaxisnak megfelelő UML-diagram (Unified Modeling Language, Fowler 2003) a 3.1. ábrán látható.



3.1. ábra. *UML-osztálydiagram*

Az alábbi példában egy `Student` nevű osztály látható két adattaggal és egy metódussal:

```php
class Student
{
    public string $name;
    public int $age;

    public function introduce(): string
    {
        return "Szia, " . $this->name . " vagyok, " .
            $this->age . " éves.";
    }
}
```

3.1. kódpélda. *`Student` osztály*

Objektumok létrehozására (példányosítására) a `new` operátort használjuk, az adattagok elérésére pedig a `->` operátort:

```php
$student = new Student();
$student->name = "Kovács János";
$student->age = 20;

echo $student->introduce();
// Kimenet: Szia, Kovács János vagyok, 20 éves.
```

3.2. kódpélda. *`Student` osztály példányosítása*

Az osztályok tartalmazhatnak úgynevezett *magic metódusokat* is. Ezek neve két aláhúzással (`__`) kezdődik, és a PHP automatikusan hívja meg őket bizonyos eseményekkor. A leggyakoribb a `__construct()` konstruktor, amely az objektum létrehozásakor fut le, és jellemzően a tulajdonságok kezdőértékét állítja be. A magic metódusok megadása nem kötelező: ha egy osztályban nincs konstruktor, a `new` operátor akkor is működik, ahogy a fenti példában is látható. A konstruktort a 3.4. alfejezetben, a további magic metódusokat a 3.10.2. alfejezetben mutatjuk be.

3.2.2. A `$this` kulcsszó

A `$this` egy speciális változó, amely az aktuális objektumra mutat. Az osztályon belül a `$this` segítségével érhetjük el az objektum tulajdonságait és metódusait.

```php
class Student
{
    public string $name;
    public int $age;

    public function setName(string $name): void
    {
        $this->name = $name;  // az objektum name
            tulajdonságának beállítása
    }

    public function setAge(int $age): void
    {
        $this->age = $age;  // az objektum age
            tulajdonságának beállítása
    }

    public function introduce(): string
    {
        // az objektum tulajdonságainak elérése
```

```
        return "Szia, " . $this->name . " vagyok, " .
            $this->age . " éves.";
    }

    public function getInfo(): string
    {
        // saját metódus meghívása
        return $this->introduce() . " Örülök a
            találkozásnak!";
    }
}

$student = new Student();
$student->setName("Kovács János");
$student->setAge(20);

echo $student->introduce();
// Kimenet: Szia, Kovács János vagyok, 20 éves.

echo $student->getInfo();
// Kimenet: Szia, Kovács János vagyok, 20 éves. Örülök a
    találkozásnak!
```

3.3. kódpélda. *A `$this` kulcsszó használata*

A `$this` kulcsszó használata:

- `$this->propertyName`: az objektum tulajdonságának elérése;
- `$this->methodName()`: az objektum metódusának meghívása;
- csak nem statikus kontextusban használható;
- megkülönbözteti a lokális változókat (például a paramétereket) az objektum tulajdonságaitól.

Az alábbi példa szemlélteti, hogyan különbözteti meg a `$this` a lokális változókat az objektum tulajdonságaitól:

```
class Student
{
    public string $name;

    public function setName(string $name): void
    {
        // $name: a metódus paramétere (lokális változó)
        // $this->name: az objektum tulajdonsága
        $this->name = $name;
```

```
    }

    public function greet(string $greeting): string
    {
        // $greeting: csak a metóduson belül létezik
        // $this->name: az objektum tulajdonsága, az
            osztályon belül bárhol elérhető
        return $greeting . ", " . $this->name . " vagyok.";
    }
}

$student = new Student();
$student->setName("Eszter");

echo $student->greet("Szia");
// Kimenet: Szia, Eszter vagyok.
```

3.4. kódpélda. *Lokális változó és tulajdonság megkülönböztetése*

3.3. Tulajdonságok, metódusok és láthatóság

3.3.1. Láthatósági módosítók

A láthatósági módosítók szabályozzák, hogy a tulajdonságok és metódusok honnan érhetők el (zárójelben az UML-osztálydiagramokban használt jelölésük):

- `public` (+): az osztályon belülről és kívülről egyaránt elérhető;
- `protected` (#): az osztályon belülről és a leszármazott osztályokból érhető el;
- `private` (-): csak az osztályon belül érhető el.

```
class Student
{
    private string $id;
    public string $name;
    protected int $age;
    public function setId(string $id): void
    {
        $this->id = $id;
    }
    public function getId(): string
    {
```

```
        return $this->id;
    }
    public function setAge(int $age): void
    {
        $this->age = $age;
    }
    protected function getAge(): int
    {
        return $this->age;
    }
    public function introduce(): string
    {
        return "Szia, " . $this->name . " vagyok, " .
            $this->age . " éves.";
    }
    public function getInfo(): string
    {
        // belülről a private és protected tagok is elérhetők
        return "Név: " . $this->name . ", Kor: " .
            $this->getAge() . ", Azonosító: " .
            $this->getId();
    }
}
```

3.5. kódpélda. *`Student` osztály különböző láthatóságú tagokkal*

A fenti osztály használata és a láthatósági szabályok bemutatása:

```
$student = new Student();
$student->name = "Kovács János";  // működik, mert public
$student->setId("S12345");
$student->setAge(20);
echo $student->introduce();  // működik, mert public metódus
echo $student->getInfo();    // működik, mert public metódus
// echo $student->age;       // hiba: protected property nem
    érhető el
// echo $student->getAge();  // hiba: protected metódus nem
    érhető el
// echo $student->id;        // hiba: private property nem
    érhető el
// de a public getter használható:
echo $student->getId();  // működik: S12345
```

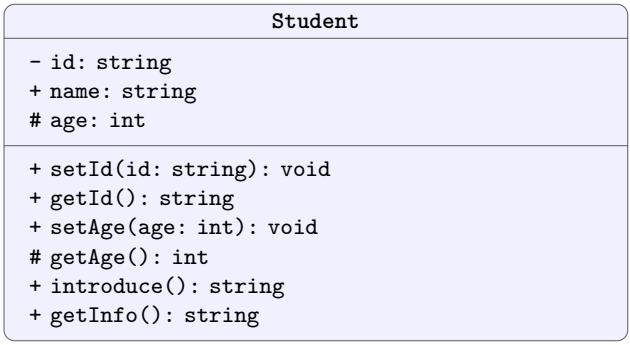
3.6. kódpélda. *`Student` osztály használata és láthatósági szabályok*

Ebben a példában:

- **name**: public tulajdonság, kívülről is elérhető és módosítható;
- **age**: protected tulajdonság, csak az osztályon belül és a leszármazott osztályokban érhető el;
- **id**: private tulajdonság, csak az osztályon belül érhető el, de public *getter* és *setter* metódusokon keresztül kívülről is hozzáférhető. A getter (`getId()`) a tulajdonság értékét adja vissza, a setter (`setId()`) pedig beállítja azt, így az osztály ellenőrizheti a beállítandó értéket;
- `getInfo()`: bemutatja, hogy az osztályon belül minden tag elérhető.

Ez a struktúra biztosítja, hogy az osztály belső működése rejtve maradjon a külvilág elől (*encapsulation*), miközben szabályozott hozzáférést biztosít az adatokhoz és műveletekhez.

Az osztály UML-diagramja a 3.2. ábrán látható.



3.2. ábra. *A `Student` osztály UML-diagramja*

3.3.2. Statikus tulajdonságok és metódusok

A `static` kulcsszóval deklarált tulajdonságok és metódusok az osztályhoz tartoznak, nem az egyes objektumpéldányokhoz. Elérésük a `OsztályNév::tag` szintaxissal történik, objektumpéldány létrehozása nélkül.

```
class Counter
{
    private static int $count = 0;

    public static function increment(): void
    {
        self::$count++;
```

```
    }

    public static function getCount(): int
    {
        return self::$count;
    }
}

Counter::increment();
Counter::increment();
echo Counter::getCount();  // 2
```

3.7. kódpélda. *Statikus tulajdonság és metódus*

Az osztályon belül a saját statikus tagjaira a `self::` kulcsszóval hivatkozunk, ahogy a fenti példában a `self::$count` esetében látható. A `self::` mindig azt az osztályt jelöli, amelyben a kód szerepel. Öröklődés esetén a PHP a `static::` kulcsszót is kínálja, amely a ténylegesen meghívott (leszármazott) osztályra hivatkozik. Ezt a 3.5.3. alfejezetben mutatjuk be.

A statikus tulajdonságok olyan adatok tárolására alkalmasak, amelyek minden objektumpéldány számára közösek. Például a féléves kreditkorlát minden hallgatóra egységesen érvényes, ezért osztályszintű tulajdonságként tárolható, míg a felvett kreditek hallgatónként eltérnek.

```
class Student
{
    private static int $maxCredits = 30;
    private int $enrolledCredits = 0;
    // Rövidített konstruktor (constructor property
       promotion),
    // részletesen a Konstruktor és destruktor alfejezetben
    public function __construct(public string $name) {}
    public function enroll(int $credits): bool
    {
        if ($this->enrolledCredits + $credits >
            self::$maxCredits) {
            return false;
        }
        $this->enrolledCredits += $credits;
        return true;
    }
    public static function setMaxCredits(int $max): void
    {
        self::$maxCredits = $max;
    }
}
```

```php
$s1 = new Student('Kovács János');
$s1->enroll(20);
$s1->enroll(15);  // false: 35 > 30
```

3.8. kódpélda. *Statikus tulajdonság a* `Student` *osztályban*

A statikus metódusok tipikus felhasználási területei:

- **Segédosztályok:** állapot nélküli műveletgyűjtemények (pl. `ViewHelper::escape()`);
- **Gyártómetódusok (factory methods):** névvel ellátott alternatív konstruktorok (Freeman–Robson 2021);
- **Osztályszintű megosztott állapot:** minden példány számára közösen elérhető értékek.

3.3.3. Csak olvasható (`readonly`) tulajdonságok

A `readonly` módosító (PHP 8.1-től érhető el) olyan tulajdonságokat hoz létre, amelyek értéke csak egyszer állítható be, utána már nem módosítható. A readonly tulajdonságok kizárólag inicializáláskor kaphatnak értéket.

```php
class Student
{
    public readonly string $id;
    public string $name;
    public int $age;
    public function setId(string $id): void
    {
        $this->id = $id;  // első értékadás: működik
    }
}
$student = new Student();
$student->setId("S12345");
$student->name = "Kovács János";
$student->age = 20;
echo $student->id;   // működik: S12345
// $student->id = "S99999";  // hiba: cannot modify readonly
    property
$student->name = "Varga Anna";  // működik, mert nem readonly
$student->age = 21;             // működik, mert nem readonly
```

3.9. kódpélda. *Readonly tulajdonság használata*

PHP 8.2-től `readonly` osztályok is létrehozhatók, ahol minden tulajdonság automatikusan `readonly`. A `readonly` tulajdonság csak az osztályon belülről, jellemzően a konstruktorban kaphat értéket:

```php
readonly class StudentRecord
{
    // Rövidített konstruktor (constructor property
        promotion),
    // részletesen a Konstruktor és destruktor alfejezetben
    public function __construct(
        public string $id,
        public string $name,
        public int $enrollmentYear
    ) {}
}

$record = new StudentRecord("S12345", "Kovács János", 2026);

// minden tulajdonság readonly, egyik sem módosítható többé:
// $record->name = "Varga Anna";  // hiba
// $record->enrollmentYear = 2025;  // hiba
```

3.10. kódpélda. *Readonly osztály*

A csak olvasható tulajdonságok előnyei:

- megváltoztathatatlan (immutable) objektumok létrehozása;
- biztonságosabb kód, mivel véd a véletlen módosításoktól;
- egyértelműbb szándék, mert jól látszik, hogy az adott érték nem változhat.

3.3.4. Property hooks

A property hookok (PHP 8.4-től érhető el) lehetővé teszik, hogy a getter és setter metódusok logikáját (a tulajdonság értékének lekérdezését és ellenőrzött beállítását) közvetlenül a tulajdonság deklarációjába ágyazzuk, így nincs szükség külön metódusokra.

```php
class Student
{
    public string $name {
        set(string $value) {
            // Automatikus formázás beállításkor
            $this->name = ucwords(strtolower(trim($value)));
        }
```

```
    }

    public int $age;
}

$student = new Student();
$student->name = "  kOVACS jAnos  ";  // a set hook
    automatikusan lefut
echo $student->name;  // Kimenet: Kovacs Janos

$student->name = "  vARGA aNNA  ";
echo $student->name;  // Kimenet: Varga Anna
```

3.11. kódpélda. *Property hooks használata*

Get és set hook kombinálása:

```
class Student
{
    private int $birthYear;
    public string $name;

    public int $age {
        get {
            // Életkor számítása lekérdezéskor
            return date('Y') - $this->birthYear;
        }
        set(int $value) {
            // Születési év számítása beállításkor
            $this->birthYear = date('Y') - $value;
        }
    }
}

$student = new Student();
$student->name = "Kovács János";
$student->age = 20;  // set hook: kiszámítja a születési
    évet (az aktuális évből)

echo $student->age;  // get hook: az aktuális évből számolja
    vissza
// Kimenet: 20
```

3.12. kódpélda. *Számított tulajdonság property hookkal*

Előnyök:

- tisztább szintaxis, mivel nincs szükség külön getter és setter metódusokra;

- automatikus logika beépítése, például validáció, formázás vagy számítás;
- jobb olvashatóság, mert a logika közvetlenül a tulajdonságnál látható.

3.3.5. Aszimmetrikus láthatóság

Az aszimmetrikus láthatóság (PHP 8.4-től érhető el) lehetővé teszi, hogy egy tulajdonság olvasási és írási láthatósága eltérjen egymástól. Az alábbi példában az `id`, `name` és `age` tulajdonságok kívülről olvashatók (`public`), ugyanakkor csak az osztályon belül módosíthatók (`private set`):

```php
class Student
{
    public private(set) string $id;
    public private(set) string $name;
    public private(set) int $age;

    public function setData(string $id, string $name, int
        $age): void
    {
        $this->id = $id;       // belülről módosítható
        $this->name = $name;   // belülről módosítható
        $this->age = $age;     // belülről módosítható
    }

    public function haveBirthday(): void
    {
        $this->age++;  // belülről módosítható
    }
}

$student = new Student();
$student->setData("S12345", "Kovács János", 20);

// Olvasás: működik (public)
echo $student->name;  // Kimenet: Kovács János
echo $student->age;   // Kimenet: 20

// Írás: hiba (private set)
// $student->name = "Varga Anna";  // Hiba: Cannot modify
    private(set) property
// $student->age = 21;             // Hiba: Cannot modify
    private(set) property

// de metóduson keresztül módosítható:
$student->haveBirthday();  // működik
echo $student->age;        // Kimenet: 21
```

3.13. kódpélda. *Aszimmetrikus láthatóság*

Előnyök:

- egyszerűbb megoldás, mert nincs szükség külön getter metódusokra;
- biztonságosabb működés, mivel a tulajdonságok csak ellenőrzött metódusokon keresztül módosíthatók;
- tisztább kód, mert a szándék már a deklarációból egyértelmű.

3.3.6. Null-safe műveletek

A null-safe operátor (`?->`) lehetővé teszi, hogy biztonságosan hívjunk meg metódusokat vagy érjük el tulajdonságokat olyan objektumokon, amelyek `null` értékűek is lehetnek, anélkül hogy hibát kapnánk.

A következő példában egy `Student` osztályt használunk, amelynek opcionális `Address` típusú tulajdonsága van:

```php
class Address
{
    public string $city;
    public string $street;

    public function __construct(string $city, string $street)
    {
        $this->city = $city;
        $this->street = $street;
    }
}

class Student
{
    public string $name;
    public ?Address $address;  // Lehet null

    public function __construct(string $name, ?Address
        $address = null)
    {
        $this->name = $name;
        $this->address = $address;
    }
}
```

3.14. kódpélda. *Az Address és Student osztályok*

Hagyományosan, ha egy objektum lehet `null`, minden egyes hozzáférés előtt külön ellenőrzést kell végezni:

```php
$student = new Student("Varga Péter", null);

if ($student->address !== null) {
    echo $student->address->city;
} else {
    echo "Nincs cím";  // Ez fog kiíródni
}
```

3.15. kódpélda. *Hagyományos null ellenőrzés*

A `?->` operátor automatikusan kezeli a `null` értékeket, így nincs szükség külön ellenőrzésekre:

```php
$student = new Student("Varga Péter", null);

$city = $student->address?->city;                    // null
    (nem dob hibát)
$city = $student->address?->city ?? "Nincs cím";  // "Nincs
    cím"
echo $city;
```

3.16. kódpélda. *Null-safe operátor használata*

A null-safe operátor többször is láncolható egy kifejezésben. Ha a lánc bármelyik eleme `null`, az egész kifejezés értéke `null` lesz:

```php
// Hagyományos módszer (hosszú és nehezen olvasható)
$country = null;
if ($student->address !== null) {
    if ($student->address->city !== null) {
        if ($student->address->city->country !== null) {
            $country =
                $student->address->city->country->name;
        }
    }
}

// Null-safe operátorral (egyszerűbb és tömörebb)
$country = $student->address?->city?->country?->name;
```

3.17. kódpélda. *Null-safe operátor összehasonlítása*

3.4. Konstruktor és destruktor

3.4.1. Konstruktor alapok

A konstruktor (`__construct()`) egy speciális metódus, amely az objektum létrehozásakor automatikusan meghívódik. Fő feladata az objektum inicializálása és kezdeti állapotának beállítása.

```php
class Student
{
    private string $id;
    private string $name;
    private int $age;

    public function __construct(string $id, string $name,
        int $age)
    {
        $this->id = $id;
        $this->name = $name;
        $this->age = $age;
    }

    public function introduce(): string
    {
        return "Szia, {$this->name} vagyok, {$this->age}
            éves, azonosító: {$this->id}.";
    }
}

// Objektum létrehozása konstruktorral
$student = new Student("S12345", "Molnár Eszter", 20);
echo $student->introduce();
// Kimenet: Szia, Molnár Eszter vagyok, 20 éves, azonosító:
    S12345.
```

3.18. kódpélda. *Konstruktor használata*

A konstruktor jellemzői:

- automatikusan meghívódik az objektum létrehozásakor (`new` kulcsszóval);
- paramétereket fogadhat az objektum inicializálásához;
- egy osztálynak csak egy konstruktora lehet;
- nem kötelező definiálni. Ha az osztálynak nincs konstruktora, a szülőosztály konstruktora fut le, ha pedig az sincs, az objektum paraméterek nélkül jön létre.

3.4.2. Destruktor

A destruktor (`__destruct()`) az objektum megsemmisítésekor hívódik meg, általában akkor, amikor az objektumra már nincs több hivatkozás, vagy a szkript futása véget ér.

```php
class Student
{
    private string $name;
    public function __construct(string $name)
    {
        $this->name = $name;
        echo "{$this->name} hallgató létrehozva.<br>";
    }
    public function __destruct()
    {
        echo "{$this->name} hallgató objektum
            megsemmisítve.<br>";
    }
}
function createStudent()
{
    $student = new Student("Péter");
    echo "A függvényen belül.<br>";
    // A függvény végén az objektum megsemmisül
}
createStudent();
echo "A függvényhívás után.<br>";
// Kimenet:
// Péter hallgató létrehozva.
// A függvényen belül.
// Péter hallgató objektum megsemmisítve.
// A függvényhívás után.
```

3.19. kódpélda. *Destruktor használata*

A destruktor jellemzői:

- automatikusan meghívódik az objektum megsemmisülésekor;
- nem fogad paramétereket;
- hasznos lehet fájlok bezárására vagy adatbázis-kapcsolatok lezárására;
- ritkábban használjuk, mint a konstruktort;
- a meghívás pontos időpontja nem mindig kiszámítható.

3.4.3. Modern konstruktorlehetőségek

Constructor property promotion

A constructor property promotion (PHP 8.0-tól érhető el) rövidebb szintaxist biztosít a tulajdonságok deklarálására és inicializálására. A konstruktor paraméterei közvetlenül tulajdonságként is deklarálhatók, így nincs szükség külön tulajdonságdefiníciókra és értékadásokra.

```php
// Hagyományos mód (PHP < 8.0)
class LegacyStudent
{
    private string $id;
    private string $name;
    private int $age;
    public function __construct(string $id, string $name,
        int $age)
    {
        $this->id = $id;
        $this->name = $name;
        $this->age = $age;
    }
}
// Constructor property promotion (PHP 8.0+)
class Student
{
    public function __construct(
        private string $id,
        private string $name,
        private int $age
    ) {
        // Nincs szükség külön értékadásokra,
        // a paraméterek automatikusan tulajdonságokká válnak
    }
    public function introduce(): string
    {
        return "Szia, {$this->name} vagyok, {$this->age}
            éves, azonosító: {$this->id}.";
    }
}
$student = new Student("S12345", "Molnár Eszter", 20);
echo $student->introduce();
```

3.20. kódpélda. *Hagyományos és constructor property promotion*

A property promotion előnyei:

- Tömörebb kód: kevesebb ismétlődő sor szükséges.
- Kevesebb hibalehetőség: az inicializálás automatikusan történik.
- Jobb olvashatóság: a konstruktor paraméterei és a tulajdonságok egy helyen láthatók.
- Egyszerűbb karbantarthatóság: könnyebb új tulajdonságokat hozzáadni.

Property promotion különböző módosítókkal:

```php
class Student
{
    public function __construct(
        public readonly string $id,      // Readonly
            tulajdonság
        public string $name,             // Public tulajdonság
        private int $age,                // Private
            tulajdonság
        protected string $email          // Protected
            tulajdonság
    ) {}

    public function getAge(): int
    {
        return $this->age;  // A private tulajdonság az
            osztályon belül elérhető
    }
}

$student = new Student(
    "S12345",
    "Kovács János",
    20,
    "kovacs.janos@example.com"
);

echo $student->name;  // Működik (public)
echo $student->id;    // Működik (readonly, de olvasható)

// $student->id = "S999";  // Hiba: readonly property
// echo $student->age;     // Hiba: private property
```

3.21. kódpélda. *Property promotion különböző módosítókkal*

A property promotion hagyományos tulajdonságokkal is kombinálható:

```php
class Student
{
    private array $grades = [];  // Hagyományos tulajdonság

    public function __construct(
        public readonly string $id,  // Promoted tulajdonság
        public string $name,         // Promoted tulajdonság
        private int $age             // Promoted tulajdonság
    ) {
        // További inicializáló logika is írható ide
        $this->grades = [
            "Webprogramozás" => 0,
            "Adatbázisok" => 0
        ];
    }

    public function addGrade(string $subject, int $grade):
        void
    {
        $this->grades[$subject] = $grade;
    }
}
```

3.22. kódpélda. *Promoted és hagyományos tulajdonságok*

3.5. Öröklődés

3.5.1. Mi az öröklődés?

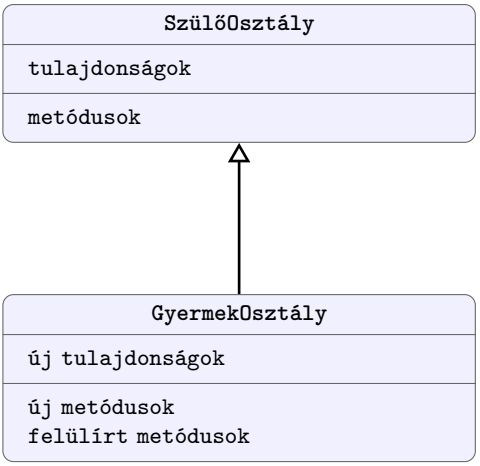
Az *öröklődés* (*inheritance*) az objektumorientált programozás egyik alapvető koncepciója, amely lehetővé teszi, hogy egy új osztályt egy meglévő osztály alapján hozzunk létre. Az új osztályt *leszármazott osztálynak* vagy *gyermekosztálynak* (*derived class*, *child class*), a kiindulási osztályt pedig *szülőosztálynak* vagy *alaposztálynak* (*parent class*, *base class*) nevezzük. A gyermekosztály örökli a szülőosztály tulajdonságait és metódusait.

Az öröklődés egy „is-a” kapcsolatot fejez ki: a gyermekosztály a szülőosztály egy speciális esete. Például egy `Student` objektum egyben `Person` is: minden hallgató személy, de nem minden személy hallgató.

Szintaxis

```
class GyermekOsztály extends SzülőOsztály
{
    // Új tulajdonságok és metódusok
    // Meglévő metódusok felülírása
}
```

Két osztály közötti öröklődési kapcsolatot a 3.3. ábra szemléltet. A kapcsolatot a gyermekosztálytól a szülőosztály felé mutató, üres háromszögben végződő folytonos vonallal jelöljük.



3.3. ábra. *Öröklődés UML-diagramja*

Az alábbi példában a `Student` osztály a `Person` osztályból származik:

```php
class Person
{
    protected string $name;
    protected int $age;

    public function __construct(string $name, int $age)
    {
        $this->name = $name;
        $this->age = $age;
    }

    public function introduce(): string
    {
```

```php
        return "Szia, {$this->name} vagyok, {$this->age}
            éves.";
    }
}

class Student extends Person
{
    private string $id;

    public function __construct(string $id, string $name,
        int $age)
    {
        parent::__construct($name, $age);
        $this->id = $id;
    }

    public function introduce(): string
    {
        return parent::introduce() . " A hallgatói
            azonosítóm: {$this->id}.";
    }

    public function study(): string
    {
        return "{$this->name} tanul.";
    }
}

$person = new Person("Kovács János", 30);
echo $person->introduce();
// Kimenet: Szia, Kovács János vagyok, 30 éves.

$student = new Student("S12345", "Molnár Eszter", 20);
echo $student->introduce();
// Kimenet: Szia, Molnár Eszter vagyok, 20 éves. A hallgatói
    azonosítóm: S12345.

echo $student->study();
// Kimenet: Molnár Eszter tanul.
```

3.23. kódpélda. *Öröklődés példa:* `Person` *és* `Student` *osztályok*

A fenti példában:

- A `Student` osztály örökli a `Person` osztály tulajdonságait és metódusait.
- A `Student` osztály egy új `$id` tulajdonsággal bővíti a `Person` osztályt.

- Az `introduce()` metódus felülírja a szülőosztály implementációját, miközben felhasználja annak eredményét is.
- A `study()` metódus csak a `Student` osztályban érhető el.

3.5.2. A `parent::` használata

A `parent::` kulcsszó lehetővé teszi, hogy a gyermekosztályból elérjük a szülőosztály metódusait és konstansait. Ez különösen hasznos akkor, amikor egy metódust felül szeretnénk írni, de az eredeti funkcionalitás egy részét is meg kívánjuk tartani. Az alábbiakban a `Student` példáján keresztül mutatjuk be a használatát.

1. **Konstruktor meghívásakor:**

```php
public function __construct(string $id, string
    $name, int $age)
{
    parent::__construct($name, $age);
    $this->id = $id;
}
```

Itt a `parent::__construct($name, $age)` a `Person` osztály konstruktorát hívja meg, és inicializálja az örökölt `$name` és `$age` tulajdonságokat. A hívás a gyermekosztály konstruktorának elején történik, még az új `$id` tulajdonság inicializálása előtt.

2. **Metódus felülírásakor:**

```php
public function introduce(): string
{
    return parent::introduce() . " A hallgatói
        azonosítóm: {$this->id}.";
}
```

Itt a `parent::introduce()` meghívja a `Person` osztály `introduce()` metódusát, majd az eredményhez hozzáfűzi a hallgatói azonosítót. Ez lehetővé teszi a szülőosztály funkcionalitásának bővítését anélkül, hogy azt teljes egészében újra kellene írni.

A `parent::` használatának előnyei:

- Lehetővé teszi a szülőosztály funkcionalitásának újrafelhasználását.
- Csökkenti a kódduplikációt.
- Segít fenntartani a konzisztenciát az osztályhierarchiában.
- Lehetővé teszi a fokozatos módosítást és kiterjesztést.

Megjegyzés

- A `parent::` csak olyan metódusokra, konstansokra és statikus tulajdonságokra hivatkozhat, amelyek `public` vagy `protected` láthatósággal rendelkeznek a szülőosztályban. A példánytulajdonságok elérése nem `parent::` használatával történik, hanem a `$this->` operátoron keresztül.
- A `study()` metódus nem használja a `parent::` kulcsszót, mert ez egy új metódus a `Student` osztályban, amely nem létezik a `Person` osztályban.
- A `protected` láthatóságú `$name` és `$age` tulajdonságok elérhetők a `Student` osztályban, ezért használhatók a `study()` metódusban.

3.5.3. A `static::` kulcsszó (late static binding)

A `self::` jelentése fordítási időben dől el, a `static::` jelentése futási időben. Ha egy szülőosztály statikus metódusa a `self::` kulcsszóval hivatkozik egy másik statikus tagra, akkor mindig a szülőosztály tagját kapja, még akkor is, ha a metódust a leszármazott osztályon keresztül hívtuk meg. A `static::` ezzel szemben mindig a meghívott osztályt veszi figyelembe. Ezt *késői statikus kötésnek* (*late static binding*) nevezzük.

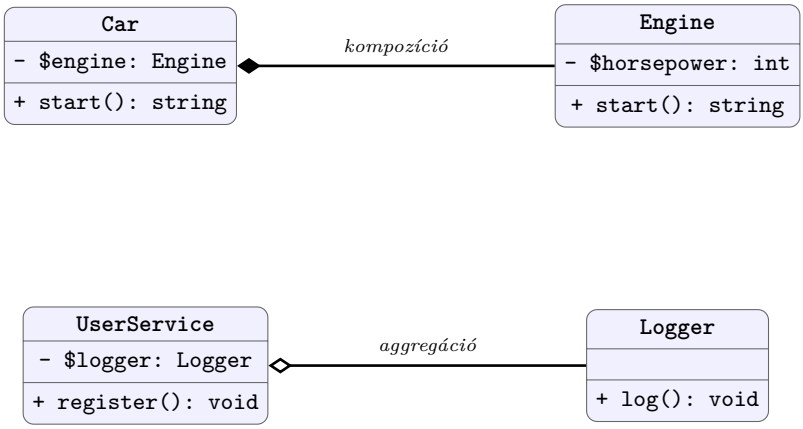
```
class Model
{
    protected static string $table = 'models';
    public static function tableWithSelf(): string
    {
        return self::$table;
    }
    public static function tableWithStatic(): string
    {
        return static::$table;
    }
}
class Student extends Model
{
    protected static string $table = 'students';
}
echo Student::tableWithSelf();     // models
echo Student::tableWithStatic();   // students
```

3.24. kódpélda. *A self:: és static:: különbsége*

A `static::` különösen hasznos gyártómetódusoknál (lásd a 3.3.2. alfejezetet): a `return new static();` utasítással a szülőosztályban írt metódus is a leszármazott osztály példányát hozza létre.

3.6. Kompozíció és aggregáció

Az öröklődés mellett az objektumok közötti kapcsolat másik fontos formája a *kompozíció* (*composition*) és az *aggregáció* (*aggregation*). Mindkettő „has-a” típusú kapcsolatot jelent, szemben az öröklődés „is-a” kapcsolatával. A különbség az életciklusban rejlik: kompozíció esetén a tartalmazott objektum nem létezhet a befoglaló objektum nélkül, míg aggregáció esetén önállóan is létezhet (lásd a 3.4. ábrát).



3.4. ábra. *Kompozíció (teli rombusz) és aggregáció (üres rombusz) UML-ben*

3.6.1. Kompozíció

Kompozíció esetén az egyik objektum tartalmaz egy másikat, és annak életciklusa teljes mértékben a befoglaló objektumhoz kötődik. Ha a befoglaló objektum megszűnik, a tartalmazott objektum is megszűnik. Az alábbi példában egy `Car` objektum tartalmaz egy `Engine` objektumot: a motor nem létezhet az autó nélkül.

```
class Engine
{
    public function __construct(private int $horsepower) {}

    public function start(): string
    {
```

```
        return "{$this->horsepower} lóerős motor beindult.";
    }
}

class Car
{
    private Engine $engine;

    public function __construct(int $horsepower)
    {
        // A motor a kocsi része, külön nem létezik
        $this->engine = new Engine($horsepower);
    }

    public function start(): string
    {
        return $this->engine->start();
    }
}

$car = new Car(150);
echo $car->start(); // 150 lóerős motor beindult.
```

3.25. kódpélda. *Kompozíció példa*

3.6.2. Aggregáció

Aggregáció esetén a tartalmazott objektum önállóan is létezhet. A kapcsolat lazább: a tartalmazott objektumot kívülről adjuk át (Dependency Injection). Az alábbi példában a `Logger` objektumot kívülről adjuk át a `UserService`-nek, így a logger önállóan is használható más osztályokban.

```
class Logger
{
    public function log(string $message): void
    {
        echo "[LOG] $message\n";
    }
}

class UserService
{
    // A Logger önállóan is létezhet, kívülről kerül átadásra
    public function __construct(private Logger $logger) {}

    public function register(string $name): void
```

```
    {
        // ... regisztrációs logika
        $this->logger->log("Új felhasználó: $name");
    }
}

$logger = new Logger();
$service = new UserService($logger);
$service->register("Kovács Anna");
```

3.26. kódpélda. *Aggregáció példa*

3.6.3. Mikor melyiket használjuk?

- **Öröklődés**: Ha az osztályok között valódi „is-a” kapcsolat áll fenn (pl. a `Student` egy `Person`).
- **Kompozíció**: Ha az egyik objektum a másik szerves részét képezi, és önállóan nincs értelme (pl. az `Engine` a `Car` nélkül).
- **Aggregáció**: Ha a függőséget kívülről adjuk át, és az objektum önállóan is felhasználható más helyeken (pl. a `Logger` több osztályban is szerepelhet).

Megjegyzés

A modern PHP-fejlesztésben, különösen a 9. fejezetben tárgyalt MVC-architektúrában, a kompozíció és az aggregáció (például Dependency Injection formájában) általában előnyt élvez az öröklődéssel szemben. Az „öröklődés helyett kompozíció” (*composition over inheritance*) elv lazán csatolt, könnyen tesztelhető és rugalmasan karbantartható kód kialakítását segíti (Martin 2008).

3.7. Absztrakt osztályok és interfészek

3.7.1. Absztrakt osztályok

Az absztrakt osztályok olyan osztályok, amelyeket nem lehet közvetlenül példányosítani, hanem más osztályok szülőosztályaként szolgálnak. Céljuk, hogy közös tulajdonságokat és viselkedéseket határozzanak meg a belőlük származó osztályok számára.

Szintaxis

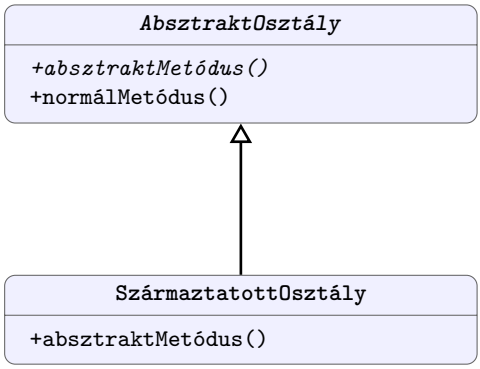
```
abstract class AbsztraktOsztály
{
    // Absztrakt metódus
    abstract public function absztraktMetódus();

    // Normál metódus
    public function normálMetódus()
    {
        // ...
    }
}

class SzármaztatottOsztály extends AbsztraktOsztály
{
    public function absztraktMetódus()
    {
        // Az absztrakt metódus implementációja
    }
}
```

Az absztrakt osztályokat az `abstract` kulcsszóval definiáljuk, és tartalmazhatnak absztrakt metódusokat is. Az absztrakt metódusokat szintén az `abstract` kulcsszó jelöli, és nincs implementációjuk. Emellett az absztrakt osztályok tartalmazhatnak konkrét (nem absztrakt) metódusokat is, amelyeket a szokásos módon implementálunk.

A fenti szintaxis UML-megfelelője a 3.5. ábrán látható. Megfigyelhető, hogy az absztrakt osztályok és metódusok dőlt betűvel vannak jelölve.



3.5. ábra. *Absztrakt osztály és származtatott osztály UML-diagramja*

Az absztrakt osztályokra vonatkozó főbb szabályok:

- Absztrakt osztályból nem lehet példányt létrehozni.
- Az absztrakt osztályból származó osztálynak minden absztrakt metódust implementálnia kell, kivéve, ha maga is absztrakt.
- Az absztrakt osztályok használhatják a láthatósági módosítókat (`public`, `protected`, `private`) metódusaik és tulajdonságaik esetében.
- Az absztrakt metódusok nem lehetnek privátak, mivel a leszármazott osztályoknak implementálniuk kell azokat.

```php
abstract class Shape
{
    protected string $color;

    public function __construct(string $color)
    {
        $this->color = $color;
    }

    abstract public function getArea(): float;

    public function getColor(): string
    {
        return $this->color;
    }
}

class Square extends Shape
{
    private float $side;

    public function __construct(string $color, float $side)
    {
        parent::__construct($color);
        $this->side = $side;
    }

    public function getArea(): float
    {
        return $this->side * $this->side;
    }
}
```

```php
$square = new Square("piros", 5);
echo "Terület: " . $square->getArea() . ", Szín: " .
    $square->getColor();
// Kimenet: Terület: 25, Szín: piros
```

3.27. kódpélda. *Absztrakt osztály példa*

A példában a `Shape` egy absztrakt osztály, amely közös `$color` tulajdonságot és egy konkrét `getColor()` metódust definiál. Emellett deklarál egy absztrakt `getArea()` metódust is. A `Square` osztály kiterjeszti a `Shape` osztályt, és implementálja a kötelező `getArea()` metódust. Konstruktorában meghívja a szülőosztály konstruktorát.

Ez a struktúra lehetővé teszi a közös funkcionalitás (színkezelés) újrafelhasználását, miközben kikényszeríti a specifikus viselkedés (területszámítás) implementálását a leszármazott osztályokban. Az absztrakt osztály használata így egységes kezelést biztosít az alakzatok számára, miközben megőrzi az egyedi implementációk rugalmasságát.

Mikor használjunk absztrakt osztályt?

Absztrakt osztályok alkalmazása a következő esetekben előnyös:

- **Közös alapfunkcionalitás definiálása:** Ha több kapcsolódó osztály közös viselkedésen osztozik, de az nem elég konkrét ahhoz, hogy az osztály önállóan példányosítható legyen.
- **Tervezési sablon biztosítása:** Ha egy alapvető struktúrát szeretnénk adni a leszármazott osztályoknak kötelezően implementálandó metódusokkal, valamint opcionálisan kész implementációval.
- **A kód újrafelhasználhatóságának növelése:** Ha a kapcsolódó osztályok között szeretnénk csökkenteni a kódismétlést, miközben megőrizzük a rugalmasságot.

3.7.2. Interfészek

Az interfész egyfajta „szerződés”, amely meghatározza, hogy egy osztálynak milyen nyilvános metódusokat kell implementálnia. Az interfészek lehetővé teszik, hogy közös viselkedést definiáljunk különböző osztályok között anélkül, hogy közöttük öröklődési kapcsolat állna fenn.

Szintaxis

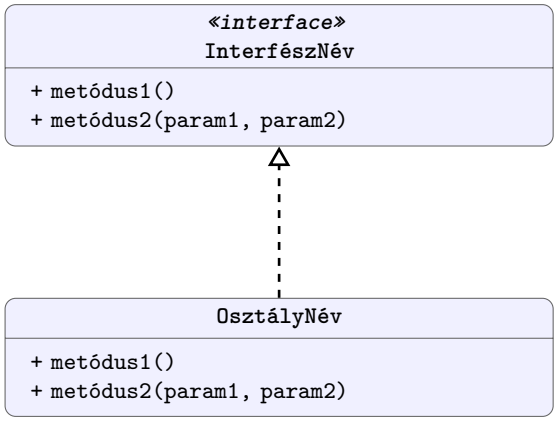
```
interface InterfészNév
{
    const KONSTANS = érték;

    public function metódus1();
    public function metódus2(param1, param2);
}

class OsztályNév implements InterfészNév
{
    public function metódus1()
    {
        // Implementáció
    }

    public function metódus2(param1, param2)
    {
        // Implementáció
    }
}
```

Az interfész definiálásához az `interface` kulcsszót használjuk, az implementálást pedig az `implements` kulcsszóval jelöljük. A fenti szintaxis UML-megfelelője a 3.6. ábrán látható. A kapcsolatot a megvalósító osztálytól az interfész felé mutató, üres háromszögben végződő szaggatott vonal jelöli.



3.6. ábra. *Interfész és az azt megvalósító osztály UML-diagramja*

Főbb szabályok:

- Interfészben csak nyilvános (`public`) metódusok és konstansok deklarálhatók.
- Az interfészben deklarált metódusoknak nincs implementációjuk.
- Egy osztály több interfészt is implementálhat, vesszővel elválasztva.
- Az interfészt implementáló osztálynak az összes deklarált metódust implementálnia kell.
- Az interfészek kiterjeszthetik egymást az `extends` kulcsszó segítségével.

Nézzünk egy egyszerű példát az interfész használatára:

```php
interface Drawable
{
    public function draw(): string;
}

class Circle implements Drawable
{
    private int $radius;

    public function __construct(int $radius)
    {
        $this->radius = $radius;
    }

    public function draw(): string
    {
        return "Kör rajzolása, sugár: {$this->radius}";
    }
}

$circle = new Circle(10);
echo $circle->draw();  // Kimenet: Kör rajzolása, sugár: 10
```

3.28. kódpélda. *Egyszerű interfész példa*

A fenti példában a `Drawable` interfész definiál egy `draw()` metódust, amelyet a `Circle` osztály implementál. Ez biztosítja, hogy minden „rajzolható” objektum rendelkezzen `draw()` metódussal.

Most lássunk egy példát a `Student` osztály esetén:

```php
interface Enrollable
{
    public function enroll(string $course): void;
    public function getEnrolledCourses(): array;
}

class Student implements Enrollable
{
    private string $id;
    private string $name;
    private array $courses = [];

    public function __construct(string $id, string $name)
    {
        $this->id = $id;
        $this->name = $name;
    }

    public function enroll(string $course): void
    {
        $this->courses[] = $course;
    }

    public function getEnrolledCourses(): array
    {
        return $this->courses;
    }
}

$student = new Student("S12345", "Molnár Eszter");
$student->enroll("Webprogramozás");
$student->enroll("Adatbázisok");

echo "Felvett kurzusok: " . implode(", ",
    $student->getEnrolledCourses());
// Kimenet: Felvett kurzusok: Webprogramozás, Adatbázisok
```

3.29. kódpélda. *Interfész példa a `Student` osztály esetén*

Ebben a példában az `Enrollable` interfész két metódust ír elő: `enroll()` és `getEnrolledCourses()`. A `Student` osztály megvalósítja ezt az interfészt, biztosítva, hogy minden hallgató képes legyen kurzusokra jelentkezni és lekérdezni azokat.

Mikor használjunk interfészeket?

Az interfészek használata különösen előnyös a következő esetekben:

- **Közös viselkedés definiálása:** Ha különböző, nem feltétlenül kapcsolódó osztályoknak azonos metódusokat kell biztosítaniuk, miközben a megvalósításuk eltérhet.
- **Lazán csatolt kód kialakítása:** Az interfészek segítségével az osztályok konkrét implementációk helyett szerződésekre támaszkodnak, ami elősegíti a komponensek függetlenségét és a függőségek absztrakcióját (Dependency Injection).
- **Tesztelhetőség javítása:** Az interfészek megkönnyítik az egységtesztek (unit tesztek) írását, mivel a valódi implementációk helyett mock objektumok használhatók.

3.8. Objektumok vizsgálata és összehasonlítása

Az objektumorientált kódban gyakran szükség van annak eldöntésére, hogy egy objektum melyik osztályhoz tartozik, illetve hogy két objektum azonos-e. A PHP mindkét feladathoz több eszközt kínál.

3.8.1. Objektum osztályának vizsgálata

PHP-ben több módszer is rendelkezésre áll egy objektum osztályának vizsgálatára:

- Az `instanceof` operátor ellenőrzi, hogy egy objektum egy adott osztály példánya-e, vagy megvalósít-e egy adott interfészt.
- A `get_class()` függvény visszaadja egy objektum osztályának nevét.
- Az `is_a()` függvény ellenőrzi, hogy egy objektum egy adott osztály példánya-e, vagy annak leszármazottja-e.
- Az `is_subclass_of()` függvény ellenőrzi, hogy egy osztály egy másik osztály leszármazottja-e.

```php
$student = new Student("S12345", "Molnár Eszter", 21);
var_dump($student instanceof Student);  // bool(true)
var_dump($student instanceof Person);   // bool(true)
echo get_class($student);               // Student
var_dump(is_a($student, 'Person'));     // bool(true)
var_dump(is_subclass_of('Student', 'Person')); // bool(true)
```

3.30. kódpélda. *Objektum osztályának vizsgálata*

3.8.2. Objektumok összehasonlítása

PHP-ben az objektumokat kétféleképpen hasonlíthatjuk össze:

- Az == (egyenlőség) operátor szerint két objektum akkor egyenlő, ha ugyanannak az osztálynak a példányai, és tulajdonságaik értéke megegyezik.
- Az === (azonosság) operátor azt vizsgálja, hogy két változó ugyanarra az objektumpéldányra hivatkozik-e.

```php
class Student
{
    private string $id;
    private string $name;
    public function __construct(string $id, string $name)
    {
        $this->id = $id;
        $this->name = $name;
    }
    public function getName(): string
    {
        return $this->name;
    }
    public function setName(string $name): void
    {
        $this->name = $name;
    }
}
$student1 = new Student("S001", "Kovács Anna");
$student2 = new Student("S001", "Kovács Anna");
$student3 = $student1;
var_dump($student1 == $student2);   // bool(true)  - azonos
    tulajdonságok
var_dump($student1 === $student2);  // bool(false) -
    különböző példányok
var_dump($student1 === $student3);  // bool(true)  -
    ugyanarra a példányra hivatkoznak
// Tulajdonság módosítása az egyik példányon
$student3->setName("Nagy Péter");
echo $student1->getName();  // Kimenet: Nagy Péter
echo $student2->getName();  // Kimenet: Kovács Anna
```

3.31. kódpélda. *Objektumok összehasonlítása*

Ebben a példában:

- A `$student1` és `$student2` különböző objektumpéldányok, de azonos tulajdonságokkal rendelkeznek, ezért az `==` operátor igaz, míg az `===` operátor hamis értéket ad.
- A `$student1` és `$student3` ugyanarra az objektumpéldányra hivatkoznak, ezért mind az `==`, mind az `===` operátor igaz értéket ad.
- A `$student3` objektumon végzett módosítások a `$student1` példányon is megjelennek, mivel ugyanarra a memóriabeli objektumra hivatkoznak.
- A `$student2` példány független az előzőektől, ezért annak állapota nem változik.

3.9. Polimorfizmus

A *polimorfizmus* (többalakúság) egy objektumorientált programozási elv, amely lehetővé teszi, hogy egy adott interfészt vagy szülőosztályt használva különböző típusú objektumokat egységes módon kezeljünk, miközben azok megőrzik saját, specifikus viselkedésüket.

Nézzünk egy példát a polimorfizmusra a korábbi `Shape` osztály alapján:

```
abstract class Shape
{
    abstract public function getArea(): float;
}

class Circle extends Shape
{
    private float $radius;

    public function __construct(float $radius)
    {
        $this->radius = $radius;
    }

    public function getArea(): float
    {
        return pi() * $this->radius * $this->radius;
    }
}

class Rectangle extends Shape
{
```

```php
    private float $width;
    private float $height;

    public function __construct(float $width, float $height)
    {
        $this->width = $width;
        $this->height = $height;
    }

    public function getArea(): float
    {
        return $this->width * $this->height;
    }
}

function printArea(Shape $shape): void
{
    echo "A terület: " . $shape->getArea() . "\n";
}

$circle = new Circle(5);
$rectangle = new Rectangle(4, 6);

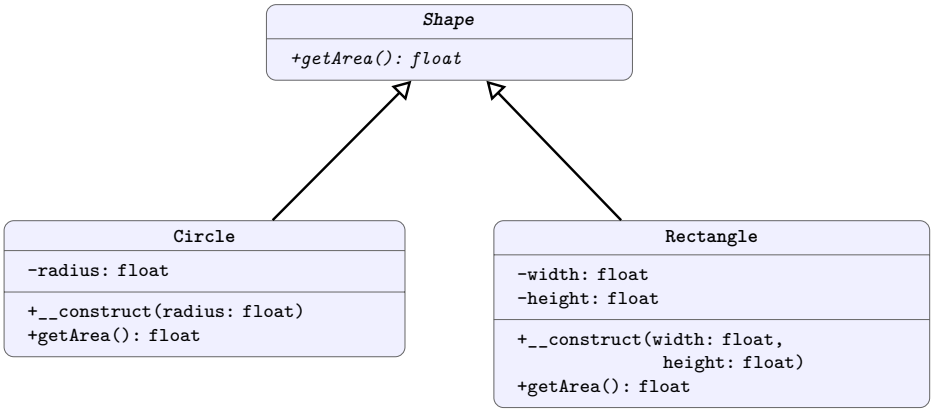
printArea($circle);      // Kimenet: A terület:
    78.539816339745
printArea($rectangle);  // Kimenet: A terület: 24
```

3.32. kódpélda. *Példa polimorfizmusra*

Ebben a példában:

- A `Shape` absztrakt osztály egy közös `getArea()` metódust definiál.
- A `Circle` és `Rectangle` osztályok kiterjesztik a `Shape` osztályt, és saját logikájuk szerint valósítják meg a `getArea()` metódust.
- A `printArea()` függvény bármilyen `Shape` típusú objektumot fogad, majd meghívja annak `getArea()` metódusát.
- A polimorfizmus lehetővé teszi, hogy ugyanazt a függvényt különböző alakzatok esetén is használjuk a terület kiszámítására és megjelenítésére.

A 3.7. ábra a példában szereplő osztályhierarchiát szemlélteti.



3.7. ábra. *A Shape, Circle és Rectangle osztályok öröklési hierarchiája*

A polimorfizmus előnyei:

- **Rugalmasság:** Új típusok vezethetők be a meglévő kód módosítása nélkül.
- **A kód újrafelhasználhatósága:** Ugyanaz a logika különböző típusú objektumokon is alkalmazható.
- **Absztrakció:** A kód a közös viselkedésre fókuszál, nem pedig a konkrét implementációkra.
- **Karbantarthatóság:** A változtatások helyben tarthatók, és nem terjednek szét a teljes kódbázisban.
- **Kiterjeszthetőség:** Új osztályok könnyen illeszthetők a meglévő struktúrába.

3.10. Metódustúlterhelés és magic metódusok

3.10.1. Metódustúlterhelés (method overloading)

A PHP nem támogatja a hagyományos értelemben vett metódustúlterhelést, amely más objektumorientált nyelvekben (pl. Java, C++) lehetővé teszi, hogy ugyanazzal a névvel, de eltérő paraméterlistákkal több metódust definiáljunk egy osztályon belül. A PHP azonban alternatív megoldásokat kínál, amelyekkel hasonló funkcionalitás valósítható meg.

Az egyik megközelítés az opcionális paraméterek használata:

```php
class Student
{
    public function displayInfo($name, $age = null, $grade =
        null)
    {
        echo "Név: $name";
        if ($age !== null) {
            echo ", Kor: $age";
        }
        if ($grade !== null) {
            echo ", Jegy: $grade";
        }
        echo "\n";
    }
}

$student = new Student();
$student->displayInfo("Kovács János");
$student->displayInfo("Varga Anna", 20);
$student->displayInfo("Nagy Péter", 22, "5");
```

3.33. kódpélda. *Opcionális paraméterek használata*

Ebben a példában a `displayInfo()` metódus rugalmasan kezeli a paramétereket, lehetővé téve egy, kettő vagy három adat megjelenítését.

A túlterhelés szimulálására változó számú paraméterek (`...`) is használhatók (lásd a 2.5.4. alfejezetet):

```php
class Student
{
    public function enroll(...$courses)
    {
        echo "Felvett kurzusok: " . implode(", ", $courses)
            . "\n";
    }
}

$student = new Student();
$student->enroll("Matematika");
$student->enroll("Fizika", "Kémia");
$student->enroll("Biológia", "Angol", "Rajz");
```

3.34. kódpélda. *Változó számú paraméter használata*

Ez a megoldás lehetővé teszi tetszőleges számú kurzus átadását az `enroll()` metódusnak.

3.10.2. Magic metódusok

A PHP számos *magic* metódust kínál. Ezek speciális nevű metódusok, amelyek automatikusan meghívódnak bizonyos események bekövetkezésekor. A magic metódusok neve mindig két aláhúzásjellel kezdődik (`__`). Segítségükkel az osztályok viselkedése különböző helyzetekben testreszabható. Az összes fontosabb magic metódust a 3.1. táblázat foglalja össze.

3.1. táblázat. *PHP magic metódusok összefoglalása*

Metódus	Mikor hívódik meg
`__construct()`	Objektum létrehozásakor
`__destruct()`	Objektum megsemmisítésekor
`__call($name, $args)`	Nem létező vagy nem elérhető metódus meghívásakor
`__callStatic($name, $args)`	Nem létező vagy nem elérhető statikus metódus meghívásakor
`__get($name)`	Nem elérhető tulajdonság olvasásakor
`__set($name, $value)`	Nem elérhető tulajdonság írásakor
`__isset($name)`	`isset()` vagy `empty()` használatakor nem elérhető tulajdonságon
`__unset($name)`	`unset()` használatakor nem elérhető tulajdonságon
`__toString()`	Objektum sztringként történő használatakor
`__invoke(...)`	Objektum függvényként való meghívásakor
`__clone()`	A `clone` kulcsszó használatakor

`__construct()` és `__destruct()`

Ezeket a metódusokat már korábban tárgyaltuk. A `__construct()` az objektum létrehozásakor, míg a `__destruct()` az objektum megsemmisítésekor hívódik meg.

```
class Student
{
    private string $name;
    private string $id;
    public function __construct(string $name, string $id)
    {
        $this->name = $name;
        $this->id = $id;
```

```
        echo "$name hallgató felvételre került (azonosító:
            $id).\n";
    }
    public function __destruct()
    {
        echo "$this->name hallgató elvégezte a tanulmányait
            (azonosító: $this->id).\n";
    }
}
$student = new Student("Kovács Anna", "S12345");
unset($student);
```

3.35. kódpélda. *`__construct()` és `__destruct()` használata*

A példában a konstruktor inicializálja a hallgató nevét és azonosítóját, majd egy üzenetet jelenít meg az objektum létrehozásakor. A destruktor az objektum megszüntetésekor automatikusan lefut, és jelzi az objektum megszűnését.

`__call()` és `__callStatic()`

Ezek a metódusok akkor hívódnak meg, amikor egy objektumon vagy osztályon nem létező vagy nem elérhető metódust próbálunk meghívni.

```
class Student
{
    public function __call(string $name, array $arguments):
        void
    {
        echo "Nem létező metódus hívása: '$name',
            argumentumok: "
            . implode(', ', $arguments) . "\n";
    }
    public static function __callStatic(string $name, array
        $arguments): void
    {
        echo "Nem létező statikus metódus hívása: '$name',
            argumentumok: "
            . implode(', ', $arguments) . "\n";
    }
}
```

3.36. kódpélda. *`__call()` és `__callStatic()` használata*

A `__call()` metódus objektumszintű, míg a `__callStatic()` statikus metódushívások kezelésére szolgál. Ezeket gyakran használják dinamikus API-k megvalósítására.

`__get()` és `__set()`

Ezek a metódusok lehetővé teszik a nem elérhető tulajdonságok olvasásának és módosításának testreszabását.

```php
class Student
{
    private array $data = [];

    public function __get(string $name): mixed
    {
        return $this->data[$name]
            ?? "A(z) '$name' tulajdonság nem létezik.";
    }

    public function __set(string $name, mixed $value): void
    {
        $this->data[$name] = $value;
    }
}
```

3.37. kódpélda. *`__get()` és `__set()` használata*

A példában a nem létező vagy nem elérhető tulajdonságok értékei egy belső tömbben tárolódnak. Ez a megközelítés rugalmas adatkezelést tesz lehetővé, ugyanakkor körültekintően érdemes alkalmazni, mert csökkentheti a kód átláthatóságát.

`__isset()` és `__unset()`

Ezek a metódusok a dinamikusan kezelt vagy nem elérhető tulajdonságok vizsgálatára és eltávolítására szolgálnak.

```php
class Student
{
    private array $data = [
        'name' => 'Kovács János',
        'age' => 20
    ];

    public function __isset(string $name): bool
    {
        return isset($this->data[$name]);
    }

    public function __unset(string $name): void
```

```
    {
        unset($this->data[$name]);
    }
}
```

3.38. kódpélda. *__isset() és __unset() használata*

Az `__isset()` metódus lehetővé teszi annak ellenőrzését, hogy egy dinamikusan kezelt tulajdonság rendelkezik-e értékkel, míg a `__unset()` eltávolítja azt a belső adattárolóból.

`__toString()`

Ez a metódus meghatározza, hogyan jelenjen meg az objektum szöveges környezetben.

```
class Student
{
    private string $name;
    private string $id;

    public function __construct(string $name, string $id)
    {
        $this->name = $name;
        $this->id = $id;
    }

    public function __toString(): string
    {
        return "Hallgató: {$this->name} (azonosító:
            {$this->id})";
    }
}
```

3.39. kódpélda. *__toString() használata*

A `__toString()` metódus automatikusan meghívódik, amikor az objektumot szövegként próbáljuk megjeleníteni, például az `echo` utasítás használatakor.

`__invoke()`

Ez a metódus lehetővé teszi, hogy egy objektum függvényként legyen meghívható.

```php
class GradeCalculator
{
    private array $weights;

    public function __construct(array $weights)
    {
        $this->weights = $weights;
    }

    public function __invoke(array $scores): float
    {
        $total = 0;

        foreach ($scores as $subject => $score) {
            $total += $score * ($this->weights[$subject] ??
                1);
        }

        return $total / array_sum($this->weights);
    }
}
```

3.40. kódpélda. *`__invoke()` használata*

Ez a megoldás különösen hasznos lehet *callback*ek, *middleware*-ek vagy funkcionális jellegű komponensek megvalósításakor.

`__clone()`

A `__clone()` metódus akkor hívódik meg, amikor egy objektumot a `clone` kulcsszó segítségével másolunk.

```php
class Student
{
    public string $name;
    public array $courses = [];

    public function __construct(string $name)
    {
        $this->name = $name;
    }

    public function __clone(): void
    {
        $this->courses = [];
```

```
        echo "{$this->name} hallgató klónozva, üres
            kurzuslistával.\n";
    }
}

$s1 = new Student("Kovács János");
$s1->courses[] = "Webprogramozás";

$s2 = clone $s1;  // Kimenet: Kovács János hallgató
    klónozva, üres kurzuslistával.
echo count($s1->courses);  // 1
echo count($s2->courses);  // 0
```

3.41. kódpélda. *`__clone()` használata*

A `clone` kulcsszó először sekély másolatot (shallow copy) készít az objektumról. Az új objektum megkapja az eredeti objektum tulajdonságainak értékeit. Ha valamelyik tulajdonság egy másik objektumra hivatkozik, a másolat ugyanarra az objektumra fog hivatkozni, mint az eredeti, a hivatkozott objektum tehát nem másolódik. Ezután lefut a másolaton a `__clone()` metódus, amelyben a tulajdonságok értéke módosítható, a közös objektumhivatkozások pedig lecserélhetők új, független példányokra. A példában a klónozott objektum üres kurzuslistát kap, az eredeti listája változatlan marad.

3.10.3. Mikor használjunk magic metódusokat?

A magic metódusok az alábbi esetekben lehetnek hasznosak, azonban érdemes őket mértékkel alkalmazni, mert csökkenthetik a kód átláthatóságát és megnehezíthetik a karbantartást:

- késleltetett betöltés (lazy loading): erőforrás-igényes műveletek késleltetett végrehajtása;
- dinamikus metódus- és tulajdonságkezelés;
- objektumok sztringgé alakítása;
- objektumok függvényként való használata;
- klónozás testreszabása.

3.10.4. Magic konstansok

A PHP számos beépített, ún. *magic konstanst* biztosít, amelyek értéke a környezettől és a végrehajtási kontextustól függően automatikusan változik. Ezeket a konstansokat a PHP előre definiálja, és jellemzően hibakeresésre, naplózásra vagy diagnosztikai célokra használjuk.

A legfontosabb magic konstansok:

- `__CLASS__`: az aktuális osztály neve;
- `__METHOD__`: az aktuális osztály és metódus neve;
- `__FUNCTION__`: az aktuális függvény neve;
- `__LINE__`: a forrásfájl aktuális sora;
- `__FILE__`: a fájl teljes elérési útja és neve;
- `__DIR__`: a fájlt tartalmazó könyvtár elérési útja;
- `__NAMESPACE__`: az aktuális névtér neve.

```php
namespace School;

class Student
{
    public function displayInfo(): void
    {
        echo "Osztály: " . __CLASS__ . "\n";      //
            School\Student
        echo "Metódus: " . __METHOD__ . "\n";     //
            School\Student::displayInfo
        echo "Függvény: " . __FUNCTION__ . "\n"; //
            displayInfo
        echo "Névtér: " . __NAMESPACE__ . "\n";  // School
        echo "Fájl: " . __FILE__ . "\n";          //
            C:\xampp\htdocs\student.php
        echo "Sor: " . __LINE__ . "\n";           // a sor
            száma a fájlban, pl. 13
    }
}

$student = new Student();
$student->displayInfo();
```

3.42. kódpélda. *Magic konstansok használata*

3.11. Traitek

A traitek (PHP 5.4+) olyan kódmegosztási mechanizmust biztosítanak, amely lehetővé teszi metódusok és tulajdonságok újrafelhasználását különböző

osztályok között a `use` kulcsszó segítségével. A traitek a PHP egyszeres öröklődési modelljét egészítik ki, és különösen akkor hasznosak, amikor több, egymástól független osztályban azonos funkcionalitásra van szükség, de ezt nem célszerű közös szülőosztály létrehozásával megvalósítani.

Tipikus példa lehet egy olyan funkcionalitás, mint az azonosítógenerálás, a naplózás vagy az időbélyegek kezelése, amelyre különböző osztályokban (például `Student`, `Teacher` vagy `Course`) egyaránt szükség lehet.

Szintaxis

```
trait TraitNév
{
    // tulajdonságok és metódusok
}

class OsztályNév
{
    use TraitNév;           // egy trait használata
    use Trait1, Trait2;     // több trait használata
}
```

A traitek legfontosabb jellemzői:

- nem példányosíthatók közvetlenül, tehát nem használható a `new TraitNév()` forma;
- horizontális kódmegosztást tesznek lehetővé, nem hierarchikus öröklődést;
- egy osztály egyszerre több traitet is használhat;
- kombinálhatók hagyományos osztályörökléssel.

3.11.1. A trait alapvető használata

Az alábbi példa egy `Identifiable` traitet mutat be, amely egyedi azonosítók generálásáért felel. A traitet a `Student` osztály használja.

```
// Trait definiálása
trait Identifiable
{
    private string $id;

    public function generateId(): void
    {
        $this->id = uniqid('ID_', true);
    }
```

```
    public function getId(): string
    {
        return $this->id;
    }
}

// Trait használata a Student osztályban
class Student
{
    use Identifiable;

    public function __construct(
        public string $name,
        public int $age
    ) {
        $this->generateId();
    }

    public function introduce(): string
    {
        return "Hallgató: {$this->name} (azonosító:
            {$this->getId()})";
    }
}

$student = new Student("Kovács János", 20);

echo $student->introduce();
// Kimenet: Hallgató: Kovács János (azonosító: ID_...)

echo $student->getId();
// A trait metódusa közvetlenül elérhető
```

3.43. kódpélda. *Egyszerű trait definiálása és használata*

A fenti példában:

- az `Identifiable` trait az azonosító kezeléséhez szükséges logikát tartalmazza;
- a `Student` osztály a `use` kulcsszó segítségével használja a traitet;
- a trait metódusai (`generateId()`, `getId()`) úgy viselkednek, mintha közvetlenül az osztályban lennének definiálva;
- a traitben definiált `private $id` tulajdonság az osztály példányának részeként működik.

3.11.2. Több trait használata

Egy osztály egyszerre több traitet is használhat. Ebben az esetben a traiteket vesszővel elválasztva kell megadni a `use` kulcsszó után.

```php
trait Timestampable
{
    private DateTime $createdAt;

    public function setCreatedAt(): void
    {
        $this->createdAt = new DateTime();
    }

    public function getCreatedAt(): string
    {
        return $this->createdAt->format('Y-m-d H:i:s');
    }
}

// Az Identifiable trait az előző példából származik
class Student
{
    use Identifiable, Timestampable;

    public function __construct(public string $name)
    {
        $this->generateId();     // Identifiable trait
            metódusa
        $this->setCreatedAt();   // Timestampable trait
            metódusa
    }
}

$student = new Student("Kovács János");

echo $student->getId();
echo $student->getCreatedAt();
```

3.44. kódpélda. *Több trait használata*

Ebben a példában a `Student` osztály egyszerre két traitet használ:

- az `Identifiable` trait az egyedi azonosító kezelését biztosítja;
- a `Timestampable` trait a létrehozás időpontjának kezeléséért felel.

Mindkét trait metódusai elérhetők a `Student` osztályban. A PHP a

traitek kódját fordítási időben beilleszti az osztályba, ezért azok futásidőben az osztály saját metódusaiként viselkednek.

3.11.3. Traitek és öröklődés kombinálása

A traitek jól kombinálhatók a hagyományos osztályöröklődéssel. Ez lehetővé teszi, hogy egy osztály egyszerre örököljön egy szülőosztályból, miközben további funkcionalitásokat traitek segítségével kapjon meg.

```php
trait Gradeable
{
    private array $grades = [];

    public function addGrade(string $subject, int $grade):
        void
    {
        $this->grades[$subject] = $grade;
    }

    public function getAverage(): float
    {
        return empty($this->grades)
            ? 0.0
            : array_sum($this->grades) /
                count($this->grades);
    }
}

class Person
{
    public function __construct(public string $name) {}
}

class Student extends Person
{
    use Gradeable;
}

$student = new Student("Varga Anna");

$student->addGrade("Webprogramozás", 8);
$student->addGrade("Adatbázisok", 9);

echo $student->getAverage(); // 8.5
```

3.45. kódpélda. *Trait és öröklődés együttes használata*

A fenti példában a `Student` osztály örökli a `Person` osztály tulajdonságait, miközben a `Gradeable` trait segítségével jegykezelési funkcionalitást is kap.

Ez jól szemlélteti a traitek egyik legfontosabb előnyét: a meglévő osztályhierarchiák új funkcionalitásokkal bővíthetők anélkül, hogy az öröklési struktúrát módosítani kellene. A jegykezelési logika így más osztályokban is könnyen újrafelhasználható.

3.11.4. Metódusnevek ütközése és konfliktuskezelés

Ha két trait azonos nevű metódust definiál, névütközés keletkezik. A PHP erre beépített megoldást kínál az `insteadof` és az `as` kulcsszavak segítségével.

```php
trait EnglishGreeting
{
    public function greet(): string
    {
        return "Hello!";
    }
}

trait HungarianGreeting
{
    public function greet(): string
    {
        return "Szia!";
    }
}

class Student
{
    use EnglishGreeting, HungarianGreeting {
        EnglishGreeting::greet insteadof HungarianGreeting;
        HungarianGreeting::greet as greetHungarian;
    }

    public function __construct(public string $name) {}
}

$student = new Student("Kovács János");

echo $student->greet();
echo $student->greetHungarian();
```

3.46. kódpélda. *Konfliktuskezelés traitek között*

Ebben a példában mindkét trait definiál egy `greet()` metódust. A konfliktus feloldásához:

- az `insteadof` kulcsszó meghatározza, hogy alapértelmezetten melyik trait metódusa legyen használatban;
- az `as` kulcsszó álnevet ad a másik metódusnak, így mindkét implementáció elérhető marad.

Ennek eredményeként:

- a `greet()` metódus az angol köszönést adja vissza;
- a `greetHungarian()` metódus pedig a magyar változatot.

3.12. Felsorolások (enum típusok)

A felsorolás (PHP 8.1+) egy speciális adattípus, amely előre meghatározott értékek halmazát írja le. Használatával biztosítható, hogy egy változó csak előre definiált, érvényes értéket vehessen fel, ezáltal növelve a típusbiztonságot és csökkentve a hibalehetőségeket.

Szintaxis

```
// Egyszerű (unit) enum
enum FelsorolásNév
{
    case Érték1;
    ...
    case ÉrtékN;
}

// Értékkel rendelkező (backed) enum
enum FelsorolásNév: string|int
{
    case Érték1 = 'ertek1';
    ...
    case ÉrtékN = 'ertekN';
}
```

A felsorolások főbb jellemzői:

- Típusbiztonságot garantálnak: csak előre definiált értékek használhatók.
- Lehetnek egyszerű (*unit enum*) vagy értékkel rendelkező (*backed enum*) típusok.
- Minden enum eset rendelkezik `->name` tulajdonsággal, backed enum esetén pedig a `->value` is elérhető.
- Gyakori felhasználási területek: státuszok, szerepkörök, prioritások.

3.12.1. Egyszerű (unit) enum

Az egyszerű felsorolás kizárólag `case`-eket tartalmaz, saját érték hozzárendelése nélkül, ezért nem alakítható közvetlenül `string` vagy `int` típussá:

```
enum StudentStatus
{
    case Active;
    case Suspended;
    case Graduated;
    case Dropout;
}

$status = StudentStatus::Active;

echo $status->name;  // "Active"

if ($status === StudentStatus::Active) {
    echo "Aktív hallgató";
}

$label = match($status) {
    StudentStatus::Active    => "Aktív",
    StudentStatus::Suspended => "Felfüggesztett",
    StudentStatus::Graduated => "Végzett",
    StudentStatus::Dropout   => "Lemorzsolódott",
};
```

3.47. kódpélda. *Egyszerű felsorolás használata*

A példában a `StudentStatus` enum négy lehetséges hallgatói állapotot definiál. Az enum esetek összehasonlítása az `===` operátorral történik, mivel objektumként viselkednek. A `match` szerkezet segítségével az enum értékekhez könnyen rendelhetünk ember által olvasható megnevezéseket.

3.12.2. Értékkel rendelkező (backed) enum

A backed enum esetén minden `case` egy skalárértékhez (`string` vagy `int`) van kötve. Ez különösen hasznos adatbázisban történő tárolás vagy API-kommunikáció esetén.

```
enum StudentStatus: string
{
    case Active    = 'active';
    case Suspended = 'suspended';
    case Graduated = 'graduated';
}
```

```php
echo StudentStatus::Active->value;  // "active"

$status = StudentStatus::from('graduated');  //
    StudentStatus::Graduated
$status = StudentStatus::tryFrom('invalid'); // null
```

3.48. kódpélda. *Felsorolás sztringértékkel*

A `from()` metódus `ValueError` kivételt dob, ha az adott érték nem létezik az enum esetei között, míg a `tryFrom()` metódus hiba helyett `null` értékkel tér vissza.

Egészszám-alapú enum esetén hasonló működés figyelhető meg:

```php
enum Grade: int
{
    case Excellent    = 10;
    case Good         = 8;
    case Satisfactory = 7;
    case Sufficient   = 6;
    case Pass         = 5;
    case Fail         = 4;
}

$grade = Grade::from(8);

echo $grade->value;  // 8
echo $grade->name;   // "Good"
```

3.49. kódpélda. *Felsorolás egészszám-értékkel*

A `Grade` enum lehetővé teszi a jegyek típusbiztos kezelését, miközben azok egyszerű egész számként tárolhatók például adatbázisban is.

3.12.3. Metódusok és konstansok a felsorolásokban

Az enum típusok metódusokat és konstansokat is tartalmazhatnak, így az enumhoz kapcsolódó üzleti logika közvetlenül a felsorolásban helyezhető el.

```php
enum StudentStatus: string
{
    case Active    = 'active';
    case Suspended = 'suspended';
    case Graduated = 'graduated';

    public const DEFAULT = self::Active;

    public function label(): string
    {
```

```
        return match($this) {
            self::Active    => 'Aktív',
            self::Suspended => 'Felfüggesztett',
            self::Graduated => 'Végzett',
        };
    }

    public function canEnroll(): bool
    {
        return $this === self::Active;
    }

    public static function activeStatuses(): array
    {
        return [self::Active];
    }
}

echo StudentStatus::Active->label();                    //
    "Aktív"
echo StudentStatus::Active->canEnroll() ? 'Igen' : 'Nem';

$default = StudentStatus::DEFAULT;
$active  = StudentStatus::activeStatuses();
```

3.50. kódpélda. *Felsorolás metódusokkal és konstansokkal*

A példában a `label()` metódus ember által olvasható megnevezést ad az enum értékéhez, míg a `canEnroll()` metódus üzleti logikát valósít meg. A `DEFAULT` konstans alapértelmezett állapotot definiál, az `activeStatuses()` statikus metódus pedig az aktív állapotokat adja vissza tömb formájában.

3.12.4. Az összes érték lekérése

A `cases()` metódus tömb formájában adja vissza az enum összes esetét:

```
$allStatuses = StudentStatus::cases();

foreach ($allStatuses as $status) {
    echo $status->name . ': ' . $status->value . "\n";
}
```

3.51. kódpélda. *Felsorolás értékeinek lekérése*

A `cases()` metódus különösen hasznos lehet például legördülő listák generálásakor, validáció során vagy adminisztrációs felületeken megjelenített választható opciók előállításához.

3.13. Automatikus osztálybetöltés

Nagyobb projektekben az osztályok és interfészek száma gyorsan megnövekedhet. Az automatikus osztálybetöltés (*autoloading*) lehetővé teszi, hogy a PHP a szükséges fájlokat automatikusan betöltse, így nincs szükség azok kézi betöltésére. Ennek köszönhetően a kód áttekinthetőbbé, könnyebben karbantarthatóvá és jobban skálázhatóvá válik.

3.13.1. Fájlok betöltése (`include`, `require`)

Az automatikus osztálybetöltés megértéséhez érdemes először áttekinteni a hagyományos megközelítést. A PHP több utasítást is biztosít külső fájlok betöltésére:

- `include`: betölti a megadott fájlt; ha a fájl nem található, figyelmeztetést (*warning*) generál, de a program futása folytatódik;
- `require`: az `include` utasításhoz hasonló, azonban hiányzó fájl esetén végzetes hibát (*fatal error*) okoz, és a program futása leáll;
- `include_once` és `require_once`: az előző utasítások olyan változatai, amelyek biztosítják, hogy ugyanaz a fájl csak egyszer kerüljön betöltésre.

```php
// Student.php
class Student
{
    public function __construct(
        private string $name,
        private int $age
    ) {}

    public function getInfo(): string
    {
        return "Hallgató: {$this->name}, kor: {$this->age}";
    }
}

// main.php
require_once 'Student.php';

$student = new Student("Kovács János", 22);
echo $student->getInfo();
```

3.52. kódpélda. *Hagyományos osztálybetöltés*

Több osztály esetén ez a megközelítés gyorsan nehezen kezelhetővé válik, mivel minden fájlt manuálisan kell betölteni, és a betöltések sorrendje is problémát okozhat.

3.13.2. Automatikus osztálybetöltés

Az automatikus osztálybetöltés lényege, hogy amikor a PHP egy még nem betöltött osztályra, interfészre vagy traitre hivatkozik, automatikusan megkeresi és betölti a hozzá tartozó fájlt. Így nincs szükség külön `require` vagy `include` utasításokra minden egyes osztály esetén.

A folyamatot egy úgynevezett *autoloader* függvény végzi, amely az osztály neve alapján meghatározza, melyik fájlt kell betölteni.

`spl_autoload_register()`

Az `spl_autoload_register()` függvény a Standard PHP Library (SPL) része, és lehetőséget biztosít egy vagy több autoloader függvény regisztrálására. Amikor a PHP egy még nem betöltött osztályra vagy interfészre hivatkozik, automatikusan meghívja a regisztrált autoloader függvényeket.

Szintaxis

```
spl_autoload_register(
    callable $autoload_function = null,
    bool $throw = true,
    bool $prepend = false
): bool
```

Paraméterek:

- `$autoload_function` (opcionális): a regisztrálandó autoloader függvény;
- `$throw` (opcionális): ha értéke `true`, hiba esetén kivételt dob; alapértelmezett értéke `true`;
- `$prepend` (opcionális): ha értéke `true`, az autoloader a regisztrált függvények listájának elejére kerül, így elsőként fut le; alapértelmezett értéke `false`.

Visszatérési érték: `true`, ha a regisztráció sikeres, egyébként `false`.

Az autoload folyamat lépései:

1. A PHP egy még nem betöltött osztályra, interfészre vagy traitre hivatkozik.

2. A PHP sorban meghívja a regisztrált autoloader függvényeket.
3. Az autoloader az osztály neve alapján megkeresi és betölti a megfelelő fájlt.
4. Ha egyik autoloader sem tudja betölteni az osztályt, a PHP hibát jelez.

Autoloader regisztrálása

Az alábbi példában az osztályok automatikusan a `classes/` könyvtárból töltődnek be. Az autoloader az osztály neve alapján határozza meg a betöltendő fájl nevét.

```php
// autoloader.php
spl_autoload_register(function (string $className): void {
    $file = 'classes/' . $className . '.php';

    if (file_exists($file)) {
        require $file;
    }
});

// main.php
require 'autoloader.php';

$student = new Student("Kovács János", 22);
echo $student->getInfo();

$course = new Course("Webprogramozás", 3);
echo $course->getInfo();
```

3.53. kódpélda. *Autoloader regisztrálása*

A példában a `Student` és `Course` osztályok fájljait nem kell külön kézzel betölteni. Amikor a PHP először találkozik egy osztály nevével, az autoloader automatikusan megkeresi és betölti a megfelelő fájlt.

Modern autoloading: PSR-4 és Composer

A modern PHP-projektek többségében már nem kézzel írt autoloadert használnak, hanem a Composer csomagkezelő által generált megoldást, amely a PSR-4-szabványra épül.

A PSR-4 (*PHP Standards Recommendation*) meghatározza, hogyan képezzük le a névtereket a fájlrendszer struktúrájára. Például az `App\Model\Student` osztály a `src/Model/Student.php` fájlnak felelhet meg.

Ez a megközelítés egységes projektstruktúrát biztosít, csökkenti a manuális konfiguráció szükségességét, és jól skálázható alapot nyújt nagyobb alkalmazások fejlesztéséhez.

A Composer és a PSR-4-alapú autoloading részletes bemutatására a 8.4. alfejezetben kerül sor.

3.14. Névterek

A névterek (*namespaces*) a PHP 5.3-ban bevezetett eszközök a kód logikai szervezésére. Segítségükkel elkerülhetők a névütközések, és nagyobb projektek esetén is áttekinthető marad az osztályok, függvények és konstansok struktúrája.

Szintaxis

```
// Névtér deklarálása
namespace NévtérNév;
namespace NévtérNév\AlNévtér;
```

3.14.1. Névtér deklarálása

A névtereket a `namespace` kulcsszóval deklaráljuk, amelynek egy PHP-fájlban (a `declare` utasítást kivéve) az első utasításnak kell lennie. A névterek hierarchikusan is szervezhetők: több szintből állhatnak, amelyeket backslash (\) karakter választ el egymástól.

```php
namespace MyProject\Models;

class User
{
    private int $id;
    private string $name;

    public function __construct(int $id, string $name)
    {
        $this->id   = $id;
        $this->name = $name;
    }

    public function getName(): string
    {
        return $this->name;
    }
}
```

3.54. kódpélda. *Névtér deklarálása*

Megjegyzés

Bár technikailag lehetséges, egy fájlban több névtér használata általában nem ajánlott, mert csökkentheti a kód átláthatóságát, és megnehezítheti az autoloading működésének követését.

```php
namespace MyProject\Models;

class User {}

namespace MyProject\Services;

class UserService {}
```

3.55. kódpélda. *Több névtér egy fájlban*

3.14.2. Névterek importálása és álnevek

A `use` kulcsszó segítségével osztályokat, függvényeket vagy konstansokat importálhatunk egy másik névtérből. Az `as` kulcsszóval *álnév* (alias) is megadható, amely különösen hasznos névütközések elkerülésére vagy hosszú nevek rövidítésére.

```php
use MyProject\Models\User;
use MyProject\Services\UserService as Service;
use function MyProject\Helpers\formatDate;
use const MyProject\Config\MAX_USERS;

$user    = new User(1, 'Kovács János');
$service = new Service();

echo formatDate('2024-01-01');
echo MAX_USERS;
```

3.56. kódpélda. *Névterek importálása és álnevek*

3.14.3. Névtérelemek elérése

A névterekben definiált elemeket (osztályokat, függvényeket és konstansokat) háromféleképpen érhetjük el:

- **Nem minősített név** (*Unqualified Name*): csak az elem neve szerepel névtér-előtag nélkül. Akkor használható, ha az elem ugyanabban a névtérben található, vagy korábban `use` segítségével importáltuk.

- **Minősített név** (*Qualified Name*): részben megadott útvonal, amely nem a gyökérnévtérből indul (pl. `Models\User`). A PHP ezt az aktuális névtér és az importok alapján oldja fel.
- **Teljesen minősített név** (*Fully Qualified Name*): teljes útvonal a gyökérnévtértől, backslash karakterrel kezdve. Egyértelműen azonosítja az elemet, függetlenül az aktuális névtértől.

```php
namespace MyProject;

// Nem minősített név: az aktuális névtérben található elem
class MyClass {}

$a = new MyClass();

// Minősített név: részleges útvonal
$b = new Models\User(1, 'Kovács János');

// Teljesen minősített név: abszolút útvonal
$c = new \MyProject\Models\User(1, 'Kovács János');
```

3.57. kódpélda. *Névtérelemek elérése*

> **Megjegyzés**
>
> A PHP a névfeloldás során először a `use` importokat veszi figyelembe, majd az aktuális névteret vizsgálja, végül pedig (függvények és konstansok esetén) a globális névteret is ellenőrzi.

3.14.4. Globális névtér

A globális névtér a névtér nélküli kódot jelenti. Névteres környezetben a beépített PHP-függvények és osztályok eléréséhez backslash (\) használható, így egyértelművé válik, hogy globális függvényt vagy osztályt használunk, nem pedig egy azonos nevű, saját névtérben definiált elemet.

```php
namespace MyProject;
$array    = \array_filter([1, null, 3]);  // beépített
    függvény
$datetime = new \DateTime();              // beépített
    osztály
// Saját strlen, amely elfedi a beépített függvényt
function strlen(string $str): string
{
```

```
    return 'Egyéni strlen megvalósítás';
}
echo \strlen('Hello');  // 5 (beépített függvény)
echo strlen('Hello');   // Egyéni strlen megvalósítás
```

3.58. kódpélda. *Globális névtér elérése*

3.15. Hibakezelés és kivételkezelés

A hibakezelés és a kivételkezelés fontos szerepet játszik a megbízható PHP-alkalmazások fejlesztésében. A PHP-ben két fő megközelítést különböztetünk meg: a hagyományos hibakezelési mechanizmusokat és az objektumorientált kivételkezelést.

3.15.1. Hibatípusok PHP-ben

A PHP-ben többféle hibatípus létezik, amelyek eltérő súlyosságú problémákat jeleznek:

- **Notice**: nem kritikus, tájékoztató jellegű üzenet, amely hibára utaló kódrészletre figyelmeztet.
- **Warning**: súlyosabb hiba, amely nem szakítja meg a szkript futását, például nem inicializált változó használatakor vagy nem létező fájl megnyitásakor.
- **Error**: kritikus futásidejű hiba, amely a szkript leállásához vezethet.
- **Parse error**: szintaktikai hiba, amely megakadályozza a szkript elindulását.

3.15.2. Hibakezelő függvények

A PHP lehetőséget biztosít a hibajelentési szint (*error reporting level*) konfigurálására, valamint egyedi hibakezelő függvények regisztrálására. A beállítások jellemzően eltérnek fejlesztési és éles környezetben.

```
// Minden hiba megjelenítése (fejlesztési környezetben)
error_reporting(E_ALL);
ini_set('display_errors', '1');
// Hibák naplózása (éles környezetben)
ini_set('log_errors', '1');
ini_set('error_log', '/path/to/error.log');
```

3.59. kódpélda. *Hibajelentés beállítása*

A `set_error_handler()` függvény segítségével saját hibakezelő logika is definiálható:

```
function customErrorHandler(
    int $errno,
    string $errstr,
    string $errfile,
    int $errline
): bool
{
    switch ($errno) {
        case E_USER_ERROR:
            error_log("[$errno] $errstr in
                $errfile:$errline");
            exit(1);

        case E_USER_WARNING:
            error_log("Warning [$errno]: $errstr");
            break;

        case E_USER_NOTICE:
            error_log("Notice [$errno]: $errstr");
            break;

        default:
            error_log("Unknown error [$errno]: $errstr");
            break;
    }

    return true; // a beépített hibakezelő ne fusson le
}

set_error_handler('customErrorHandler');

trigger_error('Teszthiba', E_USER_WARNING);
```

3.60. kódpélda. *Saját hibakezelő függvény*

Fontos megjegyezni, hogy a PHP-hibák (például a `Notice` vagy a `Warning`) alapértelmezés szerint nem kivételek, hanem külön hibakezelési mechanizmus részét képezik. Emiatt közvetlenül nem kezelhetők `try-catch` blokkokkal. Szükség esetén azonban egyedi hibakezelő segítségével kivételekké alakíthatók.

3.15.3. Kivételek (Exceptions)

A kivételek objektumorientált megközelítést kínálnak a hibakezeléshez, amely strukturáltabb és rugalmasabb hibakezelést tesz lehetővé.

Kivétel dobása

Kivételt a `throw` kulcsszó segítségével dobhatunk:

```php
function divide(float $a, float $b): float
{
    if ($b == 0.0) {
        throw new DivisionByZeroError(
            "Nullával való osztás"
        );
    }

    return $a / $b;
}

function processNumber(mixed $number): float
{
    if (!is_numeric($number)) {
        throw new InvalidArgumentException(
            "Az argumentumnak számnak kell lennie"
        );
    }

    return (float)$number * 2;
}
```

3.61. kódpélda. *Kivétel dobása*

Kivétel kezelése

A kivételeket `try-catch` blokkok segítségével kezeljük:

```php
try {
    echo divide(10, 0);

} catch (DivisionByZeroError $e) {
    echo "Kivétel: "
        . $e->getMessage() . "\n";
}

try {
    echo processNumber("nem szám");
```

```
} catch (InvalidArgumentException $e) {
    echo "Érvénytelen argumentum: "
        . $e->getMessage() . "\n";
}
```

3.62. kódpélda. *Kivétel kezelése*

Több kivétel kezelése

Több `catch` ág is használható különböző kivételtípusok kezelésére. Fontos, hogy a specifikusabb kivételek mindig az általánosabbak előtt szerepeljenek.

```
try {
    echo divide(10, 0);

} catch (DivisionByZeroError $e) {
    echo "Osztási hiba: "
        . $e->getMessage() . "\n";

} catch (Exception $e) {
    echo "Általános hiba: "
        . $e->getMessage() . "\n";
}
```

3.63. kódpélda. *Több kivétel kezelése*

Több kivételtípus egyetlen `catch` blokkban is kezelhető a `|` (pipe) operátor segítségével:

```
try {
    // Kód, amely kivételt dobhat

} catch (
    InvalidArgumentException | LogicException $e
) {
    echo "Kivétel: "
        . $e->getMessage();
}
```

3.64. kódpélda. *Több kivételtípus egy catch blokkban*

Finally blokk

A `finally` blokk minden esetben lefut, függetlenül attól, hogy történt-e kivétel. Akkor is végrehajtódik, ha a `try` vagy `catch` ágban `return` utasítás szerepel.

```php
function testFinally(): bool
{
    try {
        throw new Exception("Teszt kivétel");

    } catch (Exception $e) {
        echo "Kivétel: "
            . $e->getMessage() . "\n";

        return false;

    } finally {
        echo "A finally blokk lefutott\n";
    }
}

testFinally();
```

3.65. kódpélda. *Finally blokk használata*

3.15.4. Saját kivételosztályok

Saját kivételosztály a beépített `Exception` osztályból származtatva hozható létre. A konstruktorban a `parent::__construct()` hívással továbbítható az üzenet a szülőosztálynak, miközben további tulajdonságok és metódusok is definiálhatók a specifikus hibainformációk kezelésére.

```php
class StudentGradeException extends Exception
{
    public function __construct(
        string $message,
        protected string $studentId
    ) {
        parent::__construct($message);
    }

    public function getStudentId(): string
    {
        return $this->studentId;
    }
}
```

3.66. kódpélda. *A `StudentGradeException` osztály definíciója*

Az alábbi példában a `StudentGradeException` kivétel érvénytelen jegy esetén a program ezt a kivételt dobja:

```php
class Student
{
    private ?int $grade = null;

    public function __construct(
        private string $id,
        private string $name
    ) {}

    public function setGrade(int $grade): void
    {
        if ($grade < 1 || $grade > 10) {
            throw new StudentGradeException(
                "Érvénytelen jegy. Az értéknek "
                . "1 és 10 közé kell esnie.",
                $this->id
            );
        }

        $this->grade = $grade;
    }
}

// Használat:
try {
    $student = new Student(
        "S001",
        "Kovács János"
    );

    $student->setGrade(11);

} catch (StudentGradeException $e) {
    echo "Hiba: "
        . $e->getMessage() . "\n";

    echo "Hallgató azonosító: "
        . $e->getStudentId() . "\n";
}
```

3.67. kódpélda. *A `Student` osztály kivételkezeléssel*

Ebben a példában:

- a `setGrade()` metódus kivételt dob érvénytelen érték esetén;
- a `try-catch` blokk kezeli a kivételt, majd kiolvassa a kapcsolódó adatokat.

3.16. SOLID-elvek

Az objektumorientált programozás eszközei (például az osztályok, az öröklődés vagy az interfészek) önmagukban még nem garantálják, hogy a kód jól strukturált és könnyen karbantartható lesz. Ehhez megfelelő tervezési elvekre is szükség van. Robert C. Martin (Martin 2002) a SOLID betűszóban foglalta össze az objektumorientált fejlesztés öt alapvető irányelvét.

S – Single Responsibility Principle (egyetlen felelősség elve): egy osztálynak lehetőleg egy jól meghatározott feladata legyen. Ha például egy `User` osztály egyszerre tárolja a felhasználói adatokat és végzi az adatbázis-műveleteket is, akkor több, egymástól független ok miatt módosulhat.

O – Open/Closed Principle (nyitott/zárt elv): az osztályok legyenek bővíthetők anélkül, hogy a meglévő, már működő kódot módosítani kellene. Új viselkedést gyakran interfészek vagy öröklődés segítségével adunk hozzá.

L – Liskov Substitution Principle (Liskov-féle helyettesíthetőség elve): egy leszármazott osztály objektuma minden esetben használható legyen a szülőosztály helyén anélkül, hogy a program működése megváltozna. Ez az öröklődés helyes alkalmazásának egyik alapelve.

I – Interface Segregation Principle (interfész-szétválasztás elve): érdemes több kisebb, jól körülhatárolt interfészt használni egyetlen nagy interfész helyett. Így az osztályoknak csak azokat a metódusokat kell implementálniuk, amelyekre valóban szükségük van.

D – Dependency Inversion Principle (függőség-megfordítás elve): a magas szintű komponensek ne közvetlenül konkrét osztályoktól függjenek, hanem absztrakcióktól, például interfészektől. Ez az elv a dependency injection alapját is képezi, amellyel a 9. fejezetben találkozunk részletesebben.

A SOLID-elvek nem szigorú szabályok, hanem általános iránymutatások. Elsősorban nagyobb vagy hosszabb életciklusú projektek esetén segítik a kód karbantarthatóságát, bővíthetőségét és tesztelhetőségét.

3.17. Gyakorlófeladatok

1. Készítsünk egy `Product` osztályt az alábbi elemekkel:
   - `private string $name`, `private float $price`, `private int $stock` tulajdonságok;
   - konstruktor, amely mindhárom tulajdonságot inicializálja;
   - getter és setter metódusok az árhoz és a készlethez (negatív értékek ne legyenek megengedettek);
   - `__toString(): string` metódus, amely formázottan adja vissza a termék adatait.

Példányosítsunk két különböző `Product` objektumot, módosítsuk az egyik árát, majd jelenítsük meg mindkettőt `echo` segítségével.

2. Alakítsuk át a `Product` osztályt constructor property promotion használatával, és a `$name` tulajdonságot tegyük `readonly`-vá. Egészítsük ki az osztályt az alábbi elemekkel:

   - `__clone()` metódus, amely klónozáskor a készletet nullára állítja.

   Példányosítsuk az osztályt, klónozzuk, majd az új példányon állítsunk be új árat és készletet, végül írjuk ki mindkettőt `echo`-val.

3. Definiáljunk egy `Discountable` interfészt egy `getDiscount(): float` metódussal, majd bővítsük a `Product` osztályt ennek implementálásával (alapértelmezetten 0% kedvezménnyel). Ezután készítsünk egy `ElectronicsProduct` leszármazott osztályt, amely:

   - 15%-os kedvezményt ad;
   - bővíti a szülőosztály konstruktorát;
   - a kedvezményes árat egy `getFinalPrice(): float` metódusban számítja ki.

   Példányosítsuk mindkét osztályt, és írjuk ki az eredeti, valamint a kedvezményes árakat.

4. Készítsünk egy absztrakt `Payment` osztályt az alábbi elemekkel:

   - absztrakt `pay(float $amount): bool` metódus;
   - konkrét `getDescription(): string` metódus.

   Valósítsuk meg a `CashPayment` és `CardPayment` leszármazott osztályokat saját logikával. A kártyás fizetés dobjon kivételt, ha az összeg meghaladja a rendelkezésre álló keretet. Írjunk egy `checkout(Payment $payment, float $total): void` függvényt, amely polimorf módon kezeli a különböző fizetési módokat.

5. Készítsünk két traitet:

   - `Loggable`: `log(string $message): void` metódussal, amely kiírja az osztály nevét és az üzenetet;
   - `Timestampable`: tárolja a létrehozás (`createdAt`) és az utolsó módosítás (`updatedAt`) időpontját, és minden módosításnál frissíti az `updatedAt` értékét.

   Alkalmazzuk mindkét traitet a `Product` és egy új `Order` osztályban.

6. Definiáljunk egy `ProductStatus` backed enumot az alábbi elemekkel:

   - `Active`, `Inactive`, `Discontinued` esetek sztringértékekkel;
   - `label(): string` metódus, amely magyar nyelvű megnevezést ad vissza.

   Készítsünk egy `OrderStatus` unit enumot (`Pending`, `Processing`, `Shipped`, `Delivered` esetekkel). Az `Order` osztályban használjuk az `OrderStatus` enumot, az állapotváltozásokat pedig naplózzuk a `Loggable` trait segítségével.

7. Szervezzük az eddigi osztályokat névterekbe:

   - `App\Models`: `Product`, `Order`;
   - `App\Payment`: `Payment`, `CashPayment`, `CardPayment`.

   Készítsünk egy `index.php` belépési pontot, amely importálja és példányosítja az osztályokat. Az osztályok betöltéséhez használjunk `spl_autoload_register()` alapú autoloadert.

8. Írjunk egy `ProductNotFoundException` és egy `OutOfStockException` kivételosztályt, amelyek a beépített `RuntimeException` osztályból öröklődnek. Alkalmazzuk ezeket a `Product` osztályban:

   - ha egy termék nem található, dobjon `ProductNotFoundException` kivételt;
   - ha a készlet nem elegendő, dobjon `OutOfStockException` kivételt.

   Egészítsük ki a `checkout()` függvényt `try/catch/finally` blokkokkal.

4. FEJEZET

ŰRLAPOK KEZELÉSE

4.1. Űrlapok alapjai

Az űrlapok a webalkalmazások alapvető elemei, amelyek lehetővé teszik az adatok továbbítását a felhasználó és a szerver között. Segítségükkel a felhasználók adatokat küldhetnek a szervernek feldolgozás céljából.

4.1.1. Űrlap szintaxisa

A HTML-űrlapok alapvető szintaxisa a következő:

```
<form action="..." method="...">
    <!-- Űrlap mezők -->
</form>
```

4.1. kódpélda. *Alap HTML-űrlap-struktúra*

Az űrlap legfontosabb attribútumai:

- `action`: megadja azt a szerveroldali szkriptet (URL-t), amely feldolgozza az űrlap adatait;
- `method`: az adatküldés módja. Két fő értéke lehet:
  - `GET`: a böngésző az adatokat az URL-ben küldi el;
  - `POST`: a böngésző az adatokat a HTTP-kérés törzsében küldi el.

```
<form action="process.php" method="POST">
    <input type="text" name="name">
    <input type="submit" value="Küldés">
</form>
```

4.2. kódpélda. *Űrlap küldése*

4.1.2. Űrlapelemek

Az űrlapok különböző beviteli mezőket tartalmazhatnak. A legfontosabb űrlapelemek:

- `<input>`: különböző típusú beviteli mezők
  - `type="text"`: szöveges beviteli mező.
  - `type="password"`: jelszómező.
  - `type="email"`: e-mail-cím bevitelére optimalizált mező.
  - `type="number"`: numerikus beviteli mező léptetőnyilakkal.
  - `type="url"`: URL bevitelére optimalizált mező.
  - `type="tel"`: telefonszám bevitelére optimalizált mező.
  - `type="date"`: dátumválasztó mező.
  - `type="range"`: csúszka értéktartományon belüli érték kiválasztására.
  - `type="color"`: színválasztó mező.
  - `type="radio"`: rádiógomb.
  - `type="checkbox"`: jelölőnégyzet.
  - `type="file"`: fájlfeltöltő mező.
  - `type="hidden"`: rejtett mező.
  - `type="submit"`: küldés gomb.
- `<textarea>`: többsoros szövegbeviteli mező;
- `<select>`: legördülő lista;
- `<button>`: gomb;
- `<label>`: címke az űrlapelemekhez;
- `<fieldset>` és `<legend>`: űrlapelemek csoportosítása.

Az alábbi összetettebb regisztrációs űrlap több különböző űrlapelem használatát mutatja be:

```
<form action="process.php" method="POST">
    <label for="name">Név:</label>
    <input type="text" id="name" name="name" required>

    <label for="email">Email:</label>
    <input type="email" id="email" name="email" required>
```

```
    <label for="password">Jelszó:</label>
    <input type="password" id="password" name="password"
        required>

    <label for="gender">Nem:</label>
    <select id="gender" name="gender">
        <option value="male">Férfi</option>
        <option value="female">Nő</option>
    </select>

    <label>Érdeklődési körök:</label>
    <input type="checkbox" id="sports" name="interests[]"
        value="sports">
    <label for="sports">Sport</label>

    <input type="checkbox" id="music" name="interests[]"
        value="music">
    <label for="music">Zene</label>

    <label for="comments">Megjegyzések:</label>
    <textarea id="comments" name="comments"></textarea>

    <input type="submit" value="Küldés">
</form>
```

4.3. kódpélda. *Különböző űrlapelemek*

4.1.3. Űrlapelemek attribútumai

Az űrlapelemek számos attribútummal rendelkezhetnek a működés és a validáció szabályozására:

- `required`: kötelező mező;
- `placeholder`: helykitöltő szöveg, amely eltűnik, amikor a felhasználó elkezd gépelni;
- `maxlength`: maximális karakterszám szöveges beviteli mezők esetén;
- `min` és `max`: szám típusú mezők értéktartományának megadása;
- `pattern`: reguláris kifejezés a bevitel ellenőrzésére;
- `autocomplete`: az automatikus kitöltés viselkedését szabályozza;
- `disabled`: letiltja a mezőt, így a felhasználó nem tudja módosítani;

- `readonly`: a mező csak olvasható, de az űrlappal együtt elküldhető.

Néhány attribútum együttes használata:

```
<input type="text" name="username" required
    placeholder="Felhasználónév">
<input type="number" name="age" min="18" max="100">
<input type="tel" name="phone"
    pattern="[0-9]{3}-[0-9]{3}-[0-9]{4}">
<input type="email" name="email" autocomplete="off">
<input type="text" name="code" maxlength="6" readonly
    value="ABC123">
```

4.4. kódpélda. *Űrlapelemek attribútumai*

4.2. GET- és POST-metódusok

A GET és a POST a két leggyakrabban használt HTTP-metódus az űrlapadatok továbbítására. Mindkettő eltérő működéssel és eltérő felhasználási területtel rendelkezik.

4.2.1. GET-metódus

A GET-metódus az űrlap adatait az URL-ben, név–érték párok formájában küldi el:

```
<form method="GET">
    <input type="text" name="username" />
    <input type="submit" value="Keresés" />
</form>
```

4.5. kódpélda. *A GET-metódus használata*

Ha a felhasználó beírja a „KovacsJanos” felhasználónevet, majd elküldi az űrlapot, az eredményül kapott URL a következő lesz:

```
http://example.com/search.php?username=KovacsJanos
```

A GET-metódus jellemzői:

- az adatok láthatók az URL-ben;
- korlátozott mennyiségű adat küldhető (a pontos korlát böngésző- és szerverfüggő);
- könyvjelzőzhető és megosztható URL-ek hozhatók létre;

- nem alkalmas érzékeny adatok továbbítására;
- idempotens: azonos kérés többszöri végrehajtása ugyanazt a hatást váltja ki a szerveren, mint egyetlen végrehajtása.

Használata ajánlott:

- keresési lekérdezéseknél;
- állapotmentes, kizárólag adatlekérdezést végző műveleteknél.

Megjegyzés

Jelszavakat és egyéb érzékeny adatokat soha ne küldjünk GET-metódussal, mert azok az URL-ben láthatóvá válnak, illetve megjelenhetnek a böngésző előzményeiben és a szervernaplókban is.

4.2.2. POST-metódus

A POST-metódus az űrlap adatait a HTTP-kérés törzsében küldi el:

```
<form action="process.php" method="POST">
    <input type="text" name="username" />
    <input type="password" name="password" />
    <input type="submit" value="Bejelentkezés" />
</form>
```

4.6. kódpélda. *A POST-metódus használata*

A POST-metódus jellemzői:

- az adatok nem láthatók az URL-ben;
- nagyobb mennyiségű adat küldhető;
- nem könyvjelzőzhető;
- érzékeny adatok esetén biztonságosabb (de nem titkosított!);
- nem idempotens: azonos kérés többszöri elküldése eltérő eredményeket is okozhat.

Használata ajánlott:

- bejelentkezési űrlapoknál;
- adatok létrehozásánál vagy módosításánál;
- fájlfeltöltésnél.

4.2.3. A GET és POST összehasonlítása

A két metódus közötti legfontosabb különbségeket a 4.1. táblázat foglalja össze.

4.1. táblázat. *A GET- és POST-metódusok összehasonlítása*

Jellemző	**GET**	**POST**
Adatok helye	URL-ben („query string”)	HTTP-kérés törzsében
Adatmennyiség	Korlátozott	Gyakorlatilag korlátlan
Biztonság	Kevésbé biztonságos (látható)	Biztonságosabb (de nem titkosított)
Könyvjelzőzhető	Igen	Nem
Cache-elhető	Igen	Nem
Idempotens	Igen	Nem
Böngésző előzmények	Az adatokkal együtt megjelenik	Csak az URL jelenik meg

A táblázat három fogalmát érdemes külön is tisztázni:

- **Könyvjelzőzhető**: mivel a GET-kérés paraméterei az URL-ben szerepelnek, a teljes cím elmenthető könyvjelzőként és később újra felhasználható, például `termekek.php?kategoria=laptop&oldal=2`. POST esetén ez nem lehetséges, mivel a paraméterek nem az URL-ben szerepelnek.
- **Gyorsítótárazható (cache-elhető)**: a böngésző és a köztes proxyk eltárolhatják a GET-kérések válaszát, így ismételt kérés esetén a tárolt eredmény szolgálható ki. Ez gyorsabb válaszidőt eredményez. A POST-kérések válaszait a böngészők általában nem gyorsítótárazzák, mivel feltételezhetően módosítják a szerver állapotát.
- **Idempotens**: egy kérés idempotens, ha többszöri végrehajtása ugyanazt a hatást váltja ki a szerveren, mint egyetlen végrehajtása. A GET-kérések nem módosítják a szerver állapotát, ezért idempotensek. A POST ezzel szemben általában műveleteket hajt végre (például adat rögzítését), így ismételt elküldése duplikációt okozhat. Ezért gyakran alkalmazzák a PRG (*Post/Redirect/Get*) mintát.

4.2.4. Szuperglobális változók és adatelérés

A PHP a szuperglobális változók segítségével biztosít hozzáférést az űrlapadatokhoz. Ezek beépített tömbök, amelyek a program bármely hatóköréből elérhetők.

- `$_GET`: a GET-metódussal küldött adatokat tartalmazza.
- `$_POST`: a POST-metódussal küldött adatokat tartalmazza.
- `$_REQUEST`: a `$_GET` és a `$_POST` (a beállítástól függően a `$_COOKIE`) adatait együttesen tartalmazza.
- `$_COOKIE`: a böngésző által küldött sütiket tartalmazza (lásd az 5.2. alfejezetet).
- `$_SESSION`: a munkamenet adatait tartalmazza (lásd az 5.3. alfejezetet).
- `$_FILES`: a feltöltött fájlok adatait tartalmazza.
- `$_SERVER`: szerver- és környezeti információkat tartalmaz.
- `$_ENV`: a szerver operációs rendszerének környezeti változóit tartalmazza.

Megjegyzés

A `$_REQUEST` tartalmát és a források sorrendjét a `php.ini` fájl `request_order` beállítása határozza meg (ha ez üres, a `variables_order`). A PHP-vel szállított `php.ini` fájlokban ennek értéke `"GP"`, ezért a `$_REQUEST` alapértelmezetten csak a GET- és POST-adatokat tartalmazza. Névütközés esetén a sorrendben később álló forrás értéke felülírja a korábbit: például egy azonos nevű POST-mező felülírja a GET-paramétert. Így nem mindig egyértelmű, honnan származik egy érték. Ezért éles alkalmazásban a `$_GET` és `$_POST` tömbök közvetlen használata javasolt a `$_REQUEST` helyett.

```php
// GET adat elérése
$username = $_GET['username'] ?? '';

// POST adat elérése
$password = $_POST['password'] ?? '';

// Mindkét metódus adatainak elérése
$email = $_REQUEST['email'] ?? '';
```

4.7. kódpélda. *GET- és POST-adatok elérése PHP-ben*

4.2.5. Űrlapok feldolgozási módjai

Az űrlapok feldolgozására több megközelítés is létezik PHP-ben.

Külön feldolgozó fájl

Ebben az esetben az űrlap adatait egy külön PHP-fájlba küldjük feldolgozásra:

```
<form action="process.php" method="POST">
    <input type="text" name="username" />
    <input type="submit" value="Küldés" />
</form>
```

4.8. kódpélda. *Űrlap külön feldolgozó fájllal*

A `process.php` fájl tartalma:

```
if ($_SERVER['REQUEST_METHOD'] === 'POST') {
    $username = $_POST['username'] ?? '';
    // Feldolgozás...
}
```

4.9. kódpélda. *Külön feldolgozó fájl*

Ez a megközelítés elősegíti a kód tiszta szervezését és a logika elkülönítését a megjelenítéstől. Ugyanakkor több fájl kezelése szükséges, ami nagyobb projektek esetén összetettebb projektstruktúrához vezethet.

Feldolgozás ugyanabban a fájlban

Ebben az esetben az űrlap és a feldolgozó kód ugyanabban a fájlban található:

```
<?php
if ($_SERVER['REQUEST_METHOD'] === 'POST') {
    $username = $_POST['username'] ?? '';
    // Feldolgozás...
}
?>
<form method="POST">
    <input type="text" name="username">
    <input type="submit" value="Küldés">
</form>
```

4.10. kódpélda. *Feldolgozás ugyanabban a fájlban*

Ez a módszer egyszerűbb fájlstruktúrát eredményez, és kisebb projektek esetén gyorsabb fejlesztést tesz lehetővé. Nagyobb alkalmazásoknál azonban a

kód könnyen összetettebbé válhat, és nehezebb lehet a logika és a megjelenítés szétválasztása.

Az üres `action` attribútum használata biztosítja, hogy az űrlap ugyanarra az oldalra kerüljön feldolgozásra. Ez általában biztonságosabb megoldás, mint a `$_SERVER['PHP_SELF']` használata, mivel utóbbi XSS-támadásokra lehet sérülékeny (lásd a 4.6.1. alfejezetet).

A PHP- és HTML-kód keveredése

A PHP- és HTML-kód keveredése lehetővé teszi a dinamikus tartalom közvetlen beágyazását a HTML-struktúrába. Ilyenkor a vezérlési szerkezeteknek gyakran a kettőspontos írásmódját használjuk: a nyitó kapcsos zárójel helyére kettőspont kerül, a szerkezetet pedig a nevéből képzett záró kulcsszó zárja le (`endif;`, `endforeach;`, `endwhile;`). A működés azonos a kapcsos zárójeles alakéval, de a HTML-részek között jobban látszik, melyik szerkezet hol ér véget:

```
<?php if ($_SERVER['REQUEST_METHOD'] === 'GET') : ?>
    <!-- Űrlap megjelenítése -->
<?php else : ?>
    <!-- Feldolgozás eredménye -->
<?php endif; ?>
```

4.11. kódpélda. *A PHP- és HTML-kód keveredése*

Ez a megközelítés lehetővé teszi a feltételes tartalommegjelenítést, ugyanakkor nagyobb projektekben megnehezítheti a kód karbantartását és olvashatóságát.

4.2.6. Post/Redirect/Get (PRG) minta

A Post/Redirect/Get (PRG) egy fontos tervezési minta webalkalmazásokban, amely megakadályozza az űrlapok többszöri beküldését akkor, amikor a felhasználó frissíti az oldalt (F5), vagy a böngésző Vissza gombjával navigál.

A probléma

Amikor egy POST-kérés után közvetlenül HTML-választ küldünk vissza, és a felhasználó frissíti az oldalt, a böngésző megkérdezi, hogy újra el akarja-e küldeni az űrlapot. Ilyenkor az adatok ismét feldolgozásra kerülhetnek (például ugyanazon rendelés vagy regisztráció többszöri rögzítése).

```
<?php
// HELYTELEN - dupla beküldés lehetséges!
if ($_SERVER['REQUEST_METHOD'] === 'POST') {
    $name = $_POST['name'] ?? '';
    // Adatok mentése adatbázisba...
```

```
    echo "Sikeres mentés!"; // F5 esetén újra lefut!
}
?>
<form method="POST">
    <input type="text" name="name">
    <input type="submit" value="Küldés">
</form>
```

4.12. kódpélda. *Helytelen megközelítés (dupla beküldés lehetséges)*

A megoldás: PRG-minta

A PRG-minta lényege, hogy sikeres POST-kérés feldolgozása után egy `header("Location: ...")` átirányítást alkalmazunk. Ennek hatására a böngésző egy új GET-kérést indít, amely már biztonságosan frissíthető.

```
<?php
session_start();
if ($_SERVER['REQUEST_METHOD'] === 'POST') {
    // Validáció
    $name = trim($_POST['name'] ?? '');
    if (empty($name)) {
        // Hiba esetén az adatokat és a hibaüzenetet
            session-ben tároljuk
        $_SESSION['error'] = "A név megadása kötelező!";
        $_SESSION['form_data'] = $_POST;
        header("Location: prg.php");
        exit;
    }
    // Sikeres feldolgozás (pl. adatbázisba mentés)
    // ...
    // Sikerüzenet session-ben
    $_SESSION['success'] = "Sikeres mentés!";
    // Átirányítás GET kérésre (PRG)
    header("Location: prg.php");
    exit; // A Location fejléc után állítsuk le a szkriptet
}
// Üzenetek és korábbi adatok kiolvasása, majd törlése
$error = $_SESSION['error'] ?? null;
$success = $_SESSION['success'] ?? null;
$formData = $_SESSION['form_data'] ?? [];
unset($_SESSION['error'], $_SESSION['success'],
    $_SESSION['form_data']);
?>
<!DOCTYPE html>
```

```
<html>
<head><title>PRG példa</title></head>
<body>
    <?php if ($success): ?>
        <p style="color: green;"><?=
            htmlspecialchars($success) ?></p>
    <?php endif; ?>
    <?php if ($error): ?>
        <p style="color: red;"><?= htmlspecialchars($error)
            ?></p>
    <?php endif; ?>
    <form method="POST">
        <label>Név:</label>
        <input type="text" name="name"
               value="<?= htmlspecialchars($formData['name']
                   ?? '') ?>">
        <input type="submit" value="Küldés">
    </form>
</body>
</html>
```

4.13. kódpélda. *PRG-minta implementálása (prg.php)*

A PRG-minta előnyei:

- megakadályozza az űrlapok többszöri beküldését oldalfrissítéskor;
- a böngésző vissza gombja is biztonságosan használható;
- a könyvjelzőzés nem okoz problémát;
- tisztább URL-eket eredményez.

Megjegyzés

A `Location` fejléc elküldése után ajánlott az `exit` utasítással leállítani a szkriptet. Az átirányítás enélkül is megtörténik, de a szkript tovább fut, és a hátralévő kód (például egy adatbázis-művelet) feleslegesen vagy nem várt módon végrehajtódhat. Más fejlécek (`Content-Type`, `Set-Cookie` stb.) után természetesen nincs szükség kilépésre. A példában használt `session_start()`, `$_SESSION` és `header()` elemekkel részletesebben a következő fejezet foglalkozik.

4.3. Űrlapmezők feldolgozása PHP-ben

Az űrlapok különböző típusú mezőket tartalmazhatnak, amelyeket PHP-ben megfelelően kell feldolgozni. Ebben a részben a leggyakoribb űrlapmezők kezelését tekintjük át.

Az űrlapfeldolgozás általános menete a következő lépésekből áll:

1. **Kérés ellenőrzése**: megbizonyosodunk arról, hogy POST- vagy GET-típusú kérés érkezett-e (`$_SERVER['REQUEST_METHOD']`).
2. **Adatok beolvasása**: az adatok kiolvasása a szuperglobális tömbökből (pl. `$_POST`, `$_GET`) vagy a `filter_input()` függvény segítségével.
3. **Validáció**: ellenőrizzük, hogy az adatok érvényesek és elfogadhatók-e.
4. **Feldolgozás**: az érvényes adatok mentése, továbbítása vagy egyéb műveletek végrehajtása.
5. **Kimeneti adatok kódolása** (output encoding): a felhasználótól származó adatot HTML-kimenetbe illesztés előtt megfelelően kódolni kell. Erre a `htmlspecialchars()` függvény használható, amely a HTML speciális karaktereit HTML-entitásokká alakítja (például a `<` jelet `<`-re).

4.3.1. Szöveges mező

A szöveges mezők (`<input type="text">`, `<textarea>`) értékei egyszerűen kiolvashatók a `$_POST` vagy `$_GET` tömbökből:

```
<form method="POST">
    <input type="text" name="username" />
    <textarea name="comment"></textarea>
    <input type="submit" value="Küldés" />
</form>

<?php
if ($_SERVER['REQUEST_METHOD'] === 'POST') {
    $username = $_POST['username'] ?? '';
    $comment  = $_POST['comment'] ?? '';

    echo "Felhasználónév: " . htmlspecialchars($username) .
        "<br>";
    echo "Megjegyzés: " . htmlspecialchars($comment);
}
?>
```

4.14. kódpélda. *Szöveges mező feldolgozása*

A `$_POST` tömbből kiolvasott értékeket a `htmlspecialchars()` függvénnyel kódoljuk, mielőtt megjelenítjük őket, hogy megelőzzük az XSS-támadásokat.

4.3.2. Rádiógomb

A rádiógombok (`<input type="radio">`) feldolgozásakor a böngésző csak a kiválasztott értéket küldi el. Fontos kezelni azt az esetet is, amikor egyik opció sincs kiválasztva. Emellett minden esetben ellenőrizni kell, hogy a beérkezett érték a megengedett értékek halmazába (*whitelist*) tartozik-e.

```
<form method="POST">
    <input type="radio" name="gender" value="male"> Férfi
    <input type="radio" name="gender" value="female"> Nő
    <input type="submit" value="Küldés">
</form>

<?php
if ($_SERVER['REQUEST_METHOD'] === 'POST') {
    $allowed = ['male', 'female'];
    $gender  = $_POST['gender'] ?? '';

    if (in_array($gender, $allowed, true)) {
        echo "Választott nem: " . htmlspecialchars($gender);
    } else {
        echo "Érvénytelen vagy hiányzó választás.";
    }
}
?>
```

4.15. kódpélda. *Rádiógomb feldolgozása engedélyezett értékek ellenőrzésével*

4.3.3. Jelölőnégyzet

A jelölőnégyzetek (`<input type="checkbox">`) feldolgozásakor különbséget kell tenni az egyetlen és a több kiválasztott érték kezelésében, mivel több jelölőnégyzet esetén az adatok tömbként érkeznek a szerverre.

Egyetlen jelölőnégyzet

Egyetlen jelölőnégyzet esetén ellenőrizni kell, hogy be van-e jelölve:

```
<form method="POST">
    <input type="checkbox" name="agree" value="1" />
        Elfogadom a feltételeket
    <input type="submit" value="Küldés" />
</form>
```

```php
<?php
if ($_SERVER['REQUEST_METHOD'] === 'POST') {
    if (isset($_POST['agree']) && $_POST['agree'] === '1') {
        echo "Elfogadta a feltételeket";
    } else {
        echo "Nem fogadta el a feltételeket";
    }
}
?>
```

4.16. kódpélda. *Egyetlen jelölőnégyzet feldolgozása*

Ha a jelölőnégyzet nincs bejelölve, a böngésző nem küldi el a mezőt, ezért az `isset()` ellenőrzése elengedhetetlen.

Többválasztásos jelölőnégyzetek

Többválasztásos jelölőnégyzetek esetén tömböt használunk a kiválasztott értékek tárolására, mivel a felhasználó egyszerre több opciót is kiválaszthat.

```php
<form method="POST">
    <input type="checkbox" name="fruits[]" value="apple" />
        Alma
    <input type="checkbox" name="fruits[]" value="banana" />
        Banán
    <input type="checkbox" name="fruits[]" value="orange" />
        Narancs
    <input type="submit" value="Küldés" />
</form>

<?php
if ($_SERVER['REQUEST_METHOD'] === 'POST') {
    $fruits = $_POST['fruits'] ?? [];

    if (!empty($fruits)) {
        $safeFruits = array_map('htmlspecialchars', $fruits);
        echo "Választott gyümölcsök: " . implode(", ",
            $safeFruits);
    } else {
        echo "Nem választott gyümölcsöt.";
    }
}
?>
```

4.17. kódpélda. *Többválasztásos jelölőnégyzet feldolgozása*

A `name="fruits[]"` szintaxis hatására a PHP a kiválasztott értékeket tömbként adja át. Ha egyetlen elem sincs bejelölve, a `$_POST['fruits']` kulcs nem létezik, ezért a null coalescing operátorral (`??`) üres tömböt adunk alapértékként.

4.3.4. Legördülő lista

A legördülő lista (`<select>`) feldolgozása hasonló a rádiógombokéhoz, azonban a `multiple` attribútum használatával lehetőség van több elem egyidejű kiválasztására is.

```
<form method="POST">
    <select name="country">
        <option value="usa">USA</option>
        <option value="uk">UK</option>
        <option value="canada">Kanada</option>
    </select>

    <select name="languages[]" multiple>
        <option value="en">Angol</option>
        <option value="fr">Francia</option>
        <option value="de">Német</option>
    </select>

    <input type="submit" value="Küldés" />
</form>

<?php
if ($_SERVER['REQUEST_METHOD'] === 'POST') {
    $country   = htmlspecialchars($_POST['country'] ?? '');
    $languages = $_POST['languages'] ?? [];

    $safeLanguages = array_map('htmlspecialchars',
        $languages);

    echo "Választott ország: $country<br>";
    echo "Beszélt nyelvek: " . implode(", ", $safeLanguages);
}
?>
```

4.18. kódpélda. *Legördülő lista feldolgozása*

Ha a listából csak egy elem választható, egyetlen érték érkezik. Ha azonban a `multiple` attribútum is szerepel, a jelölőnégyzetekhez hasonlóan a kiválasztott értékek tömbként érkeznek, ezért a mező nevénél a `name="languages[]"` szintaxist kell alkalmazni.

4.3.5. Rejtett mező

A rejtett mezők (`<input type="hidden">`) nem jelennek meg a felhasználó számára, de a böngésző az értéküket ugyanúgy elküldi a szervernek, mint a többi űrlapmezőét. Elsősorban állapotmegőrzésre és olyan adatok továbbítására használjuk őket, amelyeket nem szeretnénk megjeleníteni a felhasználónak.

Bár a rejtett mezők nem láthatók a felhasználó számára, tartalmuk nem védett: a böngésző fejlesztői eszközeivel megtekinthetők és módosíthatók. Emiatt érzékeny adatok tárolására nem alkalmasak, és minden esetben szükséges a szerveroldali validáció.

```
<form method="POST">
    <input type="hidden" name="user_id" value="12345" />
    <input type="text" name="comment" placeholder="Írjon
        kommentet" />
    <input type="submit" value="Küldés" />
</form>

<?php
if ($_SERVER['REQUEST_METHOD'] === 'POST') {
    $userId  = htmlspecialchars($_POST['user_id'] ?? '');
    $comment = htmlspecialchars($_POST['comment'] ?? '');

    echo "Felhasználó azonosító: $userId<br>";
    echo "Komment: $comment";
}
?>
```

4.19. kódpélda. *Rejtett mező feldolgozása*

4.3.6. Összetett űrlapok feldolgozása

Összetett űrlapok esetén az adatokat strukturált formában kezeljük. Az összetartozó mezőket csoportosíthatjuk, így a PHP hierarchikus tömbstruktúrában dolgozza fel az adatokat.

```
<form method="POST">
    <input type="text" name="user[name]" />
    <input type="email" name="user[email]" />
    <input type="checkbox" name="interests[]" value="sport"
        />
    <input type="checkbox" name="interests[]" value="music"
        />
    <input type="submit" value="Küldés" />
</form>
```

```php
<?php
if ($_SERVER['REQUEST_METHOD'] === 'POST') {
    $user      = $_POST['user'] ?? [];
    $interests = $_POST['interests'] ?? [];

    echo "Név: " . htmlspecialchars($user['name'] ?? '') .
        "<br>";
    echo "Email: " . htmlspecialchars($user['email'] ?? '')
        . "<br>";

    $safeInterests = array_map('htmlspecialchars',
        $interests);
    echo "Érdeklődési körök: " . implode(", ",
        $safeInterests);
}
?>
```

4.20. kódpélda. *Összetett űrlap feldolgozása*

A `name="user[name]"` szintaxis hatására a PHP többdimenziós tömböt hoz létre, amelyben az összetartozó mezők egy kulcs alatt érhetők el (`$_POST['user']['name']`).

4.3.7. Állapotmegőrzés

Hiba esetén érdemes a korábban beküldött értékeket visszatölteni az űrlapmezőkbe, hogy a felhasználónak ne kelljen azokat újra megadnia. A beküldött adatokat egy tömbben tároljuk, majd a HTML-űrlap mezőit ezekkel az értékekkel előre kitöltjük.

```php
$formData = ['username' => '', 'interests' => [], 'bio' =>
    ''];
$errors   = [];

if ($_SERVER['REQUEST_METHOD'] === 'POST') {
    $formData['username']  = trim($_POST['username'] ?? '');
    $formData['interests'] = $_POST['interests'] ?? [];
    $formData['bio']       = trim($_POST['bio'] ?? '');

    if (empty($formData['username'])) {
        $errors['username'] = 'A felhasználónév megadása
            kötelező.';
    }
}
```

4.21. kódpélda. *Állapotmegőrzés*

```
<form method="POST">
    <!-- Szöveges mező: value attribútumban -->
    <input type="text" name="username"
           value="<?=
               htmlspecialchars($formData['username']) ?>">

    <!-- Jelölőnégyzet: checked attribútum feltételesen -->
    <input type="checkbox" name="interests[]" value="sport"
           <?= in_array('sport', $formData['interests'],
               true) ? 'checked' : '' ?>> Sport

    <!-- Textarea: tartalom a tagek között -->
    <textarea name="bio"><?=
        htmlspecialchars($formData['bio']) ?></textarea>

    <input type="submit" value="Küldés">
</form>
```

4.22. kódpélda. *Állapotmegőrzés (HTML rész)*

Megjegyzés

- Érdemes minden kimenetnél a `htmlspecialchars()` függvényt használni, mivel ez véd az XSS-támadások ellen.
- A `textarea` esetén a tartalom a nyitó és záró címke közé kerül, nem pedig a `value` attribútumba.
- Rádiógomboknál és jelölőnégyzeteknél a `checked`, legördülő listáknál pedig a `selected` attribútumot kell feltételesen alkalmazni.
- Jelszómezők esetén biztonsági okokból az értéket nem szokás visszatölteni.

4.4. Kliensoldali és szerveroldali validáció

Az űrlapok validációja két szinten valósulhat meg: kliensoldalon (a böngészőben) és szerveroldalon (PHP-ben). A két szint eltérő célokat szolgál, de együttes alkalmazásuk biztosítja a megfelelő működést és a biztonságot.

4.4.1. Kliensoldali validáció

A kliensoldali validáció a felhasználó böngészőjében történik HTML5-attribútumok vagy JavaScript segítségével. Előnye, hogy azonnali visszajelzést ad a felhasználónak, valamint csökkenti a szerver terhelését.

```
<form action="process.php" method="POST">
    <input type="email" name="email" required>
    <input type="text" name="phone"
        pattern="[0-9]{2}-[0-9]{3}-[0-9]{4}">
    <input type="number" name="age" min="18" max="99">
    <input type="submit" value="Küldés">
</form>
```

4.23. kódpélda. *HTML5 kliensoldali validáció*

JavaScript segítségével részletesebb visszajelzés is adható a felhasználónak, például hibaüzenetek megjelenítésével még az űrlap beküldése előtt:

```
document.querySelector('form').addEventListener('submit',
    function (e) {
    const email = document.querySelector('[name="email"]').
        value.trim();

    if (email === '') {
        e.preventDefault();
        alert('Az e-mail cím megadása kötelező!');
    }
});
```

4.24. kódpélda. *JavaScript kliensoldali validáció*

Megjegyzés

A JavaScript részletes tárgyalása meghaladja e fejezet kereteit. A kliensoldali validáció ráadásul könnyen megkerülhető: a felhasználó kikapcsolhatja a JavaScriptet, vagy manipulált HTTP-kéréseket küldhet a szerver felé. Ezért nem helyettesítheti a szerveroldali validációt. Az ajánlott megközelítés a két szint együttes alkalmazása.

4.4.2. Szerveroldali validáció

A szerveroldali validáció PHP-ben történik, és az adatok ellenőrzésének legmegbízhatóbb szintje. Minden felhasználói bemenetet potenciálisan veszélyesnek kell tekinteni, ezért az adatokat minden esetben szerveroldalon is ellenőrizni kell.

```
$errors = [];

if ($_SERVER['REQUEST_METHOD'] === 'POST') {
    // Email validáció
```

```
    $email = trim($_POST['email'] ?? '');
    if (empty($email)) {
        $errors['email'] = "Az email cím megadása kötelező.";
    } elseif (!filter_var($email, FILTER_VALIDATE_EMAIL)) {
        $errors['email'] = "Érvénytelen email cím formátum.";
    }

    // Életkor validáció
    $age = filter_var($_POST['age'] ?? '',
        FILTER_VALIDATE_INT);
    if ($age === false || $age < 18 || $age > 99) {
        $errors['age'] = "Az életkornak 18 és 99 között kell
            lennie.";
    }

    if (empty($errors)) {
        // Sikeres feldolgozás...
    }
}
```

4.25. kódpélda. *Szerveroldali validáció*

A hibákat egy `$errors` tömbbe gyűjtjük, így egyszerre több validációs hiba is kezelhető. Az `empty($errors)` ellenőrzés biztosítja, hogy a feldolgozás csak akkor folytatódjon, ha minden adat érvényes.

4.5. Adatok ellenőrzése és szűrése

Az űrlapadatok feldolgozásánál különbséget kell tenni a *bemeneti validáció* és a *kimeneti kódolás* között. A validáció az adatok helyességét ellenőrzi, míg a kódolás a biztonságos megjelenítést biztosítja. A két eljárás nem helyettesíti egymást, ezért mindkettő alkalmazása szükséges.

4.5.1. `htmlspecialchars()`

A `htmlspecialchars()` függvény a PHP egyik alapvető biztonsági eszköze. Feladata, hogy segítse az XSS (Cross-Site Scripting) támadások megelőzését azáltal, hogy a speciális HTML-karaktereket HTML-entitásokká alakítja.

Szintaxis

```
htmlspecialchars(
    string $string,
    int $flags = ENT_QUOTES | ENT_SUBSTITUTE,
    ?string $encoding = 'UTF-8',
    bool $double_encode = true
): string
```

Paraméterek:

- `$string`: az átalakítandó szöveg;
- `$flags` (opcionális): az idézőjelek és az érvénytelen karakterek kezelési módja. Ajánlott érték: `ENT_QUOTES | ENT_SUBSTITUTE`;
- `$encoding` (opcionális): a karakterkódolás. Ajánlott az `'UTF-8'` explicit megadása;
- `$double_encode` (opcionális): ha értéke `false`, a már kódolt entitásokat nem kódolja újra. Alapértelmezett értéke `true`.

Visszatérési érték: HTML-biztonságos, entitásokká alakított szöveg.

4.2. táblázat. *A `htmlspecialchars()` függvény karakterkonverziói*

Karakter	Helyettesítés
& (ampersand)	`&`
" (idézőjel)	`"` (kivéve, ha `ENT_NOQUOTES`)
' (aposztróf)	`'`
< (kisebb mint)	`<`
> (nagyobb mint)	`>`

```php
$userInput = '<script>alert("XSS");</script>';
$safeInput = htmlspecialchars($userInput, ENT_QUOTES,
    'UTF-8');

echo $safeInput;
// Kimenet:
// &lt;script&gt;alert(&quot;XSS&quot;);&lt;/script&gt;
```

4.26. kódpélda. *`htmlspecialchars()` használata*

4.5.2. `filter_input()` és `filter_input_array()`

A `filter_input()` függvény a HTTP-kérés bemeneti adatait (GET, POST, COOKIE stb.) olvassa, és egyetlen lépésben elvégzi azok validálását vagy szűrését. Ezáltal egyesíti az adatok beolvasását és ellenőrzését, csökkentve a hibalehetőségeket a közvetlen szuperglobális tömbök használatához képest.

Szintaxis

```
filter_input(
    int $type,
    string $var_name,
    int $filter = FILTER_DEFAULT,
    array|int $options = 0
): mixed
```

Paraméterek:

- `$type`: a bemeneti forrás típusa (`INPUT_GET`, `INPUT_POST`, `INPUT_COOKIE`, `INPUT_SERVER`);
- `$var_name`: a lekérdezett változó neve;
- `$filter` (opcionális): az alkalmazandó szűrő konstansa. Alapértelmezett értéke: `FILTER_DEFAULT`;
- `$options` (opcionális): a szűrő beállításai (például `min_range`, `max_range`).

Visszatérési érték: a szűrt érték, `false`, ha a validáció sikertelen, `null`, ha a változó nem létezik.

```php
// Email cím olvasása és validálása POST-ból
$email = filter_input(INPUT_POST, 'email',
    FILTER_VALIDATE_EMAIL);

if ($email === false) {
    echo "Érvénytelen email cím formátum.";
} elseif ($email === null) {
    echo "Az email mező nem lett elküldve.";
} else {
    echo "Érvényes email: " . htmlspecialchars($email);
}

// Egész szám olvasása és validálása GET-ből
$age = filter_input(INPUT_GET, 'age', FILTER_VALIDATE_INT, [
    'options' => ['min_range' => 18, 'max_range' => 99]
]);
```

4.27. kódpélda. *`filter_input()` használata*

A `filter_input_array()` függvény lehetővé teszi több bemeneti mező egyidejű szűrését és validálását egy konfigurációs tömb alapján.

```php
$filters = [
    'username' => FILTER_DEFAULT,
    'email' => FILTER_VALIDATE_EMAIL,
    'age' => [
        'filter' => FILTER_VALIDATE_INT,
        'options' => ['min_range' => 18, 'max_range' => 99]
    ],
    'website' => FILTER_VALIDATE_URL
];

$inputs = filter_input_array(INPUT_POST, $filters);

if ($inputs['email'] === false) {
    echo "Érvénytelen email!";
}
```

4.28. kódpélda. *`filter_input_array()` használata*

A `filter_input_array()` előnye, hogy egyetlen hívással több mező is feldolgozható, és a szűrési szabályok jól olvasható konfigurációs tömbben adhatók meg, ahelyett hogy minden mezőt külön olvasnánk be.

4.5.3. `filter_var()`

A `filter_var()` függvény tetszőleges változó értékének validálására vagy szűrésére szolgál. Míg a `filter_input()` közvetlenül a HTTP-kérés szuperglobális tömbjeiből olvas, a `filter_var()` már meglévő változókra alkalmazható.

A szűrők két fő csoportba sorolhatók:

- **Validációs szűrők**: ellenőrzik, hogy az érték megfelel-e egy adott formátumnak vagy feltételnek. Siker esetén az értéket adják vissza, szükség esetén a megfelelő típusra alakítva (a `FILTER_VALIDATE_INT` például egész számként), hiba esetén `false` értéket.
- **Tisztító (sanitize) szűrők**: eltávolítják vagy átalakítják a nem megengedett karaktereket, és megtisztított értéket adnak vissza.

Szintaxis

```
filter_var(
    mixed $value,
    int $filter = FILTER_DEFAULT,
    array|int $options = 0
): mixed
```

Paraméterek:

- `$value`: az ellenőrizendő vagy szűrendő érték;
- `$filter` (opcionális): az alkalmazandó szűrő konstansa. Alapértelmezett értéke: `FILTER_DEFAULT`;
- `$options` (opcionális): szűrőspecifikus beállítások (például `min_range`, `max_range`).

Visszatérési érték:

- validációs szűrők esetén: a szűrt érték vagy `false`;
- tisztító szűrők esetén: a megtisztított érték.

```php
$email = "user@example.com";

// Email validálása
if (filter_var($email, FILTER_VALIDATE_EMAIL) !== false) {
    echo "Érvényes email cím.";
} else {
    echo "Érvénytelen email cím.";
}

// Egész szám validálása tartománnyal
$age = 25;
$options = [
    "options" => [
        "min_range" => 18,
        "max_range" => 99
    ]
];

if (filter_var($age, FILTER_VALIDATE_INT, $options) !==
    false) {
    echo "Érvényes életkor.";
} else {
    echo "Érvénytelen életkor.";
}
```

4.29. kódpélda. *`filter_var()` használata*

4.3. táblázat. *Gyakran használt* `filter_var()` *szűrők*

Szűrő konstans	Leírás
Validációs szűrők	
FILTER_VALIDATE_EMAIL	E-mail-cím formátumának ellenőrzése
FILTER_VALIDATE_URL	URL formátumának ellenőrzése
FILTER_VALIDATE_IP	IP-cím formátumának ellenőrzése
FILTER_VALIDATE_INT	Egész szám ellenőrzése (opcionális tartománnyal)
FILTER_VALIDATE_FLOAT	Lebegőpontos szám ellenőrzése
FILTER_VALIDATE_BOOL	Logikai érték ellenőrzése
Tisztító (sanitize) szűrők	
FILTER_SANITIZE_EMAIL	Nem megengedett karakterek eltávolítása e-mail-címből
FILTER_SANITIZE_URL	URL-ben nem engedélyezett karakterek eltávolítása
FILTER_SANITIZE_NUMBER_INT	Csak számjegyek és előjel megtartása
FILTER_SANITIZE_SPECIAL_CHARS	HTML speciális karakterek kódolása

A validációs szűrők `false` értékkel jelzik az érvénytelen bemenetet, a tisztító szűrők pedig átalakított, megtisztított értéket adnak vissza. A kettő célja eltérő, ezért nem helyettesíthetik egymást.

4.5.4. `preg_match()`

A `preg_match()` függvény reguláris kifejezés segítségével megvizsgálja, hogy egy sztring illeszkedik-e egy adott mintára. Opcionálisan lehetőséget biztosít a részegyezések (részminták) kinyerésére is. A reguláris kifejezések részletes ismertetése meghaladja a könyv kereteit, a webprogramozásban azonban nélkülözhetetlenek, ezért érdemes velük külön is foglalkozni. A PHP a Perl-kompatibilis (PCRE) szintaxist használja, amelyhez számos oktatóanyag érhető el.

Szintaxis

```
preg_match(
    string $pattern,
    string $subject,
    array  &$matches = null,
    int    $flags = 0,
    int    $offset = 0
): int|false
```

Paraméterek:

- `$pattern`: a reguláris kifejezés (határoló karakterekkel);
- `$subject`: az ellenőrizendő sztring;
- `$matches` (opcionális): az illeszkedő részeket tartalmazó tömb
  - `$matches[0]`: a teljes egyezés.
  - `$matches[1..n]`: a zárójelezett részminták eredményei.
- `$flags`, `$offset` (opcionális): a keresés viselkedését szabályozó paraméterek.

Visszatérési érték:

- `1`: ha a minta illeszkedik a sztringre;
- `0`: ha nincs illeszkedés;
- `false`: hiba esetén (például hibás reguláris kifejezés esetén).

Egy egyszerű példa, ahol ellenőrizzük, hogy egy irányítószám 6 számjegyből áll-e:

```php
$zip = "400001";

if (preg_match('/^\d{6}$/', $zip)) {
    echo "Érvényes irányítószám.";
} else {
    echo "Érvénytelen irányítószám.";
}
```

4.30. kódpélda. *Irányítószám ellenőrzése*

A `$matches` paraméter segítségével a reguláris kifejezés zárójelezett részei külön is kinyerhetők:

```php
$email = "kovacs.peter@example.com";

if (preg_match('/^([^@]+)@(.+)$/', $email, $matches)) {
    echo "Felhasználónév: " . $matches[1] . "<br>";
    echo "Domain: " . $matches[2];
}
```

4.31. kódpélda. *Részminták kinyerése*

A jelszóellenőrzés során gyakran több feltételnek kell egyszerre teljesülnie. Erre alkalmasak az előretekintő feltételek (*lookahead assertions*, `(?=...)`), amelyek lehetővé teszik, hogy a különböző feltételek egymástól függetlenül kerüljenek ellenőrzésre:

```php
$password = "Abcd1234!";

$pattern = "/^(?=.*[a-z])(?=.*[A-Z])(?=.*\d)(?=.*[@$!%*?&])[A
    -Za-z\d@$!%*?&]{8,}$/";

if (preg_match($pattern, $password)) {
    echo "A jelszó megfelel a követelményeknek.";
} else {
    echo "A jelszó nem felel meg a követelményeknek.";
}
```

4.32. kódpélda. *Jelszó ellenőrzése reguláris kifejezéssel*

A minta négy előretekintő feltétele rendre kisbetűt, nagybetűt, számjegyet és egy speciális karaktert követel meg, a záró rész pedig a minimális hosszt (8 karakter) írja elő. A példa szándékosan egyszerűsített: speciális karakterként csak a felsorolt jeleket (`@$!%*?&`) fogadja el, így például az aláhúzást nem, betűként pedig csak az angol ábécé betűit, ezért az ékezetes karaktereket tartalmazó jelszót elutasítja. Éles alkalmazásban a speciális karaktereket a `[[:punct:]]` karakterosztállyal, az ékezetes betűket pedig a `\p{L}` Unicode-osztállyal (a `/u` módosítóval együtt) kezelhetjük, ez azonban a példát feleslegesen bonyolítaná.

4.6. Biztonsági megfontolások

Az űrlapokból érkező adatok alapvetően nem tekinthetők megbízhatónak, ezért minden felhasználói bemenetet validálni és szükség esetén tisztítani kell (OWASP Foundation 2021). Ebben a részben a leggyakoribb webes támadástípusokat és az ellenük alkalmazható védekezési technikákat tekintjük át.

4.6.1. XSS (Cross-Site Scripting) elleni védelem

Az XSS-támadások[1] során a támadó olyan bemenetet juttat az alkalmazásba, amely a böngészőben futtatható kódot (például JavaScriptet) tartalmaz, és ezáltal képes műveleteket végrehajtani a felhasználó böngészőjében.

A védekezés alapja a kimeneti adatok megfelelő HTML-kódolása.

```php
/* HELYTELEN - XSS-re sérülékeny */
echo "<p>Üdvözöljük, " . $_POST['name'] . "!</p>";

/* HELYES - biztonságos kimeneti kezelés */
echo "<p>Üdvözöljük, " . htmlspecialchars($_POST['name'],
    ENT_QUOTES, 'UTF-8') . "!</p>";
```

4.33. kódpélda. *XSS-védelem*

Ez a módszer akkor megfelelő, ha a felhasználótól származó adatot egyszerű szövegként kell megjeleníteni. Ha az alkalmazásnak formázott tartalmat, azaz bizonyos HTML-elemeket is át kell engednie (például egy fórum hozzászólásaiban), akkor a kimenetet nem kódolni, hanem az engedélyezett elemekre szűrni kell. Erre a célra kész könyvtárak állnak rendelkezésre, például a Symfony HTML Sanitizer vagy a HTML Purifier.

4.6.2. CSRF (Cross-Site Request Forgery) elleni védelem

A CSRF-támadás során a támadó egy bejelentkezett felhasználót arra késztet, hogy akarata ellenére kérést küldjön a webalkalmazás felé.

A védekezés egyik leggyakoribb módja a tokenalapú ellenőrzés, amely biztosítja, hogy a kérés az eredeti felhasználótól származik.

```php
<?php
session_start();
// CSRF token generálása
if (empty($_SESSION['csrf_token'])) {
    $_SESSION['csrf_token'] = bin2hex(random_bytes(32));
}
// A beküldött token ellenőrzése az űrlap kiírása előtt
if ($_SERVER['REQUEST_METHOD'] === 'POST') {
    $submittedToken = $_POST['csrf_token'] ?? '';
    if (!hash_equals($_SESSION['csrf_token'],
        $submittedToken)) {
        die('Érvénytelen CSRF token!');
    }
    // Sikeres ellenőrzés után új token generálása
```

[1] https://owasp.org/www-community/attacks/xss/

```
    $_SESSION['csrf_token'] = bin2hex(random_bytes(32));
}
?>
<form method="POST" action="">
    <input type="hidden" name="csrf_token"
           value="<?=
              htmlspecialchars($_SESSION['csrf_token']) ?>">
    <input type="text" name="username">
    <input type="submit" value="Küldés">
</form>
```

4.34. kódpélda. *CSRF-védelem implementálása*

Megjegyzés

A CSRF-védelem a `session_start()` és a munkamenet (`$_SESSION`) használatára épül. Ezek működésével részletesebben a következő fejezet foglalkozik.

4.6.3. Jelszókezelés

A jelszavak tárolása kiemelt biztonsági kockázatot jelent, ezért azokat soha nem szabad nyers (plain text) formában eltárolni, hanem kizárólag kriptográfiai hash formájában.

PHP-ben a jelszókezeléshez három alapvető függvény tartozik:

- `password_hash()`: biztonságos hash készítése a jelszóból;
- `password_verify()`: a megadott jelszó ellenőrzése a tárolt hash alapján;
- `password_needs_rehash()`: ellenőrzi, hogy a hash megfelel-e az aktuális beállításoknak.

`password_hash()`

A `password_hash()` függvény a jelszóból biztonságos hash-t készít.

Szintaxis

```
password_hash(
    string $password,
    string|int|null $algo = PASSWORD_DEFAULT,
    array $options = []
): string
```

Paraméterek:

- `$password`: a jelszó, amelyből a hash készül;
- `$algo`: a hash-algoritmus (ajánlott: `PASSWORD_DEFAULT`);
- `$options`: algoritmusfüggő beállítások (pl. `cost`).

Visszatérési érték: a generált jelszóhash.

`password_verify()`

A `password_verify()` függvény ellenőrzi, hogy a megadott jelszó megfelel-e a tárolt hashnek. A hash tartalmazza az elkészítésekor használt algoritmust és annak beállításait, ezért ezeket az ellenőrzéskor nem kell külön megadni.

Szintaxis

```
password_verify(
    string $password,
    string $hash
): bool
```

Paraméterek:

- `$password`: a felhasználó által megadott jelszó;
- `$hash`: a tárolt jelszóhash.

Visszatérési érték: `true`, ha a jelszó helyes, `false`, ha a jelszó helytelen.

`password_needs_rehash()`

A `password_needs_rehash()` függvény ellenőrzi, hogy a tárolt hash megfelel-e az aktuális algoritmusnak és beállításoknak.

Szintaxis

```
password_needs_rehash(
    string $hash,
    string|int|null $algo,
    array $options = []
): bool
```

Paraméterek:

- `$hash`: a tárolt jelszóhash;

- `$algo`: a hash-algoritmus;
- `$options`: algoritmusfüggő beállítások.

Visszatérési érték: `true`, ha a hash frissítése szükséges, `false`, ha a hash megfelelő.

A következő példa bemutatja a teljes jelszókezelési folyamatot:

```php
// Regisztráció
$password = $_POST['password'];
$hashedPassword = password_hash($password, PASSWORD_DEFAULT);

// Bejelentkezés
$inputPassword = $_POST['password'];
$storedHash = '...';

if (password_verify($inputPassword, $storedHash)) {
    echo "Sikeres bejelentkezés!";

    if (password_needs_rehash($storedHash,
        PASSWORD_DEFAULT)) {
        $newHash = password_hash($inputPassword,
            PASSWORD_DEFAULT);
        // adatbázis frissítése
    }
}
```

4.35. kódpélda. *Biztonságos jelszókezelés*

Megjegyzés

- Soha ne használjunk MD5-öt vagy SHA-1-et jelszavak hashének előállítására.
- A `PASSWORD_DEFAULT` mindig az aktuálisan ajánlott algoritmust használja.

4.7. Fájlfeltöltés kezelése

A fájlfeltöltés kezelése fokozott odafigyelést igényel mind a funkcionalitás, mind a biztonság szempontjából.

4.7.1. Űrlap előkészítése

Fájlfeltöltéshez a szokásos `method="POST"` mellett az `enctype="multipart/form-data"` attribútum is kötelező. Enélkül

a böngésző nem küldi el a fájl tartalmát, csak a fájlnevet szövegként. A feltöltendő mezőt `<input type="file">` elemmel hozzuk létre:

```
<form action="upload.php" method="POST"
    enctype="multipart/form-data">
     <input type="file" name="fileToUpload">
     <input type="submit" value="Feltöltés">
</form>
```

4.36. kódpélda. *Fájlfeltöltő űrlap*

4.7.2. A $_FILES szuperglobális

A fájlok típusát MIME-típussal azonosítjuk. Ez egy szabványos szöveges azonosító, amely meghatározza a fájl tartalmát (pl. `image/jpeg`, `application/pdf`, `text/plain`). Feltöltéskor a böngésző is küld egy MIME-típust, erre azonban nem szabad támaszkodni, mert hamisítható. A tényleges típust a szerveren, a fájl tartalmából kell meghatározni.

Sikeres feltöltés után a fájl adatai a `$_FILES` szuperglobális tömbben érhetők el, az `<input>` mező `name` attribútuma alapján indexelve. Minden feltöltött fájlhoz öt kulcs tartozik (lásd a 4.4. táblázatot):

4.4. táblázat. *A $_FILES tömb kulcsai*

Kulcs	Leírás
`name`	Az eredeti fájlnév a felhasználó gépén
`type`	A böngésző által küldött MIME-típus (nem megbízható, hamisítható)
`size`	A fájl mérete bájtokban
`tmp_name`	Az ideiglenes fájl elérési útja a szerveren
`error`	Hibakód (`UPLOAD_ERR_OK` = 0 sikeres feltöltés esetén)

```
$file = $_FILES["fileToUpload"];
echo "Fájlnév: " . htmlspecialchars($file["name"]) . "<br>";
echo "Típus: "   . htmlspecialchars($file["type"]) . "<br>";
echo "Méret: "   . $file["size"] . " bytes<br>";
echo "Hibakód: " . $file["error"];
```

4.37. kódpélda. *$_FILES tömb használata*

Fontos megjegyezni, hogy a böngésző által küldött `type` mező hamisítható, ezért a fájl tényleges típusát szerveroldalon kell meghatározni a fájl tartalmából.

4.7.3. Fájlfeltöltési hibakódok

A feltöltés eredményét az `error` mező jelzi (lásd a 4.5. táblázatot), ezért az értékét feldolgozás előtt mindig ellenőrizni kell.

4.5. táblázat. *Fájlfeltöltési hibakódok*

Konstans	Érték	Jelentés
`UPLOAD_ERR_OK`	0	Sikeres feltöltés
`UPLOAD_ERR_INI_SIZE`	1	A fájl nagyobb, mint a `php.ini` `upload_max_filesize` beállítása
`UPLOAD_ERR_FORM_SIZE`	2	A fájl nagyobb, mint a `MAX_FILE_SIZE` űrlapmezőben megadott érték (könnyen megkerülhető korlát)
`UPLOAD_ERR_PARTIAL`	3	A fájl csak részben töltődött fel
`UPLOAD_ERR_NO_FILE`	4	Nem töltöttek fel fájlt
`UPLOAD_ERR_NO_TMP_DIR`	6	Hiányzik az ideiglenes mappa
`UPLOAD_ERR_CANT_WRITE`	7	Nem sikerült lemezre írni
`UPLOAD_ERR_EXTENSION`	8	Egy PHP-kiterjesztés megszakította a fájlfeltöltési folyamatot

A feltölthető fájl maximális méretét a szerveroldali `php.ini` konfigurációs fájl `upload_max_filesize` beállítása szabályozza. Ugyanott a `post_max_size` értékének is legalább ekkorának kell lennie, mert ez a teljes POST-kérés méretét korlátozza, amelybe a feltöltött fájl is beletartozik.

4.7.4. Biztonságos fájlfeltöltés

Fájlfeltöltésnél a kiterjesztés ellenőrzése önmagában nem elegendő: a MIME-típust közvetlenül a fájl tartalmából kell meghatározni, mivel a böngésző által küldött `type` mező hamisítható.

A célkönyvtár (`uploads/`) létezése és írhatósága a sikeres mentés feltétele, mivel a webszervernek írási jogosultságra van szüksége.

Az alábbi példa a 4.4. és 4.5. táblázat mezői alapján hibakód-, méret-, kiterjesztés- és MIME-ellenőrzést végez, majd véletlenszerű fájlnevet generál:

```php
$errors = [];
$file = $_FILES['fileToUpload'] ?? null;
if ($file === null || $file['error'] === UPLOAD_ERR_NO_FILE)
    {
    $errors[] = "Nem választottunk ki fájlt.";
} elseif ($file['error'] !== UPLOAD_ERR_OK) {
    $errors[] = "Feltöltési hiba (kód: {$file['error']}).";
} elseif (!is_uploaded_file($file['tmp_name'])) {
    // Ellenőrzés: a fájl valóban HTTP POST feltöltésből
        származik-e
    $errors[] = "Érvénytelen feltöltés.";
} else {
    // Kiterjesztés kinyerése: a pathinfo() a fájlnév részeit
    // (könyvtár, név, kiterjesztés) adja vissza
    $extension = strtolower(pathinfo($file['name'],
        PATHINFO_EXTENSION));
    // Méret ellenőrzése (max 5MB)
    if ($file['size'] > 5 * 1024 * 1024) {
        $errors[] = "A fájl túl nagy.";
    }
    // Kiterjesztés ellenőrzése
    $allowedExtensions = ["jpg", "jpeg", "png", "gif",
        "pdf"];
    if (!in_array($extension, $allowedExtensions)) {
        $errors[] = "Nem engedélyezett fájltípus.";
    }
    // MIME típus ellenőrzése a fájl tartalma alapján
    $finfo = new finfo(FILEINFO_MIME_TYPE);
    $mime = $finfo->file($file['tmp_name']);
    $allowedMimes = ['image/jpeg', 'image/png', 'image/gif',
        'application/pdf'];
    if (!in_array($mime, $allowedMimes)) {
        $errors[] = "Nem megengedett MIME típus.";
    }
    // Feltöltés biztonságos, véletlen fájlnévvel
    if (empty($errors)) {
        $safeFilename = bin2hex(random_bytes(16)) . '.' .
            $extension;
        if (!move_uploaded_file($file['tmp_name'],
            "uploads/" . $safeFilename)) {
            $errors[] = "A fájl mentése nem sikerült.";
        }
    }
}
```

4.38. kódpélda. *Biztonságos fájlfeltöltés*

A példa több egymást követő ellenőrzést alkalmaz. Először a hibakódot vizsgálja, és ha nem érkezett fájl, vagy a feltöltés hibás volt, a további lépéseket kihagyja. Erre azért van szükség, mert hiányzó fájl esetén a `tmp_name` mező üres, és a `finfo->file()` hívás hibát okozna. Ezután ellenőrzi, hogy a fájl valóban HTTP POST-feltöltésből származik-e, majd a fájlméretet, az engedélyezett kiterjesztéseket, végül a fájltartalomból meghatározott MIME-típust. Az űrlap `<input type="file">` mezőjén a `required` attribútummal már kliensoldalon megakadályozható az üres beküldés, a szerveroldali ellenőrzés azonban ekkor sem hagyható el. Az eredeti fájlnév helyett véletlenszerű nevet generálunk, hogy elkerüljük a névütközéseket és a potenciálisan veszélyes fájlneveket.

Megjegyzés

Nagyobb PHP-alkalmazásokban, sok beviteli mezőt tartalmazó űrlapok esetén az adatok ellenőrzését gyakran erre a célra készült komponensek vagy a használt keretrendszer validációs szolgáltatásai végzik. Ezek lehetővé teszik, hogy az egyes mezőkre vonatkozó szabályokat deklaratív módon, egy helyen adjuk meg. Ilyen például a Respect\Validation könyvtár vagy a Symfony Validator komponens.

4.8. Gyakorlófeladatok

1. Készítsünk termékkeresési szkriptet GET-metódus használatával. Az űrlap tartalmazza az alábbi mezőket:

   - szöveges keresőmező (terméknév);
   - minimum és maximum ár (számbeviteli mező);
   - kategória (legördülő lista).

   A PHP-szkript jelenítse meg a beküldött keresési feltételeket. Mutassuk be a `htmlspecialchars()` szükségességét: írjuk ki ugyanazt az értéket egyszer kódolás nélkül, egyszer védve. Próbáljuk ki, mi történik, ha a keresőmezőbe a következő érték kerül: `<script>alert('XSS')</script>`.

2. Készítsünk regisztrációs szkriptet POST-metódussal. Az űrlap tartalmazza az alábbi mezőket:

   - felhasználónév, e-mail-cím, életkor és telefonszám;
   - jelszó (jelszómező);
   - nem (rádiógomb: férfi / nő);

- érdeklődési körök (jelölőnégyzetek, legalább 4 opció);
- lakhely (legördülő lista).

Az összetartozó mezőket `user[name]`, `user[email]` formátumban adjuk meg. A rádiógomb értékét az engedélyezett értékek listája alapján ellenőrizzük. A jelölőnégyzetek tömbjét `htmlspecialchars()` függvénnyel kezeljük, majd jelenítsük meg az összes beküldött adatot.

3. Bővítsük a 2. feladatban készített regisztrációs szkriptet szerveroldali validációval. Az adatok beolvasásához és szűréséhez használjuk a `filter_input_array()` függvényt, és végezzük el az alábbi ellenőrzéseket:

   - e-mail-cím: `FILTER_VALIDATE_EMAIL` szűrővel;
   - életkor: `FILTER_VALIDATE_INT` szűrővel (18–99 intervallumban);
   - telefonszám formátuma: `preg_match()` segítségével.

   A hibákat gyűjtsük össze egy tömbben, és hiba esetén jelenítsük meg listaként az űrlap felett. Valósítsuk meg az állapotmegőrzést: hiba esetén a korábban megadott adatok maradjanak a mezőkben.

4. Alakítsuk át a 3. feladatban megvalósított regisztrációs szkriptet a Post/Redirect/Get (PRG) mintának megfelelően:

   - sikeres beküldés esetén irányítsuk át a felhasználót `header()` segítségével ugyanarra az oldalra, `?success=1` paramétert hozzáfűzve az URL-hez;
   - a `header()` hívása után azonnal hívjuk meg az `exit` utasítást;
   - a GET-tel betöltött oldalon ellenőrizzük a `$_GET['success']` értékét, és ha be van állítva, jelenítsük meg a visszajelzést;
   - hiba esetén maradjunk az eredeti oldalon, és jelenítsük meg a hibákat.

5. Valósítsunk meg egy kétlépéses regisztrációs folyamatot rejtett mezők segítségével:

   - az első lépésben kérjünk be alapadatokat (felhasználónév, e-mail-cím, jelszó);
   - az elküldés után ezek az értékek rejtett (`hidden`) mezőkként kerüljenek át a második oldalra;
   - a második lépésben kérjünk be további adatokat (telefonszám, lakhely, születési év);

- a végső beküldéskor jelenítsük meg az összes begyűjtött adatot.

A rejtett mezők értékét megjelenítés előtt minden esetben `htmlspecialchars()` függvénnyel kezeljük.

6. Készítsünk bejelentkezési szkriptet biztonságos jelszókezeléssel:
   - hozzunk létre egy szimulált felhasználói tömböt, amelyben a jelszavak a `password_hash()` függvénnyel előállított hash formában szerepelnek;
   - a bejelentkezési űrlap beküldésekor (POST-kérés) ellenőrizzük a megadott jelszót `password_verify()` függvénnyel;
   - ha a `password_needs_rehash()` függvény szerint a hash nem felel meg az aktuális beállításoknak, generáljunk újat, és frissítsük a tömbben;
   - jelenítsük meg, hogy sikeres vagy sikertelen volt-e a bejelentkezés.
7. Készítsünk profilkép-feltöltő szkriptet. A feltöltött fájlon végezzük el az alábbi ellenőrzéseket:
   - valóban HTTP POST-feltöltésből származik-e (`is_uploaded_file()`);
   - a hibakód értéke (lásd a 4.5. táblázatot);
   - a fájl mérete (maximum 2 MB);
   - a kiterjesztés az engedélyezett értékek listája alapján (jpg, jpeg, png).

   Sikeres ellenőrzés után mentsük el az `uploads/` mappába véletlenszerű névvel (`bin2hex(random_bytes(16))`), majd jelenítsük meg a feltöltött képet az oldalon.

5. FEJEZET

SÜTIK ÉS MUNKAMENETEK KEZELÉSE

5.1. Állapottartási technikák

A HTTP-protokoll alapvetően állapotmentes: minden kérés önálló, a szerver nem őrzi meg a korábbi kérésekhez tartozó információkat. Ennek ellenére a modern webalkalmazások gyakran igénylik a felhasználói állapot megőrzését, például bejelentkezett munkamenetek, kosárkezelés vagy felhasználói beállítások kezelése esetén. Az állapotkövetésre két fő megközelítés létezik: *kliensoldali* és *szerveroldali* megoldások.

5.1.1. Kliensoldali megoldások

Kliensoldali megoldások esetén az adatokat a böngésző tárolja, és minden kérésnél visszaküldi a szervernek. A három leggyakoribb megközelítés: az URL-paraméterek, a rejtett űrlapmezők és a sütik.

URL-paraméterek

A böngésző az állapotinformációkat az URL-ben, lekérdezési paraméterek formájában továbbítja. A szerver minden kérésnél a GET-paraméterekből olvassa ki az adatokat:

```
<?php
$count = isset($_GET['count']) ? (int)$_GET['count'] + 1 : 1;
?>
<h1>Oldal megtekintve: <?= $count ?> alkalommal</h1>
<a href="counter.php?count=<?= $count ?>">Következő</a>
```

5.1. kódpélda. *Kattintásszámláló URL-paraméterrel*

Az URL-paraméterek előnyeit és hátrányait az 5.1. táblázat foglalja össze.

5.1. táblázat. *URL-paraméterek előnyei és hátrányai*

Előnyök	Hátrányok
Egyszerű implementáció	Korlátozott adatmennyiség (az URL hossza korlátozott)
Böngészőfüggetlen működés	Biztonsági kockázat (az adatok láthatók)
Könyvjelzőzhető	Könnyen manipulálható
	Minden kérésnél elküldi a böngésző

Az URL-paraméterek nem alkalmasak érzékeny adatok továbbítására, mivel az adatok naplózásra kerülhetnek (például szervernaplókban vagy böngészőelőzményekben).

Rejtett űrlapmezők

Az állapotinformációkat HTML-űrlapok rejtett mezőiben tároljuk, és POST-kérésekkel küldjük vissza a szervernek:

```
<?php
$count = isset($_POST['count']) ? (int)$_POST['count'] + 1 :
    1;
?>
<h1>Oldal megtekintve: <?= $count ?> alkalommal</h1>
<form method="POST" action="counter.php">
    <input type="hidden" name="count" value="<?= $count ?>">
    <input type="submit" value="Következő">
</form>
```

5.2. kódpélda. *Kattintásszámláló rejtett mezővel*

A rejtett mezők előnyeit és hátrányait az 5.2. táblázat foglalja össze.

5.2. táblázat. *Rejtett mezők előnyei és hátrányai*

Előnyök	Hátrányok
Nem jelenik meg az URL-ben	Csak űrlapok esetén használható
Nagyobb adatmennyiség továbbítható, mint URL-paraméterekkel	Kliensoldalon módosítható
Böngészőfüggetlen működés	Minden kérésnél elküldi a böngésző

Sütik

A sütik a böngészőben tárolt kis méretű adatok, amelyeket a böngésző automatikusan minden kérésnél elküld a szervernek. Az alábbi kódrészlet a sütiben való adattárolás lényegét mutatja be: a `setcookie()` függvény sütit küld a böngészőnek, a `$_COOKIE` szuperglobális tömbből pedig a visszaérkező értéket olvassuk ki. A sütik kezelését az 5.2. alfejezetben mutatjuk be részletesen.

```
<?php
$count = isset($_COOKIE['count']) ? (int)$_COOKIE['count'] +
    1 : 1;
setcookie('count', $count, time() + 3600); // 1 óra lejárat
?>
<h1>Oldal megtekintve: <?= $count ?> alkalommal</h1>
<a href="counter.php">Következő</a>
```

5.3. kódpélda. *Kattintásszámláló sütikkel*

A sütik előnyeit és hátrányait az 5.3. táblázat foglalja össze.

5.3. táblázat. *Sütik előnyei és hátrányai*

Előnyök	Hátrányok
Automatikusan minden kérésnél elküldi a böngésző	A felhasználó letilthatja
Lejárati idő beállítható	Korlátozott méret (kb. 4 KB)
Domain és útvonal alapján korlátozható	Kliensoldalon módosítható

5.1.2. Szerveroldali megoldások

Szerveroldali megoldások (munkamenet, *session*) esetén a szerver tárolja az állapotinformációkat, a kliens pedig csak egy azonosítót (munkamenet-azonosítót) kap, általában süti formájában.

Ez a megoldás általában biztonságosabb, mivel az adatokat a szerver tárolja, a kliensoldalon pedig csak a munkamenet-azonosító található meg. Emellett nem szükséges minden kérésnél nagy mennyiségű adatot továbbítani:

```
<?php
session_start();
$_SESSION['count'] = isset($_SESSION['count']) ?
    $_SESSION['count'] + 1 : 1;
?>
<h1>Oldal megtekintve: <?= $_SESSION['count'] ?>
    alkalommal</h1>
<a href="counter.php">Következő</a>
```

5.4. kódpélda. *Kattintásszámláló munkamenettel*

A `session_start()` elindítja (vagy folytatja) a munkamenetet, és a `$_SESSION` tömbben tárolt adatok automatikusan megmaradnak a következő kérésig, így a számlálót nem kell URL-paraméterben vagy rejtett mezőben továbbadni. A munkamenetek kezelését az 5.3. alfejezetben tárgyaljuk részletesen.

5.2. Sütik létrehozása és kezelése

5.2.1. Alapfogalmak

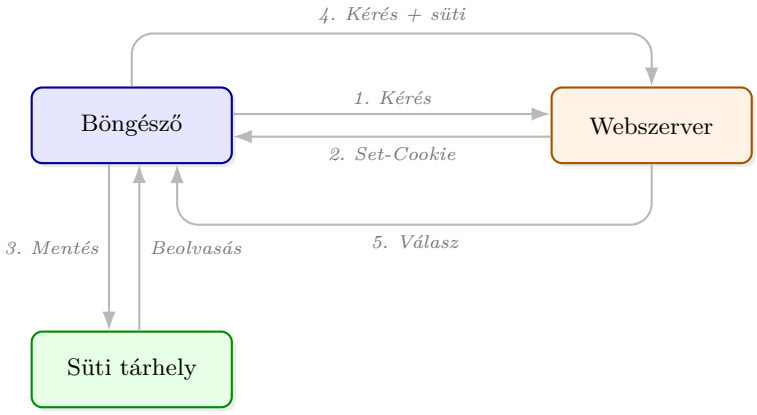
A süti (*cookie*) a böngészőben tárolt kis méretű, kulcs–érték párokból álló adat, amelyet a böngésző a webszerver kérésére eltárol, majd a későbbi HTTP-kérések során automatikusan visszaküld.[1] A sütik lehetővé teszik a webszerverek számára, hogy „emlékezzenek” a felhasználókra és azok beállításaira, bár a HTTP-protokoll alapvetően állapotmentes.

A sütik működése a következő lépésekből áll (lásd az 5.1. ábrát):

1. A felhasználó meglátogat egy weboldalt.
2. A webszerver a HTTP-válaszban utasítja a böngészőt egy vagy több süti tárolására.
3. A böngésző elmenti a sütiket a felhasználó eszközén.

[1] https://tools.ietf.org/html/rfc6265

4. A további kérések során a böngésző automatikusan elküldi a sütiket a szervernek.
5. A szerver a sütik alapján azonosítja a felhasználót vagy betölti annak beállításait.



5.1. ábra. *A sütik működésének folyamata*

A sütiket két fő kategóriába sorolhatjuk:

- **Tartós (persistent) sütik:** Meghatározott lejárati idővel rendelkeznek, és eddig az időpontig a felhasználó eszközén maradnak.
- **Munkamenet (session) sütik:** Nem rendelkeznek lejárati idővel, és a böngésző bezárásakor általában törlődnek.

Egy süti maximális mérete általában 4 KB, és egy domain alatt legalább 50, a mai böngészőkben jellemzően 150–180 süti tárolható.

5.2.2. Sütik kezelése PHP-ben

Süti létrehozása

PHP-ben a `setcookie()` függvénnyel hozhatunk létre sütiket:

Szintaxis

```
setcookie(
    string $name,
    string $value = "",
    array $options = []
): bool
```

Paraméterek:

- `$name`: a süti neve (kötelező);
- `$value` (opcionális): a süti értéke;
- `$options` (opcionális): asszociatív tömb a süti beállításaihoz:
  - `expires`: lejárati idő UNIX-időbélyegként megadva. Alapértelmezés szerint `0`, ekkor munkamenet-süti jön létre;
  - `path`: az útvonal, amelyen a süti érvényes;
  - `domain`: a domain, amelyen a süti érvényes;
  - `secure`: ha `true`, a süti csak HTTPS-kapcsolaton keresztül küldhető el;
  - `httponly`: ha `true`, a süti JavaScriptből nem érhető el, így védelmet nyújt bizonyos XSS-támadások ellen;
  - `samesite`: CSRF-védelmet biztosító beállítás; értéke `Strict`, `Lax` vagy `None` lehet. A `None` használatához kötelező a `secure=true` beállítás.

Visszatérési érték: `true`, ha a sütit tartalmazó `Set-Cookie` HTTP-fejlécet sikerült elküldeni (vagyis még nem történt kimenet), egyébként `false`.

Az alábbi példa három tipikus esetet mutat be: egyszerű süti, biztonsági beállításokkal rendelkező tartós süti és süti törlése.

```php
// Egyszerű süti 1 órás lejárattal
setcookie("felhasznalo", "kovacs_peter", ['expires' =>
    time() + 3600, 'path' => '/']);

// Tartós süti biztonsági beállításokkal
setcookie("beallitasok", "sotetmod=1", [
    'expires'  => time() + 30 * 24 * 60 * 60,  // 30 nap
    'path'     => '/',
    'secure'   => true,
    'httponly' => true,
    'samesite' => 'Strict'
]);

// Süti törlése (lejárt idővel)
setcookie("felhasznalo", "", ['expires' => time() - 3600,
    'path' => '/']);
```

5.5. kódpélda. *Sütik létrehozása különböző beállításokkal*

Visszafelé kompatibilitási okokból a hagyományos paraméterlistás szintaxis is használható: `setcookie($name, $value, $expires, $path, $domain, $secure, $httponly)`.

Sütik olvasása

A sütik értékét a `$_COOKIE` szuperglobális tömbből olvashatjuk ki:

```php
// Süti közvetlen olvasása
if (isset($_COOKIE['felhasznalo'])) {
    echo "Üdvözöljük, " .
        htmlspecialchars($_COOKIE['felhasznalo']) . "!";
}

// Olvasás alapértelmezett értékkel
$tema = $_COOKIE['tema'] ?? 'vilagos';

// Összes süti kiírása
foreach ($_COOKIE as $nev => $ertek) {
    echo htmlspecialchars($nev) . ": " .
         htmlspecialchars($ertek) . "<br>";
}
```

5.6. kódpélda. *Sütik olvasása*

Sütik törlése

A sütik törléséhez a `setcookie()` függvényt úgy használjuk, hogy a lejárati időt egy múltbeli időpontra állítjuk. Fontos, hogy a törléshez ugyanazokat a `path` és `domain` beállításokat kell megadni, amelyekkel a süti létrejött, különben a böngésző nem feltétlenül a megfelelő sütit törli.

```php
// Egyszerű törlés
setcookie("felhasznalo", "", time() - 3600);

// Törlés azonos path/domain beállításokkal
setcookie("felhasznalo", "", [
    'expires' => time() - 3600,
    'path'    => '/',
    'domain'  => ''
]);

// Összes süti törlése
foreach ($_COOKIE as $nev => $ertek) {
    setcookie($nev, "", time() - 3600);
}
```

5.7. kódpélda. *Sütik törlése*

Megjegyzés

- A törlés hatása csak a következő kérés során érvényesül; az aktuális kérésben a `$_COOKIE` tömb még tartalmazza az értéket.
- A törlés előtt célszerű ellenőrizni a süti létezését az `isset()` függvénnyel.
- Biztonságos sütik esetén a törléskor is meg kell adni a korábban használt beállításokat (például `path`, `secure`, `httponly`).

5.2.3. Biztonsági szempontok

A sütik kezelése során több biztonsági szempontot is figyelembe kell venni.

Biztonsági attribútumok

- **HttpOnly attribútum:** megakadályozza a süti JavaScriptből történő elérését, így csökkenti az XSS (Cross-Site Scripting) támadások kockázatát. Ha egy támadó rosszindulatú JavaScript-kódot futtat az oldalon, nem tudja kiolvasni a sütit.
- **Secure attribútum:** biztosítja, hogy a süti kizárólag HTTPS-kapcsolaton keresztül küldje el. Ez megakadályozza, hogy titkosítatlan HTTP-forgalom esetén a süti lehallgatható legyen.
- **SameSite attribútum:** a CSRF (Cross-Site Request Forgery) támadások ellen nyújt védelmet azáltal, hogy szabályozza, mikor küldheti el a böngésző a sütit cross-site kérésekkel együtt.
  - `Strict`: a böngésző a sütit csak azonos webhelyről érkező kérések esetén küldi el.
  - `Lax`: engedékenyebb beállítás, amelyet a Chromium-alapú böngészők (Chrome, Edge) alapértelmezetten alkalmaznak. Elküldi a sütit, ha a felhasználó egy külső oldalról navigál ide (például linkre kattint), de POST-kéréseknél nem.
  - `None`: nincs korlátozás, de használatához kötelező a `secure=true` beállítás.
- **Domain és Path:** meghatározza, hogy a süti mely domaineken és útvonalakon érhető el.
- **Érzékeny adatok:** ne tároljunk jelszavakat vagy személyes adatokat sütiben; helyettük munkamenet-azonosítót használjunk, az adatok pedig maradjanak a szerveren.

CSRF-védelem és a SameSite attribútum

A CSRF (*Cross-Site Request Forgery*) támadás során egy rosszindulatú weboldal kihasználja, hogy a felhasználó be van jelentkezve egy másik webalkalmazásba, és a felhasználó nevében hajt végre műveleteket.

Támadás menete példával:

1. A felhasználó be van jelentkezve a `peldabank.com` oldalra;
2. A munkamenet-azonosító (session ID) sütiben tárolódik a böngészőben;
3. A felhasználó meglátogat egy rosszindulatú weboldalt (`rosszoldal.com`);
4. A rosszindulatú oldal egy rejtett űrlapot küld a `peldabank.com` oldalra, például: „1000 RON átutalása a támadó számlájára”;
5. Ha nincs megfelelő védelem, a böngésző automatikusan elküldi a sütit, és a művelet végrehajtásra kerül.

A `SameSite=Strict` vagy `Lax` beállítás ezt részben vagy teljesen megakadályozza:

```php
// Bejelentkezési süti CSRF védelemmel
setcookie("session_id", $sessionId, [
    'expires'  => time() + 3600,
    'path'     => '/',
    'secure'   => true,
    'httponly' => true,
    'samesite' => 'Strict'  // Csak saját domainről érkező
        kérések
]);
```

5.8. kódpélda. *CSRF elleni védelem SameSite attribútummal*

Megjegyzés

- Ha a `samesite` attribútum nincs megadva, a Chromium-alapú böngészők `Lax` értéket alkalmaznak, a Firefox és a Safari viszont nem. Ezért érdemes mindig explicit módon megadni.
- Érzékeny műveletek (például fizetés, jelszóváltoztatás vagy adminisztrációs funkciók) esetén ajánlott explicit `Strict` értéket beállítani.
- A `SameSite` attribútum nem helyettesíti a klasszikus, tokenalapú CSRF-védelmet, hanem annak kiegészítéseként alkalmazandó.

5.2.4. HTTP-fejlécek küldése

A `header()` függvényt gyakran használjuk a sütik kezelésével együtt átirányítások és egyéb HTTP-fejlécek küldésére.

Szintaxis

```
header(
    string $header,
    bool $replace = true,
    int $response_code = 0
): void
```

Paraméterek:

- `$header`: a küldendő HTTP-fejléc;
- `$replace`: ha `true`, felülírja az azonos típusú korábban elküldött fejléceket;
- `$response_code`: opcionális HTTP-státuszkód.

A `header()` függvény leggyakoribb használati esetei:

```php
// Átirányítás másik oldalra
header("Location: login.php");

// HTTP válaszkód beállítása
header("HTTP/1.1 404 Not Found");

// Cache vezérlés
header("Cache-Control: no-cache, must-revalidate");

// Tartalomtípus beállítása
header("Content-Type: application/json");

// Fájl letöltésének kezdeményezése
header("Content-Disposition: attachment;
    filename=document.pdf");
```

5.9. kódpélda. *`header()` függvény használata*

Fontos szabályok:

- Átirányítás után ajánlott az `exit` utasítással leállítani a szkriptet;
- A `header()` függvény csak a még el nem küldött HTTP-fejléceket tudja módosítani.

Példa az átirányítás helyes használatára:

```php
if (!isset($_COOKIE['user'])) {
    header("Location: login.php");
    exit;
}

echo "Üdvözöljük az oldalon!";
```

5.10. kódpélda. *Átirányítás helyes használata*

Az átirányítás önmagában nem állítja le a PHP-szkript futását, ezért a `Location` fejléc küldése után ajánlott az `exit` utasítással leállítani a szkriptet.

5.2.5. A „headers already sent” hiba elkerülése

A sütiket és HTTP-fejléceket a HTTP-válasz fejlécében kell elküldeni, ezért a `setcookie()` és `header()` függvényeket minden esetben a kimenet (HTML) előtt kell meghívni.

```php
<?php
// Helyes: süti beállítása a kimenet előtt
setcookie("user", "John");
echo "Hello World!";

// Helytelen: hibát okoz
echo "Hello World!";
setcookie("user", "John"); // Hiba: Headers already sent
?>
```

5.11. kódpélda. *A „headers already sent” hiba elkerülése*

Gyakori okok:

- szóköz, üres sor vagy UTF-8 BOM (bájtsorrend-jelölő) a `<?php` címke előtt;
- `echo` vagy `print` utasítás a `header()` vagy `setcookie()` hívás előtt;
- include-olt fájl által generált kimenet.

Ha az architektúra nem teszi lehetővé a megfelelő sorrend betartását, az `ob_start()` és `ob_end_flush()` függvényekkel (*output buffering*) elkerülhető a hiba, mivel a PHP a kimenetet pufferelve küldi el. Megjegyzendő, hogy egyes PHP-telepítések alapértelmezetten bekapcsolt kimeneti pufferelést használnak (a `php.ini-production` és `php.ini-development` gyári beállítása `output_buffering = 4096`), ezért kis méretű kimenetnél a hiba nem feltétlenül jelentkezik. Erre azonban nem érdemes építeni, a fejlécek küldését minden esetben a kimenet elé kell szervezni.

5.2.6. Gyakorlati példák

A süti tartalmát a felhasználó a böngészőjében szabadon módosíthatja, ezért ellenőrzés nélkül csak olyan adatot érdemes benne tárolni, amelynek megváltoztatása nem okoz kárt (például nyelvi vagy megjelenítési beállítást). Az alábbi példákban a bejelentkezett felhasználó neve és a kosár tartalma kerül sütibe. Az állapot ilyen, sütiben történő tárolását kliensoldali munkamenetnek is nevezik. Ez a sütik működésének szemléltetésére alkalmas, éles alkalmazásban azonban az ilyen adatot vagy aláírt, illetve titkosított sütiben, vagy inkább szerveroldali munkamenetben tároljuk (lásd az 5.3. alfejezetet).

Bejelentkezés kezelése

A bejelentkezés logikája három lépésből áll: a süti létrehozása bejelentkezéskor, a süti ellenőrzése a védett oldalakon, valamint a süti törlése kijelentkezéskor.

```php
// login.php - bejelentkezés és süti beállítása
if ($_SERVER['REQUEST_METHOD'] === 'POST') {
    $username = $_POST['username'] ?? '';
    $password = $_POST['password'] ?? '';

    if (password_verify($password, $storedHash)) {
        setcookie("felhasznalo", $username, [
            'expires'  => time() + 3600,
            'path'     => '/',
            'secure'   => true,
            'httponly' => true,
            'samesite' => 'Strict'
        ]);

        header("Location: fooldal.php");
        exit;
    }
}

// fooldal.php - védett oldal ellenőrzése
if (!isset($_COOKIE['felhasznalo'])) {
    header("Location: login.php");
    exit;
}

echo "Üdvözöljük, "
    . htmlspecialchars($_COOKIE['felhasznalo'], ENT_QUOTES,
        'UTF-8')
    . "!";
```

```php
// kijelentkezes.php - süti törlése
setcookie("felhasznalo", "", [
    'expires' => time() - 3600,
    'path'    => '/'
]);

header("Location: login.php");
exit;
```

5.12. kódpélda. *Bejelentkezés kezelése sütikkel*

A példa három részből áll: bejelentkezéskor a felhasználónév sütiben tárolódik, a védett oldal ellenőrzi a süti meglétét és szükség esetén visszairányítja a felhasználót, kijelentkezéskor pedig a süti lejárt időre állítva törlődik.

Bevásárlókosár megvalósítása

A kosár tartalmát JSON-formátumban tároljuk sütiben, amely lehetővé teszi összetettebb adatok (például termékek és mennyiségek) egyszerű kezelését. A sütiben tárolt adatokat minden kérésnél kiolvassuk, majd visszaalakítjuk tömbbé.

```php
// Kosár betöltése sütiből
$kosar = isset($_COOKIE['kosar'])
    ? json_decode($_COOKIE['kosar'], true) ?? []
    : [];

// Termék hozzáadása
if (isset($_POST['hozzaad'])) {
    $termekId = (int) $_POST['termek_id'];

    $kosar[$termekId] = ($kosar[$termekId] ?? 0) + 1;

    setcookie('kosar', json_encode($kosar), [
        'expires' => time() + 86400,
        'path'    => '/'
    ]);
}

// Termék eltávolítása
if (isset($_POST['torol'])) {
    $termekId = (int) $_POST['termek_id'];

    unset($kosar[$termekId]);
```

```
    setcookie('kosar', json_encode($kosar), [
        'expires' => time() + 86400,
        'path'    => '/'
    ]);
}

// Kosár kiírása (példaárakkal)
foreach ($kosar as $termekId => $darab) {
    $ar = $termekArak[$termekId] ?? 0;

    echo "Termék #$termekId: $darab db - "
        . number_format($ar * $darab, 2)
        . " RON<br>";
}
```

5.13. kódpélda. *Bevásárlókosár sütikkel*

5.3. Munkamenetek használata

5.3.1. Alapfogalmak

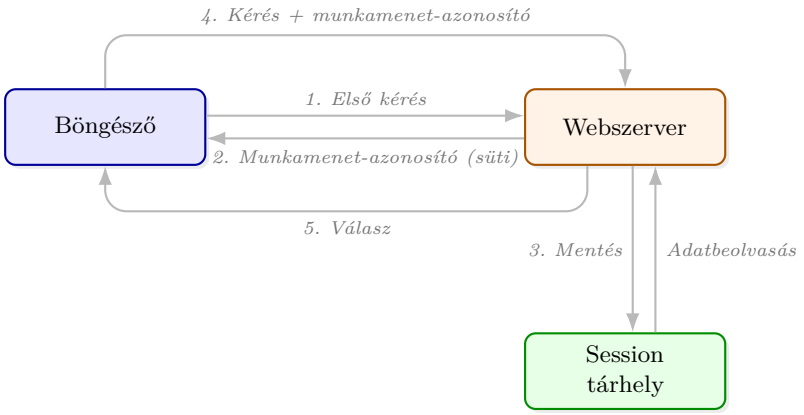
A munkamenet (*session*) egy mechanizmus, amely lehetővé teszi az adatok tárolását különböző oldalletöltések között. A PHP szerveroldali tárolást valósít meg, ahol csak egy egyedi azonosítót (munkamenet-azonosító, alapértelmezés szerint `PHPSESSID` nevű süti formájában) küld el a kliensnek. Ez biztonságosabb megoldást nyújt, mint az adatok közvetlen sütiben történő tárolása.

A munkamenet működése a következő lépésekből áll (lásd az 5.2. ábrát):

1. A felhasználó első látogatásakor a szerver létrehoz egy egyedi munkamenet-azonosítót;
2. A szerver elküldi ezt az azonosítót süti formájában a kliensnek;
3. A szerver a munkamenet adatait szerveroldalon tárolja (fájlban vagy más tárolóban);
4. A következő kéréseknél a kliens visszaküldi a munkamenet-azonosítót;
5. A szerver az azonosító alapján betölti a megfelelő munkamenetadatokat.

A munkamenetadatok tárolása többféleképpen történhet:

- **Fájlalapú tárolás:** az adatok a szerver fájlrendszerében tárolódnak (alapértelmezett megoldás);



5.2. ábra. *A munkamenetek működésének folyamata*

- **Adatbázis-alapú tárolás:** az adatok adatbázisban tárolódnak;
- **Memóriaalapú tárolás:** az adatok a szerver memóriájában tárolódnak (például Redis vagy Memcached használatával).

Megjegyzés

Alapértelmezés szerint a PHP a munkamenetadatokat a szerver fájlrendszerében tárolja, egy-egy fájlban munkamenetenként. A könyvtárat a `php.ini` `session.save_path` beállítása határozza meg: ez Linuxon tipikusan a `/tmp`, Debian és Ubuntu rendszereken a `/var/lib/php/sessions`, XAMPP esetén pedig a telepítés `tmp` mappája.

5.3.2. Munkamenetek kezelése PHP-ben

Munkamenet indítása

A munkamenet kezelését a `session_start()` függvénnyel kezdeményezzük.

Szintaxis

```
session_start(array $options = []): bool
```

Paraméterek:

- `$options` (opcionális): asszociatív tömb a munkamenet beállításaihoz (például `cookie_lifetime`, `cookie_secure`).

Visszatérési érték: `true`, ha a munkamenet sikeresen elindult, egyébként `false`.

Az alábbi példa a javasolt biztonsági beállításokat mutatja:

```php
session_start([
    'cookie_lifetime' => 3600,      // süti élettartama
        másodpercben
    'cookie_secure'   => true,      // csak HTTPS esetén
    'cookie_httponly' => true,      // JavaScript nem férhet
        hozzá
    'cookie_samesite' => 'Strict'  // CSRF védelem
]);
```

5.14. kódpélda. *Munkamenet indítása*

> **Megjegyzés**
>
> A `session_start()` hívásnak minden kimenet (HTML, `echo`, whitespace) előtt kell történnie, ellenkező esetben „Headers already sent” hiba keletkezik.

Adatok tárolása és olvasása

A munkamenetadatokat a `$_SESSION` szuperglobális tömbben tároljuk és abból olvassuk ki.

```php
session_start();

// Adatok tárolása
$_SESSION['felhasznalonev'] = 'kovacs_peter';
$_SESSION['felhasznalo_id'] = 42;
$_SESSION['admin'] = false;

// Adatok olvasása
if (isset($_SESSION['felhasznalonev'])) {
    echo "Bejelentkezett: "
        . htmlspecialchars(
            $_SESSION['felhasznalonev'],
            ENT_QUOTES,
            'UTF-8'
        );
}
```

5.15. kódpélda. *Munkamenetadatok tárolása és olvasása*

Adatok törlése

A munkamenetadatok több szinten törölhetők: egyes változók, a teljes munkamenet-adathalmaz, vagy maga a munkamenet is megszüntethető.

```php
// Egy konkrét változó törlése
unset($_SESSION['felhasznalo_id']);

// Teljes kijelentkezés:
// adatok törlése + munkamenet megszüntetése
session_unset();
session_destroy();

// A session süti törlése a böngészőből is
if (isset($_COOKIE[session_name()])) {
    setcookie(session_name(), '', time() - 3600, '/');
}
```

5.16. kódpélda. *Munkamenetadatok törlése*

Megjegyzés

A `session_destroy()` csak a szerveroldali munkamenetet törli. A böngészőben tárolt session sütit külön kell eltávolítani, ezért kijelentkezéskor mindkét lépést el kell végezni.

5.3.3. Hasznos munkamenet-függvények

A PHP számos segédfüggvényt biztosít a munkamenetek kezelésére:

- `session_name()`: a munkamenet nevének lekérdezése vagy beállítása;
- `session_id()`: a munkamenet-azonosító lekérdezése vagy beállítása;
- `session_regenerate_id()`: új munkamenet-azonosító generálása (*session fixation* elleni védelem);
- `session_status()`: a munkamenet aktuális állapotának lekérdezése;
- `session_reset()`: a munkamenetadatok visszaállítása az eredeti állapotba;
- `session_set_cookie_params()`: a session süti paramétereinek beállítása.

```php
// session_name() csak session_start() ELŐTT hívható
session_name('AlkalmazasSession');
session_start();

echo "Session név: " . session_name() . "<br>";
echo "Session ID: " . session_id() . "<br>";

// Session fixation védelem:
// új azonosító generálása
session_regenerate_id(true);

// Session állapot ellenőrzése
if (session_status() === PHP_SESSION_NONE) {
    session_start();
}
```

5.17. kódpélda. *Hasznos munkamenet-függvények*

5.3.4. Biztonsági szempontok

A munkamenetek kezelése során több biztonsági kockázatot is figyelembe kell venni. A leggyakoribb támadási módok a munkamenet-eltérítés (*session hijacking*) és a munkamenet-rögzítés (*session fixation*).

Session hijacking és session fixation

Session hijacking esetén a támadó megszerzi egy bejelentkezett felhasználó munkamenet-azonosítóját, és ezáltal hozzáfér a felhasználó fiókjához. A támadás tipikus módjai közé tartozik a hálózati forgalom lehallgatása (titkosítatlan HTTP-kapcsolat esetén) és az XSS-támadás.

Session fixation esetén a támadó előre meghatároz egy munkamenet-azonosítót, majd ráveszi a felhasználót, hogy ezzel az azonosítóval jelentkezzen be. Például elküldheti a felhasználónak a következő linket: `http://peldabank.com/login.php?PHPSESSID=abc123`. Ha a felhasználó bejelentkezik, a támadó ugyanazzal a munkamenet-azonosítóval hozzáférhet a munkamenethez. A PHP alapértelmezett beállítása (`session.use_only_cookies = 1`) az URL-ben kapott azonosítót figyelmen kívül hagyja, a támadó azonban más úton, például egy sütivel is rögzítheti az azonosítót.

Mindkét támadás ellen hatékony védelmet nyújt a munkamenet-azonosító újragenerálása sikeres bejelentkezés után:

```php
session_start([
    'use_strict_mode' => true,      // Session fixation
        elleni védelem
    'use_only_cookies' => true,     // Session ID ne URL-ben
        legyen
    'cookie_secure' => true,        // Csak HTTPS
    'cookie_httponly' => true,      // XSS elleni védelem
    'cookie_samesite' => 'Strict'   // CSRF elleni védelem
]);

// Sikeres bejelentkezés után
if ($validLogin) {
    // Új session ID generálása (régi törlésével)
    session_regenerate_id(true);

    $_SESSION['felhasznalo_id'] = $felhasznaloId;
    $_SESSION['bejelentkezve']  = true;
}
```

5.18. kódpélda. *A munkamenet-azonosító újragenerálása bejelentkezés után*

A `session_regenerate_id(true)` új munkamenet-azonosítót generál, és a `true` paraméter hatására a régi munkamenetfájl törlődik. Ez megakadályozza, hogy egy esetlegesen megszerzett régi azonosítóval hozzá lehessen férni a bejelentkezett munkamenethez.

Munkamenet lejáratának kezelése

A munkamenet lejárata biztosítja, hogy az inaktív munkamenetek automatikusan lezáródjanak. A PHP szemétgyűjtése (*garbage collection*) nem determinisztikus, ezért az inaktivitás ellenőrzését gyakran alkalmazásoldalon is megvalósítjuk:

```php
session_start();

// Inaktivitási időkorlát: 30 perc
$timeout = 1800;

if (isset($_SESSION['last_activity']) &&
    time() - $_SESSION['last_activity'] > $timeout) {

    session_unset();
    session_destroy();

    header('Location: login.php?timeout=1');
    exit;
}

$_SESSION['last_activity'] = time();
```

5.19. kódpélda. *Munkamenet lejáratának kezelése*

Minden sikeres kérés után frissítjük az utolsó aktivitás időpontját. Ha az eltelt idő meghaladja a megadott határértéket, töröljük a munkamenetet, majd a felhasználót visszairányítjuk a bejelentkezési oldalra, ahol a `timeout=1` paraméter alapján tájékoztató üzenet jeleníthető meg.

5.3.5. Bevásárlókosár kezelése munkamenettel

A következő példa bemutatja a munkamenetadatok tárolását, módosítását és törlését egy egyszerű bevásárlókosár megvalósításán keresztül.

```
<?php
session_start();

if (!isset($_SESSION['cart'])) {
    $_SESSION['cart'] = [];
}

// Termék hozzáadása
if (isset($_POST['add_to_cart'])) {
    $id  = (int)($_POST['product_id'] ?? 0);
    $qty = max(1, (int)($_POST['quantity'] ?? 1));

    if ($id > 0) {
        $_SESSION['cart'][$id] = ($_SESSION['cart'][$id] ??
            0) + $qty;
    }
}

// Termék eltávolítása
if (isset($_POST['remove_from_cart'])) {
    $id = (int)($_POST['product_id'] ?? 0);
    unset($_SESSION['cart'][$id]);
}
?>

<!-- Termék hozzáadása -->
<form method="POST">
    <input type="hidden" name="product_id" value="1">
    <input type="number" name="quantity" value="1" min="1">
    <button type="submit" name="add_to_cart">Kosárba</button>
</form>

<!-- Kosár tartalma -->
<?php
$total = 0;

foreach ($_SESSION['cart'] as $id => $qty) {
    $price    = 10; // valós alkalmazásban adatbázisból
        érkezik
    $subtotal = $price * $qty;
```

```
    $total    += $subtotal;

    echo "Termék #" . (int)$id . ": " . (int)$qty . " db - "
        . number_format($subtotal, 2) . " RON";

    echo '<form method="POST" style="display:inline">';
    echo '<input type="hidden" name="product_id" value="' .
        (int)$id . '">';
    echo '<button type="submit"
        name="remove_from_cart">Törlés</button>';
    echo '</form><br>';
}

echo "Összesen: " . number_format($total, 2) . " RON";
?>
```

5.20. kódpélda. *Bevásárlókosár kezelése munkamenettel*

A kosár tartalmát a `$_SESSION['cart']` tömbben tároljuk, ahol a kulcsok a termékazonosítók, az értékek pedig a mennyiségek. Az adatok automatikusan megmaradnak az oldalak közötti navigáció során, így nincs szükség sütiben tárolt JSON-adatokra vagy adatbázis használatára.

5.3.6. Flash üzenetek

A flash üzenet egy munkamenetben tárolt, egyszer megjelenő visszajelzés, amelyet tipikusan átirányítás után olvasunk ki és törlünk. A PRG-mintával kombinálva lehetővé teszi, hogy az űrlapfeldolgozás eredményét az átirányított oldalon jelenítsük meg.

```
session_start();
// ... feldolgozás ...
$_SESSION['flash'] = 'A hallgató sikeresen mentve.';
header('Location: index.php');
exit;
```

5.21. kódpélda. *Flash üzenet beállítása (save.php)*

```
session_start();
if (isset($_SESSION['flash'])) {
    echo '<div class="alert">'
        . htmlspecialchars($_SESSION['flash'])
        . '</div>';
    unset($_SESSION['flash']);
}
```

5.22. kódpélda. *Flash üzenet megjelenítése (index.php)*

Az `unset()` gondoskodik arról, hogy az üzenet csak egyszer jelenjen meg: újratöltés esetén már nem olvasható ki.

5.4. Gyakorlófeladatok

1. Készítsünk egy szkriptet, amely süti segítségével megjegyzi a felhasználó által választott színsémát:
   - gomb megnyomására váltson világos és sötét séma között;
   - az aktuális választást olvassuk ki a sütiből (30 napos lejárat);
   - ha a süti még nem létezik, alapértelmezetten a világos séma jelenjen meg.
2. Valósítsuk meg a „Legutóbb megtekintett termékek” funkciót süti segítségével:
   - minden oldallátogatáskor tároljuk el az aktuális termék azonosítóját egy JSON-kódolt sütiben;
   - tartsuk meg az utolsó 5 megtekintett terméket (7 napos lejárattal);
   - ismétlődő azonosítók ne kerüljenek be a listába;
   - jelenítsük meg a listát az oldalon.
3. Hozzunk létre egy sütiértesítő sávot (cookie banner):
   - az oldal első betöltésekor jelenjen meg egy sütiértesítő sáv, amely a sütik elfogadását kéri;
   - az elfogadó gombra kattintva rögzítsük a döntést egy 1 éves lejáratú sütiben;
   - ezt követően a sáv ne jelenjen meg újra.
4. Implementáljuk az „Emlékezz rám” funkciót egy bejelentkezési szkriptben. Ha a felhasználó bejelöli az opciót, hozzunk létre egy 30 napos sütit az alábbi biztonsági beállításokkal:
   - ne legyen elérhető JavaScriptből (`httponly`);
   - csak HTTPS-kapcsolaton keresztül küldje el a böngésző (`secure`);
   - legyen védett CSRF-támadások ellen (`samesite: 'Strict'`).

   Visszatérő látogatáskor a süti alapján automatikusan ismerjük fel a felhasználót.
5. Készítsünk egy bejelentkezési szkriptet munkamenet használatával. A felhasználókat egy tömbben tárolt szimulált adatok alapján ellenőrizzük (`password_verify()`), majd:

- sikeres bejelentkezés után generáljunk új munkamenet-azonosítót a *session fixation* elleni védelemként;
- tároljuk a felhasználó adatait a munkamenetben.

Valósítsuk meg a kijelentkezést is:

- töröljük a munkamenet adatait;
- szüntessük meg a munkamenetet;
- töröljük a session sütit is.

6. Valósítsunk meg egy munkamenet-lejárati mechanizmust:
   - az utolsó aktivitás időpontját tároljuk a munkamenetben;
   - minden oldalbetöltéskor ellenőrizzük, hogy az eltelt idő meghaladja-e a 10 percet;
   - ha igen, töröljük a munkamenetet, és irányítsuk át a felhasználót a bejelentkezési oldalra (`timeout=1` paraméterrel);
   - ha nem, frissítsük az aktivitás időpontját.
7. Készítsünk egy bevásárlólistát munkamenet segítségével:
   - a felhasználó tételeket adhasson hozzá (megnevezés és mennyiség megadásával);
   - a tételek egyenként törölhetők legyenek;
   - a lista maradjon meg a böngésző bezárásáig;
   - az üres listát és a legalább egy tételt tartalmazó listát jelenítsük meg eltérően;
   - tétel hozzáadása és törlése után alkalmazzuk a PRG-mintát; a visszajelzést flash üzenettel jelenítsük meg (pl. „Tétel hozzáadva.”).
8. Hozzunk létre egy háromlépcsős regisztrációs folyamatot munkamenet segítségével. Az egyes lépések adatait tároljuk munkamenetben, és tegyük lehetővé a visszalépést az előző lépésre az adatok elvesztése nélkül:
   - 1. lépés: személyes adatok (név, e-mail-cím, telefonszám);
   - 2. lépés: lakcím (város, utca, házszám);
   - 3. lépés: fiókadatok (felhasználónév, jelszó).

   A harmadik lépés sikeres beküldésekor jelenítsük meg az összes megadott adatot.

6. FEJEZET

ADATBÁZIS-KEZELÉS

6.1. Az adatbázisok szerepe és típusai

6.1.1. Az adatbázisok szerepe a webfejlesztésben

A modern webalkalmazások egyik legfontosabb komponense az adatbázis, amely lehetővé teszi az adatok strukturált és hatékony tárolását, kezelését és visszakeresését. Az adatbázisok használata számos előnnyel jár a webfejlesztésben:

- **Adatok perzisztens tárolása:** az adatok a program futásának befejezése után is megmaradnak.
- **Hatékony adatkezelés:** nagy mennyiségű adat gyors elérése és feldolgozása.
- **Adatintegritás:** az adatok konzisztenciájának és helyességének biztosítása.
- **Konkurens hozzáférés:** több felhasználó egyidejű kiszolgálása.
- **Adatbiztonság:** jogosultságkezelés és adatvédelem biztosítása.
- **Skálázhatóság:** az alkalmazás növekedésével az adatbázis is skálázható.

6.1.2. Adatbázistípusok

Az adatbázisokat két fő kategóriába sorolhatjuk: *relációs* és *NoSQL-adatbázisok*.

Relációs adatbázisok

A relációs adatbázisok táblákban tárolják az adatokat, ahol a táblák között kapcsolatok (relációk) definiálhatók (Elmasri–Navathe 2016). Főbb jellemzőik:

- strukturált adattárolás táblákban;
- SQL használata;

- ACID-tulajdonságok (Atomicity, Consistency, Isolation, Durability);
- szigorú séma;
- példák: MySQL, PostgreSQL, Oracle, SQL Server.

NoSQL-adatbázisok

A NoSQL-adatbázisok nem táblákban tárolják az adatokat, és általában rugalmasabb sémával rendelkeznek. Főbb típusaik:

- **Dokumentumalapú tárolás:** JSON-szerű dokumentumok (pl. MongoDB);
- **Kulcs–érték alapú tárolás:** egyszerű kulcs–érték párok (pl. Redis);
- **Oszlopalapú tárolás:** nagy mennyiségű adat hatékony kezelése (pl. Cassandra);
- **Gráfalapú tárolás:** kapcsolatok és hálózatok modellezése (pl. Neo4j).

Ebben a jegyzetben kizárólag a relációs adatbázisokkal foglalkozunk, mivel ezek a PHP-alapú webalkalmazások fejlesztése során a legelterjedtebbek és legalapvetőbbek.

6.1.3. Adatbázis-alapú alkalmazások működése

Az adatbázis-alapú webalkalmazások tipikusan háromrétegű architektúrát követnek (lásd a 6.1. ábrát):

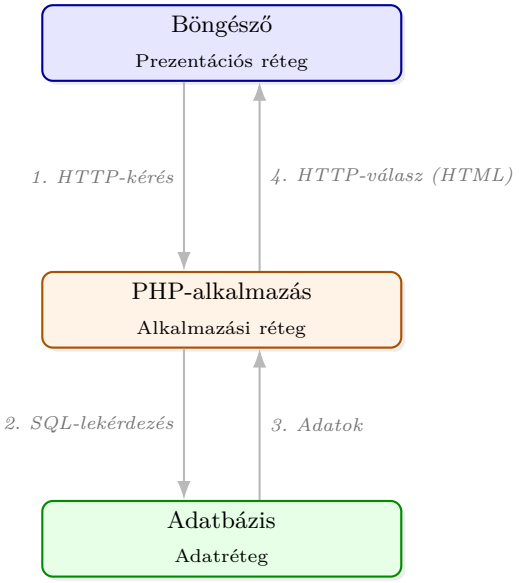
A működés főbb lépései:

1. A böngésző HTTP-kérést küld a szervernek (pl. egy űrlap elküldése után).
2. A PHP-alkalmazás SQL-lekérdezést küld az adatbázisnak.
3. Az adatbázis visszaadja a kért adatokat.
4. A PHP-alkalmazás HTML-formátumban állítja elő, majd küldi vissza a választ.

6.2. SQL- és MySQL-alapok

6.2.1. Az SQL-nyelv áttekintése

Az SQL (Structured Query Language) egy szabványosított nyelv, amely lehetővé teszi a relációs adatbázisokban tárolt adatok kezelését (Beaulieu



6.1. ábra. *Háromrétegű webalkalmazás architektúrája és adatfolyama*

2020). Az SQL-nyelvben a programozó deklaratív módon fogalmazza meg a lekérdezéseket: meghatározza, hogy milyen adatokat szeretne lekérdezni, de nem kell részletesen leírnia, hogy az adatbázis-kezelő rendszer hogyan hajtsa végre a műveletet.

SQL-alapfogalmak

- **Tábla (Table):** az adatok tárolására szolgáló alapegység, sorokból és oszlopokból áll;
- **Rekord (Record):** egy tábla egy sora, amely összetartozó adatokat tartalmaz;
- **Mező (Field):** egy tábla egy oszlopa, amely egy adott típusú adatot tárol;
- **Kulcs (Key):** egyedi azonosító, amely egy vagy több mezőből állhat. Fajtái:
  - **Elsődleges kulcs (Primary Key):** egyértelműen azonosít egy rekordot;
  - **Idegen kulcs (Foreign Key):** egy másik tábla elsődleges kulcsára hivatkozik;
- **Index:** az adatok keresését és rendezését gyorsítja.

SQL-utasítások kategóriái

Az SQL-utasításokat négy fő kategóriába sorolhatjuk:

1. **DDL (Data Definition Language).** Adatbázis-objektumok létrehozása és módosítása:
   - CREATE: objektumok létrehozása;
   - ALTER: objektumok módosítása;
   - DROP: objektumok törlése;
   - TRUNCATE: a tábla tartalmának törlése.
2. **DML (Data Manipulation Language).** Adatok kezelése:
   - SELECT: adatok lekérdezése;
   - INSERT: új adatok beszúrása;
   - UPDATE: meglévő adatok módosítása;
   - DELETE: adatok törlése.
3. **DCL (Data Control Language).** Jogosultságok kezelése:
   - GRANT: jogosultságok adása;
   - REVOKE: jogosultságok visszavonása.
4. **TCL (Transaction Control Language).** Tranzakciók kezelése:
   - COMMIT: tranzakció véglegesítése;
   - ROLLBACK: tranzakció visszavonása;
   - SAVEPOINT: visszaállítási pont létrehozása.

6.2.2. MySQL-bevezetés

A MySQL egy nyílt forráskódú relációs adatbázis-kezelő rendszer, amely az SQL-nyelvet használja. A MySQL különösen népszerű a webalkalmazások fejlesztése során, mivel:

- gyors és megbízható;
- könnyen használható;
- ingyenesen elérhető (Community Edition);
- kiváló PHP-támogatással rendelkezik;
- nagy közösséggel és kiterjedt dokumentációval rendelkezik.

Megjegyzés

Fejlesztői és éles környezetben eltérő adatbázis-felhasználót érdemes használni. A `root` az adminisztrátori fiók, amely minden műveletet elvégezhet. Fejlesztés közben, saját gépen ezt a legkényelmesebb használni (XAMPP esetén a `root` jelszava alapértelmezetten üres). Éles környezetben azonban a `root` fiókot kizárólag adminisztrációs feladatokra szabad használni: minden alkalmazáshoz külön, csak a szükséges jogosultságokkal rendelkező felhasználót hozunk létre. A könyv és a hozzá tartozó GitHub-tároló kódpéldái az egyszerűség kedvéért a fejlesztői `root` felhasználót használják. Éles rendszerben ezek helyére az alkalmazás saját, korlátozott jogú felhasználója kerül.

A MySQL több tárolómotort (táblatípust) támogat, amelyek képességei jelentősen eltérnek. Ebben a könyvben az alapértelmezett `InnoDB` motort használjuk, amelynek a könyv szempontjából fontos jellemzői: támogatja a tranzakciókat, kikényszeríti az idegen kulcsokra vonatkozó megszorításokat, és sorszintű zárolást alkalmaz, így párhuzamos hozzáférés esetén is megbízhatóan működik.

6.2.3. MySQL-adattípusok

A webfejlesztésben leggyakrabban használt MySQL-adattípusokat a 6.1. táblázat foglalja össze:

6.1. táblázat. *A legfontosabb MySQL-adattípusok*

Típus	Példa	Leírás
`INT`	`INT`	Egész szám
`DECIMAL`	`DECIMAL(10,2)`	Pontos tizedes szám (pl. árak)
`FLOAT`	`FLOAT`	Közelítő lebegőpontos szám
`VARCHAR`	`VARCHAR(255)`	Változó hosszúságú szöveg (a maximális méret a sorlimittől és a karakterkódolástól függ)
`TEXT`	`TEXT`	Hosszú szöveg
`TINYINT(1)`	`TINYINT(1)`	Logikai érték (0/1)
`DATE`	`DATE`	Dátum (YYYY-MM-DD)
`DATETIME`	`DATETIME`	Dátum és idő, időzónától független
`TIMESTAMP`	`TIMESTAMP`	Időbélyeg, UTC-ben tárolva
`JSON`	`JSON`	JSON-dokumentum tárolása (MySQL 5.7+)

A `DATETIME` és a `TIMESTAMP` közötti fő különbség az időzóna-kezelésben

rejlik: a `TIMESTAMP`-értékeket a szerver UTC-ben tárolja, és az aktuális időzónának megfelelően konvertálva adja vissza, míg a `DATETIME` az értéket változtatás nélkül tárolja. Részletesebb adattípuslista a MySQL dokumentációjában található.[1]

6.3. Adatbázis-kezelés PHP-ben

6.3.1. Mi a PDO?

A PDO (PHP Data Objects) egy egységes, objektumorientált interfész különböző adatbázis-kezelő rendszerek eléréséhez. A PDO használata több előnnyel jár:

- **Adatbázis-függetlenség:** ugyanaz a kód működik MySQL, PostgreSQL, SQLite, Oracle stb. adatbázisokkal;
- **Biztonság:** beépített előkészített utasítás (prepared statement) támogatás SQL-injection-támadások ellen;
- **Konzisztens API:** egységes metódusnevek és működés minden támogatott adatbázisnál;
- **Objektumorientált megközelítés:** modern PHP-paradigmákhoz illeszkedik;
- **Rugalmas lekérési módok:** többféle módon nyerhetjük ki az adatokat;
- **Hibakezelés:** kivételalapú (exception-based) hibakezelés.

Megjegyzés

A PDO a PHP 5.1 óta elérhető, és a modern PHP-alkalmazások egyik standard eszköze az adatbázis-műveletekhez. A PHP-ben létezik a `mysqli` kiterjesztés is, amely kizárólag MySQL/MariaDB-adatbázisokhoz használható. A PDO ezzel szemben több adatbázistípust támogató, egységes absztrakciós réteget biztosít. Ebben a jegyzetben kizárólag PDO-t használunk.

6.3.2. Kapcsolat létrehozása

A PDO-kapcsolat létrehozása a `PDO`-osztály példányosításával történik.

[1] https://dev.mysql.com/doc/refman/8.4/en/data-types.html

Szintaxis

```
PDO::__construct(
    string $dsn,
    ?string $username = null,
    ?string $password = null,
    ?array $options = null
)
```

Paraméterek:

- `$dsn`: adatkapcsolati leíró (Data Source Name), amely tartalmazza az adatbázis típusát, a gazdagépet, az adatbázis nevét és opcionálisan a karakterkódolást;
- `$username` (opcionális): adatbázis-felhasználónév;
- `$password` (opcionális): adatbázisjelszó;
- `$options` (opcionális): kapcsolati beállításokat tartalmazó tömb (pl. hibakezelési mód, lekérési mód).

Visszatérési érték: sikeres kapcsolódás esetén egy `PDO`-objektumot ad vissza. Hiba esetén (ha a hibakezelési mód kivételre van állítva) `PDOException` kivételt dob.

DSN felépítése MySQL esetén

A DSN egy speciális formátumú sztring, amely tartalmazza az adatbázis-kapcsolat konfigurációs paramétereit:

```
mysql:host=hostname;dbname=database;charset=utf8mb4;port=3306
```

Megjegyzés

A `charset=utf8mb4` használata kifejezetten ajánlott. Az `utf8mb4` a teljes Unicode-karakterkészletet támogatja, míg a MySQL `utf8` karakterkódolása csak az UTF-8 egy részhalmazát (legfeljebb 3 bájtos karaktereket) kezeli, ezért bizonyos karakterek, például az emojik tárolására nem alkalmas.

Ajánlott konfiguráció

Az `$options` tömbben három fontos PDO-beállítást érdemes minden kapcsolatnál megadni:

- **ATTR_ERRMODE:** hibakezelési mód. Ajánlott a `PDO::ERRMODE_EXCEPTION` beállítás, amely hiba esetén kivételt dob;
- **ATTR_DEFAULT_FETCH_MODE:** alapértelmezett lekérési mód. Ajánlott a `PDO::FETCH_ASSOC` érték, amely asszociatív tömbként adja vissza az eredményt;
- **ATTR_EMULATE_PREPARES:** `false` értékre állítva a PDO natív előkészített utasításokat használ, amelyek jobb típuskezelést és helyesebb SQL-végrehajtási viselkedést biztosítanak.

```php
<?php

try {
    $pdo = new PDO(
        dsn: "mysql:host=localhost;dbname=mydb;charset=
            utf8mb4",
        username: "root",
        password: "",
        options: [
            PDO::ATTR_ERRMODE            => PDO::
                ERRMODE_EXCEPTION,
            PDO::ATTR_DEFAULT_FETCH_MODE => PDO::FETCH_ASSOC,
            PDO::ATTR_EMULATE_PREPARES   => false,
        ]
    );
} catch (PDOException $e) {
    error_log("Kapcsolódási hiba: " . $e->getMessage());
    die("Adatbázis-kapcsolódási hiba történt.");
}
```

6.1. kódpélda. *PDO-kapcsolat létrehozása ajánlott konfigurációval*

Megjegyzés

A fenti példában a felhasználónév és a jelszó közvetlenül a forráskódban szerepel. Éles projektben ez biztonsági kockázatot jelent: a hitelesítő adatok bekerülnek a verziókövetőbe, és minden fejlesztő számára láthatóvá válnak. A megoldás a `.env` fájl és a környezeti változók használata, amelyekről a 8.6.2. alfejezetben lesz részletesen szó.

Kivételalapú hibakezelés

Az `ERRMODE_EXCEPTION` beállítással a PDO minden adatbázishiba esetén `PDOException` kivételt dob, amelyet try-catch blokkal kezelhetünk. Integritási

megszorítás (pl. unique constraint, foreign key) megsértésekor az SQLSTATE-hibakód 23000, amely adatbázis-integritási hibát jelöl.

```php
try {
    $stmt = $pdo->prepare(
        "INSERT INTO users (email, username) VALUES (:email,
            :username)"
    );

    $stmt->execute([
        'email'    => 'kiss.janos@example.com',
        'username' => 'kiss_janos',
    ]);

    echo "Sikeres regisztráció!";
} catch (PDOException $e) {

    if ($e->getCode() == 23000) {
        echo "Integritási hiba történt (duplikált adat).";
    } else {
        error_log("Adatbázis-hiba: " . $e->getMessage());
        echo "Hiba történt a művelet során.";
    }
}
```

6.2. kódpélda. *Integritási megszorítás kezelése*

6.4. Előkészített utasítások és biztonság

Az *előkészített utasítások* (*prepared statement*) az adatbázis-biztonság egyik legfontosabb eszközei: segítenek megelőzni az *SQL-injection*-támadásokat, és hatékonyabb végrehajtást tesznek lehetővé ismételt lekérdezések esetén.

6.4.1. SQL injection

Az SQL-injection olyan biztonsági sebezhetőség, amely során a támadó rosszindulatú SQL-kódot juttat a lekérdezésbe a felhasználói bemenet manipulálásával.[2] Ez akkor fordulhat elő, ha a felhasználói inputot közvetlenül beillesztjük az SQL-utasításba sztringösszefűzéssel.

[2] https://owasp.org/www-community/attacks/SQL_Injection

Nézzük meg a sebezhető változatot:

```
$username = $_POST['username'];

// VESZÉLYES: felhasználói adat közvetlenül az SQL-ben!
$stmt = $pdo->query("SELECT * FROM users WHERE username =
    '$username'");
```

6.3. kódpélda. *SQL injectionre érzékeny kód*

Ha a támadó a következő bemenetet adja meg:

```
' OR '1'='1
```

Az SQL-utasítás így módosul:

```
SELECT * FROM users WHERE username = '' OR '1'='1'
```

Ez a feltétel mindig igaz, ezért a lekérdezés az összes felhasználót visszaadhatja, ami akár jogosulatlan hozzáféréshez is vezethet.

6.4.2. Előkészített utasítások

Az előkészített utasítások két lépésből állnak, ami biztosítja, hogy a paraméterek soha ne legyenek az SQL-kód részei (lásd a 6.2. ábrát):

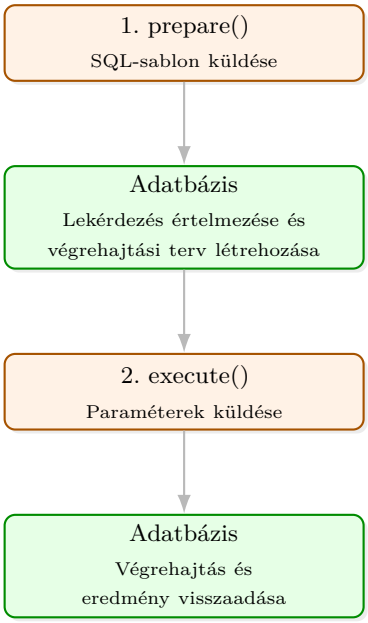
1. **Prepare:** az SQL-utasítás szerkezetét elküldjük az adatbázisnak *helyőrzőkkel* (*placeholder*);
2. **Execute:** a tényleges paraméterértékeket elküldjük, és a lekérdezés végrehajtódik.

Az előkészített utasítások esetén a paraméterek soha nem értelmeződnek SQL-kódként:

```
$username = $_POST['username'];

$stmt = $pdo->prepare("SELECT * FROM users WHERE username =
    :username");
$stmt->execute(['username' => $username]);
```

6.4. kódpélda. *Biztonságos lekérdezés előkészített utasítással*



6.2. ábra. *Az előkészített utasítás folyamata és fázisai*

6.4.3. bindParam és bindValue

A PDO két metódust kínál explicit paraméterkötésre. A `bindValue()` az aktuális értéket köti, míg a `bindParam()` a változó referenciáját használja, így az `execute()` hívásig bekövetkező változások is érvényesülnek. Emiatt a `bindParam()` használata iterációs (például ciklusos) környezetben könnyen félreértésekhez vezethet.

A legtöbb esetben az egyszerűbb és biztonságosabb megoldás az `execute()` közvetlen tömbargumentummal történő használata:

```
$stmt = $pdo->prepare(
    "INSERT INTO users (username, age) VALUES (:username,
        :age)"
);

$stmt->execute([
    'username' => 'kiss_janos',
    'age'      => 25
]);
```

6.5. kódpélda. *Paraméterek átadása az `execute()` metódussal*

A `bindParam()` és a `bindValue()` használata akkor indokolt, ha explicit adattípust kell megadni (például `PDO::PARAM_INT`), vagy ha ugyanazt az utasítást többször hajtjuk végre ciklusban, eltérő értékekkel.

6.5. CRUD-műveletek

A CRUD (Create, Read, Update, Delete) műveletek az adatbázis-kezelés alapvető műveletei. Minden alkalmazásban szükség van ezekre a funkciókra: új adatok létrehozására, meglévő adatok lekérdezésére, módosítására és törlésére. A PDO segítségével ezeket a műveleteket biztonságosan, előkészített utasítások használatával hajthatjuk végre.

6.5.1. SELECT – Adatok lekérdezése

A SELECT-művelet lehetővé teszi adatok lekérdezését az adatbázisból. Két fő módszer különböztethető meg: a `query()` metódus statikus lekérdezésekhez, valamint a `prepare()` + `execute()` kombináció dinamikus lekérdezésekhez.

Egyszerű lekérdezés

A `query()` metódust kizárólag olyan esetekben használjuk, amikor nincs felhasználói bemenet, és a lekérdezés teljesen statikus. Ez a módszer gyorsabb, de biztonsági kockázatot jelent, ha a lekérdezés felhasználói bemenettel összefűzött SQL-t tartalmaz.

```php
try {
    $stmt = $pdo->query("SELECT * FROM users ORDER BY
        username ASC");

    while ($row = $stmt->fetch()) {
        echo htmlspecialchars($row['username']) . "<br>";
    }

} catch (PDOException $e) {
    error_log("Lekérdezési hiba: " . $e->getMessage());
    die("Lekérdezési hiba.");
}
```

6.6. kódpélda. *Egyszerű SELECT `query()`-vel*

Megjegyzés

Ajánlott gyakorlat: alapértelmezés szerint mindig előkészített utasítást használjunk, kivéve, ha a lekérdezés teljesen statikus, és nem tartalmaz semmilyen felhasználói bemenetet vagy külső adatot.

Előkészített utasítás használata

A lekérdezésben helyőrzőket használunk, amelyekhez az értékeket az `execute()` metódus paramétereként adjuk meg:

```php
try {
    $stmt = $pdo->prepare("
        SELECT id, username, email, city
        FROM users
        WHERE age > :age AND city = :city
    ");

    $stmt->execute([
        'age'  => 18,
        'city' => 'Budapest'
    ]);

    while ($row = $stmt->fetch()) {
        echo htmlspecialchars($row['username']) . " - ";
        echo htmlspecialchars($row['city']) . "<br>";
    }

} catch (PDOException $e) {
    error_log("Lekérdezési hiba: " . $e->getMessage());
    die("Lekérdezési hiba.");
}
```

6.7. kódpélda. *SELECT előkészített utasítás használatával*

Névvel ellátott és pozícióalapú helyőrzők

A PDO két típusú helyőrzőt támogat: névvel ellátott (*named*) és pozícióalapú (*positional*) helyőrzőket.

A névvel ellátott helyőrzők esetén a paramétereket név alapján azonosítjuk (például `:city`), így az értékek hozzárendelése nem kötött sorrendhez. Ez olvashatóbb és karbantarthatóbb kódot eredményez:

```php
$stmt = $pdo->prepare("
    SELECT * FROM users
    WHERE age > :min_age
```

```
        AND age < :max_age
        AND city = :city
");

$stmt->execute([
    'min_age' => 18,
    'max_age' => 65,
    'city'    => 'Budapest'
]);
```

6.8. kódpélda. *Névvel ellátott helyőrzők használata*

Előnyei: olvashatóbb kód, a sorrend nem számít, és ugyanazt a paramétert többször is felhasználhatjuk.

A pozícióalapú helyőrzők esetén a paraméterek kérdőjelekkel (`?`) vannak jelölve, és az értékek hozzárendelése a sorrendjük alapján történik:

```
$stmt = $pdo->prepare("
    SELECT * FROM users
    WHERE age > ? AND age < ? AND city = ?
");
$stmt->execute([18, 65, 'Budapest']);
```

6.9. kódpélda. *Pozícióalapú helyőrzők használata*

Lekérési módok (fetch módok)

A PDO három fő lekérési módot kínál az eredmények feldolgozására: a `fetch()` egyetlen sort ad vissza, a `fetchAll()` az összes sort egy tömbbe tölti (nagyobb adatmennyiség esetén a magas memóriaigény miatt nem ajánlott), míg a `fetchColumn()` egyetlen oszlop értékét adja vissza. Ez utóbbi különösen hasznos például `COUNT(*)` lekérdezéseknél.

```
// fetch() - egyetlen sor, például ID alapú kereséshez
$stmt = $pdo->prepare("SELECT * FROM users WHERE id = :id");
$stmt->execute(['id' => 1]);
$user = $stmt->fetch(); // FETCH_ASSOC az alapértelmezett
    (ajánlott beállítás)
// fetchAll() - összes sor tömbként
$stmt = $pdo->query("SELECT id, username FROM users ORDER BY
    username ASC");
$users = $stmt->fetchAll();
// fetchColumn() - egyetlen oszlop értéke
$count = $pdo->query("SELECT COUNT(*) FROM
    users")->fetchColumn();
echo "Felhasználók száma: " . $count;
```

6.10. kódpélda. *Lekérési módok összefoglalása*

Megjegyzés

A `fetchAll()` az összes eredményt egyszerre a memóriába tölti, ezért nagyobb adatmennyiség esetén célszerű a `fetch()` metódust `while` ciklusban használni.

Objektumként való lekérés: `stdClass`

A `PDO::FETCH_OBJ` módot megadva a PDO az eredménysorokat nem tömbként, hanem `stdClass` objektumként adja vissza. A `stdClass` a PHP beépített, általános célú osztálya, amely dinamikusan bővíthető tulajdonságokkal rendelkezik. A lekérdezett oszlopnevek az objektum tulajdonságaiként érhetők el:

```php
$stmt = $pdo->query("SELECT id, username, email FROM users");
$users = $stmt->fetchAll(PDO::FETCH_OBJ);

foreach ($users as $user) {
    echo $user->username . " - " . $user->email . "<br>";
}
```

6.11. kódpélda. *Lekérés stdClass objektumként*

A `stdClass` objektumok nemcsak PDO-val keletkezhetnek: a `json_decode()` függvény is `stdClass` példányokat ad vissza alapértelmezetten, ha az eredményt nem tömbként kérjük. A mindennapi PHP-fejlesztésben általában az asszociatív tömb (`FETCH_ASSOC`) az elterjedtebb és egyszerűbben kezelhető megközelítés.

6.5.2. INSERT – Adatok beszúrása

Az INSERT-művelet új rekordok beszúrását teszi lehetővé az adatbázisba. Minden esetben előkészített utasításokat használunk a biztonság érdekében.

Egyszerű beszúrás

Az alapvető beszúrási művelet során előkészítjük az SQL-utasítást (`prepare`), majd végrehajtjuk a szükséges paraméterekkel (`execute`):

```php
try {
    $stmt = $pdo->prepare("
        INSERT INTO users (username, email, age)
        VALUES (:username, :email, :age)
    ");
    $stmt->execute([
        'username' => 'kiss_janos',
        'email'    => 'kiss.janos@example.com',
```

```
        'age'      => 25
    ]);
    $lastId = (int) $pdo->lastInsertId(); // PDO stringként
        adja vissza, ezért cast szükséges
    echo "Új felhasználó azonosítója: " . $lastId;
} catch (PDOException $e) {
    error_log("Beszúrási hiba: " . $e->getMessage());
    die("Beszúrási hiba.");
}
```

6.12. kódpélda. *Egyszerű INSERT*

Űrlapadatok beszúrása ellenőrzéssel

Valós alkalmazásokban az űrlapadatokat mindig validálni és szükség esetén szűrni kell a beszúrás előtt. A `filter_var()` függvény segítségével ellenőrizhetjük az adatok helyességét:

```
if ($_SERVER['REQUEST_METHOD'] === 'POST') {
    try {
        $username = trim($_POST['username'] ?? '');
        $email = filter_var($_POST['email'] ?? '',
            FILTER_VALIDATE_EMAIL);
        $age = filter_var($_POST['age'] ?? 0,
            FILTER_VALIDATE_INT);

        if (empty($username)) {
            throw new Exception("A felhasználónév
                kötelező!");
        }

        if ($email === false) {
            throw new Exception("Érvénytelen e-mail cím!");
        }

        if ($age === false || $age < 18 || $age > 120) {
            throw new Exception("Az életkornak 18 és 120
                között kell lennie!");
        }

        $stmt = $pdo->prepare("
            INSERT INTO users (username, email, age,
                created_at)
            VALUES (:username, :email, :age, NOW())
        ");
```

```
        $stmt->execute([
            'username' => $username,
            'email'    => $email,
            'age'      => $age
        ]);

        $userId = (int) $pdo->lastInsertId();
        echo "Sikeres regisztráció! Azonosító: " . $userId;

    } catch (PDOException $e) {
        if ((int)$e->getCode() === 23000) {
            echo "Ez az e-mail cím vagy felhasználónév már
                regisztrálva van!";
        } else {
            error_log("Beszúrási hiba: " . $e->getMessage());
            echo "Hiba történt a regisztráció során.";
        }
    } catch (Exception $e) {
        echo "Hiba: " . htmlspecialchars($e->getMessage());
    }
}
```

6.13. kódpélda. *Űrlapadatok beszúrása*

6.5.3. UPDATE – Adatok módosítása

Az UPDATE-művelet meglévő rekordok módosítását teszi lehetővé. A WHERE-feltétel megadása elengedhetetlen, mivel ennek hiányában az összes rekord módosításra kerülne:

```
try {
    $stmt = $pdo->prepare("
        UPDATE users
        SET email = :email, age = :age
        WHERE id = :id
    ");

    $stmt->execute([
        'email' => 'kovacs.peter@example.com',
        'age'   => 26,
        'id'    => 1
    ]);

    $affectedRows = $stmt->rowCount();

    if ($affectedRows > 0) {
```

```
        echo "Sikeres módosítás!";
    } else {
        echo "Nincs módosítható rekord ezzel az
            azonosítóval.";
    }

} catch (PDOException $e) {
    error_log("Módosítási hiba: " . $e->getMessage());
    die("Módosítási hiba.");
}
```

6.14. kódpélda. *Egyszerű UPDATE*

6.5.4. DELETE – Adatok törlése

A DELETE-művelet rekordok eltávolítását teszi lehetővé. Figyelnünk kell az idegen kulcsokra vonatkozó integritási megszorításokra, amelyek megakadályozhatják a törlést, illetve cascade szabályok esetén további rekordok automatikus törlését is eredményezhetik:

```
try {
    $stmt = $pdo->prepare("DELETE FROM users WHERE id =
        :id");
    $stmt->execute(['id' => 1]);

    $affectedRows = $stmt->rowCount();

    if ($affectedRows > 0) {
        echo "Sikeres törlés!";
    } else {
        echo "Nem történt törlés (nincs ilyen azonosító).";
    }

} catch (PDOException $e) {
    if ((int)$e->getCode() === 23000) {
        echo "Nem törölhető, mert más rekordok hivatkoznak
            rá!";
    } else {
        error_log("Törlési hiba: " . $e->getMessage());
        echo "Törlési hiba.";
    }
}
```

6.15. kódpélda. *Egyszerű DELETE*

6.5.5. CRUD-műveletek osztályban

A többrétegű architektúrát használó alkalmazásokban ajánlott a CRUD-műveleteket egy Repository osztályban összefogni. Ez a Repository minta (*Repository pattern*) (Fowler 2002) elkülöníti az adathozzáférési logikát az alkalmazás többi részétől, így a kód karbantarthatóbbá és újrafelhasználhatóbbá válik. A többi komponens közvetlen adatbázis-kezelés helyett a Repository osztály metódusait használja:

```php
class UserRepository {
    public function __construct(
        private PDO $pdo
    ) {}

    public function create(string $username, string $email,
        int $age): int
    {
        $stmt = $this->pdo->prepare("
            INSERT INTO users (username, email, age,
                created_at)
            VALUES (:username, :email, :age, NOW())
        ");

        $stmt->execute([
            'username' => $username,
            'email'    => $email,
            'age'      => $age,
        ]);

        return (int) $this->pdo->lastInsertId();
    }

    public function findById(int $id): ?array
    {
        $stmt = $this->pdo->prepare("SELECT * FROM users
            WHERE id = :id");
        $stmt->execute(['id' => $id]);

        $user = $stmt->fetch();
        return $user ?: null;
    }

    // findAll(), update(), delete() ugyanezt a mintát
        követik
}
```

6.16. kódpélda. *UserRepository osztály*

6.6. Tranzakciókezelés

A tranzakciók lehetővé teszik, hogy több adatbázis-műveletet egyetlen egységként kezeljünk. Ez biztosítja, hogy vagy az összes művelet végrehajtódik, vagy egyik sem, így megakadályozva a részleges módosításokat, amelyek inkonzisztens adatállapotot eredményeznének.

6.6.1. Tranzakciók alapjai

A tranzakciók négy alapvető tulajdonsággal rendelkeznek, amelyeket az ACID betűszó jelöl (Elmasri–Navathe 2016):

- **Atomosság (Atomicity):** a tranzakció oszthatatlan egység, amely esetén vagy minden művelet végrehajtódik, vagy egyik sem;
- **Konzisztencia (Consistency):** a tranzakció biztosítja, hogy az adatbázis minden művelet után érvényes, szabályoknak megfelelő állapotban maradjon;
- **Izoláció (Isolation):** a párhuzamosan futó tranzakciók egymástól függetlenül hajtódnak végre, és nem befolyásolják egymást;
- **Tartósság (Durability):** a véglegesített tranzakció eredménye tartósan megmarad rendszerhiba esetén is.

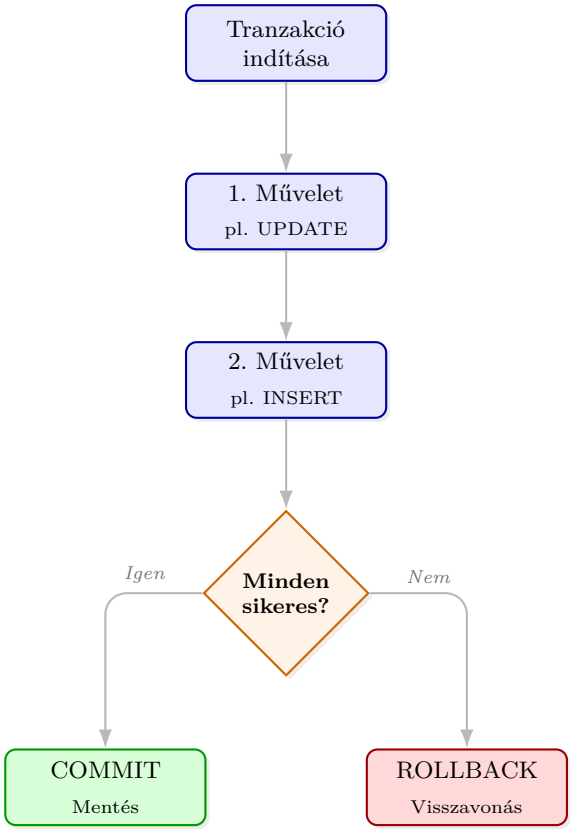
Ha több tranzakció egyszerre fut, az adatbázis-kezelő gondoskodik arról, hogy azok ne zavarják egymást. Ennek mértékét az úgynevezett izolációs szintek (*isolation levels*) szabályozzák. A MySQL alapértelmezett beállítása (`REPEATABLE READ`) azt biztosítja, hogy egy tranzakció a futása alatt következetes, más tranzakciók közbeni módosításaitól független adatokat lásson. Ez a legtöbb webalkalmazás esetében megfelelő, és a könyv példáiban nem is változtatunk rajta.

6.6.2. Tranzakciókezelés PHP-ben

A PDO három fő metódust biztosít a tranzakciók kezeléséhez:

- `beginTransaction()`: tranzakció indítása;
- `commit()`: tranzakció véglegesítése;
- `rollBack()`: tranzakció visszavonása.

A tranzakció használata a következő lépésekből áll: elindítjuk a tranzakciót, végrehajtjuk az adatbázis-műveleteket, majd sikeres végrehajtás esetén véglegesítjük, hiba esetén pedig visszavonjuk (lásd a 6.3. ábrát):



6.3. ábra. *Adatbázis-tranzakció folyamata*

```php
try {
    $pdo->beginTransaction();

    $stmt = $pdo->prepare("
        UPDATE accounts
        SET balance = balance - :amount
        WHERE id = :id
    ");
    $stmt->execute(['amount' => 100, 'id' => 1]);

    $stmt = $pdo->prepare("
        UPDATE accounts
        SET balance = balance + :amount
        WHERE id = :id
```

```
    ");
    $stmt->execute(['amount' => 100, 'id' => 2]);

    $pdo->commit();
    echo "Sikeres tranzakció!";

} catch (Exception $e) {
    if ($pdo->inTransaction()) {
        $pdo->rollBack();
    }
    error_log("Tranzakciós hiba: " . $e->getMessage());
    die("Tranzakciós hiba.");
}
```

6.17. kódpélda. *Egyszerű tranzakció*

6.6.3. Példa: banki átutalás

Az alábbi példa egy banki átutalást valósít meg, amely bemutatja a tranzakciók szerepét az adatkonzisztencia biztosításában. Az átutalás három műveletből áll: a küldő számla terheléséből, a fogadó számla jóváírásából, valamint a művelet naplózásából. Ha bármelyik művelet sikertelen, az egész tranzakció visszavonásra kerül:

```
function transferMoney(
    PDO $pdo,
    string $fromAccount,
    string $toAccount,
    float $amount
): bool {
    try {
        $pdo->beginTransaction();

        // 1. Pénz levonása a küldő számláról
        $stmt = $pdo->prepare("
            UPDATE accounts
            SET balance = balance - :amount
            WHERE account_number = :account
              AND balance >= :min_balance
        ");

        $stmt->execute([
            'amount'      => $amount,
            'account'     => $fromAccount,
            'min_balance' => $amount,
        ]);
```

```
        if ($stmt->rowCount() === 0) {
            throw new Exception("Nincs elegendő fedezet vagy
                érvénytelen számla!");
        }

        // 2. Pénz jóváírása a fogadó számlán
        $stmt = $pdo->prepare("
            UPDATE accounts
            SET balance = balance + :amount
            WHERE account_number = :account
        ");

        $stmt->execute([
            'amount'  => $amount,
            'account' => $toAccount
        ]);

        if ($stmt->rowCount() === 0) {
            throw new Exception("Fogadó számla nem
                található!");
        }

        // 3. Tranzakció naplózása
        $stmt = $pdo->prepare("
            INSERT INTO transactions (from_account,
                to_account, amount, created_at)
            VALUES (:from, :to, :amount, NOW())
        ");

        $stmt->execute([
            'from'   => $fromAccount,
            'to'     => $toAccount,
            'amount' => $amount
        ]);

        $pdo->commit();
        return true;

    } catch (Exception $e) {
        if ($pdo->inTransaction()) {
            $pdo->rollBack();
        }

        error_log("Átutalási hiba: " . $e->getMessage());
        return false;
```

```
    }
}

// Használat
$success = transferMoney($pdo, '12345', '67890', 1000.00);

if ($success) {
    echo "Sikeres átutalás!";
} else {
    echo "Az átutalás sikertelen.";
}
```

6.18. kódpélda. *Banki átutalás tranzakcióval*

Megjegyzés

Tranzakciók helyes használata:

- mindig try-catch blokkot használjunk;
- rollback előtt ellenőrizzük, hogy aktív tranzakció fut-e (`inTransaction()`);
- minimalizáljuk a tranzakciók időtartamát a zárolások csökkentése érdekében;
- ne használjunk DDL-utasításokat (CREATE, ALTER, DROP) tranzakción belül, mivel ezek adatbázistól függően automatikus commitot okozhatnak.

6.7. Teljesítményoptimalizálás

Az adatbázis teljesítményének optimalizálása kritikus fontosságú a webalkalmazások gyorsasága szempontjából. Az alábbiakban a legfontosabb technikákat tekintjük át.

6.7.1. Indexek használata

Az indexek az adatbázis-teljesítmény egyik legfontosabb eszközei. Egy index olyan, mint egy könyv tartalomjegyzéke: lehetővé teszi, hogy gyorsan megtaláljuk a keresett adatot anélkül, hogy végig kellene olvasnunk az egész táblát.

Mikor használjunk indexet?

Az indexek használata megfontolást igényel, mivel előnyökkel és hátrányokkal is jár.

Indexet érdemes létrehozni:

- `WHERE`-feltételekben gyakran használt oszlopokon;
- `JOIN`-műveletekben használt oszlopokon;
- `ORDER BY` rendezésekben használt oszlopokon;
- idegen kulcs oszlopokon (az InnoDB ezekre automatikusan indexet hoz létre, ha még nincs);
- egyedi értékű oszlopokon (pl. e-mail-cím, felhasználónév).

Index használata nem ajánlott:

- kis méretű táblákon, ahol a teljes bejárás gyorsabb lehet;
- gyakran módosított oszlopokon, mivel az index karbantartása lassítja az írási műveleteket;
- alacsony kardinalitású oszlopokon (pl. boolean típus, kevés lehetséges érték).

Index létrehozása

Az indexek létrehozása SQL-utasításokkal történik:

```sql
-- Egyszerű index egy oszlopon
CREATE INDEX idx_email ON users(email);

-- Összetett index több oszlopon
CREATE INDEX idx_city_age ON users(city, age);

-- UNIQUE index egyedi értékek biztosítására
CREATE UNIQUE INDEX idx_username ON users(username);
```

6.19. kódpélda. *Index létrehozása*

6.7.2. N+1 lekérdezés problémája

Az *N+1 lekérdezés* problémája az egyik leggyakoribb teljesítményprobléma-forrás. Akkor jelentkezik, amikor egy lista lekérdezése után minden elemhez külön lekérdezést végzünk, ami jelentősen növeli az adatbázis-műveletek számát.

A probléma bemutatása

Az alábbi kód látszólag egyszerű, de komoly teljesítményproblémát okoz:

```php
// 1 query: összes felhasználó lekérdezése
$stmt = $pdo->query("SELECT * FROM users LIMIT 10");
$users = $stmt->fetchAll(PDO::FETCH_ASSOC);

// N query: minden felhasználóhoz külön lekérdezés
foreach ($users as $user) {
    $stmt = $pdo->prepare("
        SELECT * FROM orders WHERE user_id = :user_id
    ");
    $stmt->execute(['user_id' => $user['id']]);
    $orders = $stmt->fetchAll(PDO::FETCH_ASSOC);

    echo "Felhasználó: " . $user['email'] .
         " - Rendelések: " . count($orders) . "<br>";
}
```

6.20. kódpélda. *N+1 lekérdezés problémája: hibás megközelítés*

Megoldás: JOIN használata

A megoldás a JOIN használata, amely egyetlen lekérdezésben kapcsolja össze a kapcsolódó adatokat:

```php
$stmt = $pdo->query("
    SELECT
        u.id AS user_id,
        u.email,
        o.id AS order_id,
        o.total,
        o.created_at
    FROM users u
    LEFT JOIN orders o ON u.id = o.user_id
    WHERE u.id <= 10
    ORDER BY u.id, o.created_at DESC
");

$results = $stmt->fetchAll(PDO::FETCH_ASSOC);

$users = [];
foreach ($results as $row) {
    $userId = $row['user_id'];

    if (!isset($users[$userId])) {
        $users[$userId] = [
```

```
            'email'  => $row['email'],
            'orders' => []
        ];
    }

    if ($row['order_id'] !== null) {
        $users[$userId]['orders'][] = [
            'id'    => $row['order_id'],
            'total' => $row['total']
        ];
    }
}

foreach ($users as $user) {
    echo "Felhasználó: " . $user['email'] .
         " - Rendelések: " . count($user['orders']) . "<br>";
}
```

6.21. kódpélda. *N+1 probléma megoldása: JOIN használata*

A JOIN egyetlen lekérdezésben adja vissza az összes sort: ha egy felhasználóhoz több rendelés tartozik, a felhasználó adatai annyi sorban ismétlődnek, ahány rendelése van. A kód ezért végigiterál az eredményen, és felhasználónként egyetlen bejegyzést épít fel a `$users` tömbben, amelyhez a hozzá tartozó rendelések beágyazott tömbként kapcsolódnak.

6.7.3. Lekérdezések optimalizálása

A lekérdezések optimalizálása már kisebb változtatásokkal is jelentős teljesítményjavulást eredményezhet (Beaulieu 2020).

SELECT * kerülése

A `SELECT *` használata kényelmes, de nem ajánlott. Mindig csak a szükséges mezőket kérdezzük le:

```
// Kerülendő
$stmt = $pdo->query("SELECT * FROM users");

// Ajánlott
$stmt = $pdo->query("SELECT id, email, username FROM users");
```

6.22. kódpélda. *SELECT * helyett konkrét mezők*

LIMIT használata

A `LIMIT` használata elengedhetetlen lapozásnál és nagy adathalmazok kezelésénél:

```php
$page = max(1, (int) ($_GET['page'] ?? 1));
$perPage = 20;
$offset = ($page - 1) * $perPage;
$stmt = $pdo->prepare("
    SELECT id, username, email
    FROM users
    ORDER BY created_at DESC
    LIMIT :limit OFFSET :offset
");
$stmt->bindValue('limit', $perPage, PDO::PARAM_INT);
$stmt->bindValue('offset', $offset, PDO::PARAM_INT);
$stmt->execute();
$users = $stmt->fetchAll(PDO::FETCH_ASSOC);
```

6.23. kódpélda. *LIMIT használata lapozáshoz*

COUNT- és EXISTS-optimalizálás

A `COUNT(*)` általában hatékonyabb, mint a `COUNT(column)`, és ha csak a létezést ellenőrizzük, célszerű az `EXISTS` használata:

```php
$count = (int) $pdo->query("
    SELECT COUNT(*) FROM users
")->fetchColumn();
$stmt = $pdo->prepare("
    SELECT EXISTS(
        SELECT 1 FROM users WHERE email = :email
    )
");
$stmt->execute(['email' => 'kiss.janos@example.com']);
$exists = (bool) $stmt->fetchColumn();
```

6.24. kódpélda. *COUNT(*) és EXISTS összehasonlítása*

Megjegyzés

A PHP-keretrendszerek az adatbázis-műveletekhez gyakran ORM (*Object-Relational Mapping*) könyvtárat használnak, amely az adatbázis tábláit és sorait objektumokként kezeli, így az SQL-utasítások jelentős része automatikusan előáll. Ilyen például a keretrendszertől független Doctrine ORM vagy a Laravel keretrendszer Eloquent komponense. Az ORM-ek magasabb absztrakciós szinten dolgoznak, a hátterükben azonban ugyanazok a PDO-alapú, előkészített utasításokat használó műveletek futnak, amelyeket ebben a fejezetben megismertünk.

6.8. Gyakorlófeladatok

A következő feladatok egyetlen webshopalkalmazás köré épülnek. Az 1. feladatban létrehozott séma és mintaadatok a továbbiakban változatlanul felhasználhatók. Minden feladatnál javasolt az előkészített utasítások használata és a kimenetek HTML-kódolása.

1. Futtassuk le az alábbi SQL-szkriptet egy `webshop` nevű adatbázisban:

```
CREATE TABLE categories (
    id   INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(100) NOT NULL UNIQUE
);
CREATE TABLE products (
    id          INT AUTO_INCREMENT PRIMARY KEY,
    name        VARCHAR(255) NOT NULL UNIQUE,
    price       DECIMAL(10,2) NOT NULL,
    stock       INT NOT NULL DEFAULT 0,
    category_id INT,
    FOREIGN KEY (category_id) REFERENCES categories(id)
);
CREATE TABLE orders (
    id          INT AUTO_INCREMENT PRIMARY KEY,
    total_price DECIMAL(10,2) NOT NULL,
    created_at  DATETIME NOT NULL
);
CREATE TABLE order_items (
    id         INT AUTO_INCREMENT PRIMARY KEY,
    order_id   INT NOT NULL,
    product_id INT NOT NULL,
    quantity   INT NOT NULL,
    unit_price DECIMAL(10,2) NOT NULL,
    FOREIGN KEY (order_id)   REFERENCES orders(id),
    FOREIGN KEY (product_id) REFERENCES products(id)
);
INSERT INTO categories (name) VALUES ('Elektronika'),
    ('Könyvek'), ('Ruházat');
INSERT INTO products (name, price, stock, category_id)
    VALUES
    ('Laptop',         1499.99, 10, 1),
    ('Okostelefon',     799.00,  5, 1),
    ('PHP kézikönyv',   49.90, 20, 2),
    ('Póló',            29.90, 50, 3);
```

Hozzunk létre egy `db.php` fájlt, amely PDO-kapcsolatot hoz létre a

`webshop` adatbázishoz. Hiba esetén naplózzuk az üzenetet `error_log()` segítségével, majd állítsuk le a szkript futását.

2. A `db.php` fájl felhasználásával listázzuk ki a `products` tábla összes rekordját, és jelenítsük meg HTML-táblázatban. A táblázat alatt jelenítsük meg a termékek számát. Ezután valósítsunk meg terméknév szerinti szűrést a `$_GET['name']` paraméter alapján.

3. Készítsünk egy űrlapot új termék felvételéhez (megnevezés, ár RON-ban, raktárkészlet, kategória). Elküldés után validáljuk az adatokat:

   - `trim()` az üres karakterek eltávolítására;
   - `filter_var()` az ár (pozitív szám) és a raktárkészlet (nemnegatív egész) ellenőrzésére.

   Hiba esetén jelenítsünk meg hibalistát, és őrizzük meg az űrlapmezők értékeit. Sikeres validáció esetén szúrjuk be az adatokat, és jelenítsük meg a `lastInsertId()` értékét. Ha a terméknév már létezik (SQLSTATE 23000), jelenítsünk meg tájékoztató hibaüzenetet.

4. Készítsünk egy termékszerkesztő oldalt, amely az URL-ben kapott `id` alapján lekéri a termék adatait, és előre kitöltött űrlapon jeleníti meg. Elküldés után frissítsük a rekordot, és jelezzük, hogy történt-e módosítás, illetve hogy létezik-e a megadott azonosítóhoz tartozó rekord.

5. Valósítsunk meg egy kétlépéses törlési folyamatot. Az első lépésben jelenítsük meg a termék adatait és egy megerősítő gombot, a második lépésben hajtsuk végre a törlést. Az eredmény alapján jelezzük:

   - sikeres törlés;
   - nem létező rekord;
   - integritási hiba (SQLSTATE 23000), ha a termékre rendelési tétel hivatkozik.

6. Valósítsunk meg egy `ProductRepository` osztályt, amely konstruktoron keresztül kap egy `PDO`-példányt, és tartalmazza a következő metódusokat: `create()`, `findById()`, `findAll()`, `update()`, `delete()`. Az osztály kezelje a `PDOException` kivételeket. Készítsünk egy `index.php` fájlt, amely bemutatja az osztály használatát: létrehozás, lekérdezés, frissítés és törlés.

7. Listázzuk ki a termékeket a hozzájuk tartozó kategórianévvel egyetlen `LEFT JOIN`-lekérdezéssel. Valósítsunk meg lapozást oldalanként 3 termékkel, a `LIMIT`- és `OFFSET`-értékeket `bindValue()` és `PDO::PARAM_INT` típussal megadva.

8. Valósítsunk meg egy rendelésleadási folyamatot tranzakcióval. A folyamat lépései:

   - ellenőrizzük a készletet minden terméknél (hiány esetén dobjunk kivételt);
   - szúrjunk be egy új rekordot az `orders` táblába (végösszeg RON-ban és időbélyeggel);
   - szúrjuk be a tételeket az `order_items` táblába;
   - csökkentsük a `stock` mezőt minden érintett terméknél.

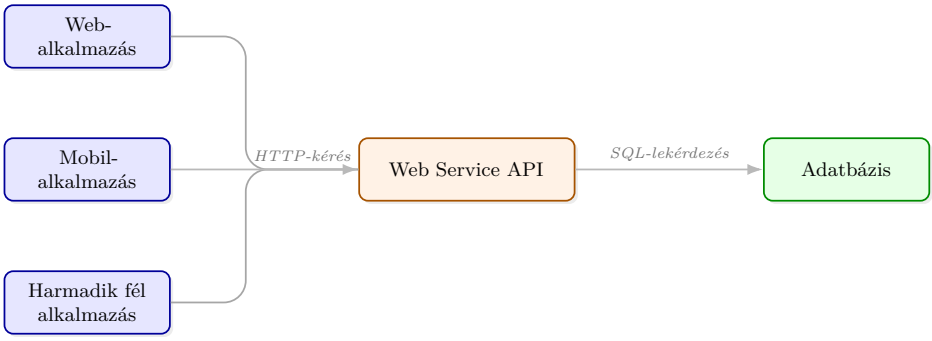
   Az összes műveletet egy tranzakcióban hajtsuk végre. Sikeres művelet esetén hívjuk meg a `commit()` metódust, hiba esetén a `rollBack()` metódust. Ellenőrizzük az `inTransaction()` állapotát is.

7. FEJEZET

REST API-FEJLESZTÉS

7.1. Webszolgáltatások áttekintése

A webszolgáltatások olyan szoftverrendszerek, amelyek lehetővé teszik különböző alkalmazások kommunikációját és adatcseréjét az interneten keresztül, ahogy azt a 7.1. ábra szemlélteti. A két legelterjedtebb megközelítés a *SOAP* és a *REST*. A *SOAP* egy XML-alapú, szigorúan szabványosított protokoll, amelyet elsősorban vállalati rendszerekben alkalmaznak. A *REST* ezzel szemben egy rugalmas, HTTP-re épülő architekturális stílus, amely mára a webes és mobilfejlesztés egyik legelterjedtebb megközelítésévé vált.



7.1. ábra. *Webszolgáltatás architektúra*

A két megközelítés főbb jellemzőit a 7.1. táblázat foglalja össze. Megjegyzendő, hogy a SOAP mára főként régebbi rendszerek integrációjánál fordul elő, új fejlesztéseknél jellemzően REST-alapú megoldásokat választanak. Ez a fejezet kizárólag a REST-alapú API-fejlesztéssel foglalkozik.

Néhány példa webszolgáltatásra: időjárás-lekérdező szolgáltatások, fizetési átjárók, közösségi média API-k, térképszolgáltatások és felhőalapú tárhelymegoldások.

7.1. táblázat. *A SOAP és a REST összehasonlítása*

	SOAP	**REST**
Protokoll	XML-alapú, saját protokoll	HTTP-protokollra épül
Szabványok	Szigorú (WSDL, WS-*)	Rugalmas konvenciók
Adatformátum	Kizárólag XML	JSON, XML, HTML, YAML
Komplexitás	Összetett, de formális	Egyszerűbb implementáció
Skálázhatóság	Nehezebb	Könnyebb
Felhasználás	Vállalati rendszerek	Webes és mobil alkalmazások

7.2. A REST alapjai

A REST (Representational State Transfer) egy architekturális stílus, amely meghatározza a webes szolgáltatások tervezésének alapelveit (Fielding 2000). A neve arra utal, hogy a kliens erőforrások reprezentációit kéri el a szervertől. Az állapot kezelése a kliens oldalán történik, és HTTP-protokollon keresztül kerül átadásra.

A REST-alapú API-k széles körben elterjedtek a modern webfejlesztésben, mivel egyszerű, skálázható és szabványos módot biztosítanak az erőforrások eléréséhez. A következő alfejezetekben áttekintjük a REST alapelveit, az erőforrások és reprezentációk fogalmát, valamint a gyakorlati megvalósítás részleteit.

7.2.1. REST-alapelvek

A REST hat fő alapelvre épül:

1. **Kliens–szerver architektúra**
   - A kliens és szerver szerepek világos szétválasztása.
   - A kliens nem foglalkozik az adattárolással.
   - A szerver nem foglalkozik a felhasználói felülettel.
   - A komponensek függetlenül fejleszthetők.
2. **Állapotmentesség**
   - Minden kérésnek tartalmaznia kell minden szükséges információt.
   - A szerver nem tárol munkamenet-állapotot.

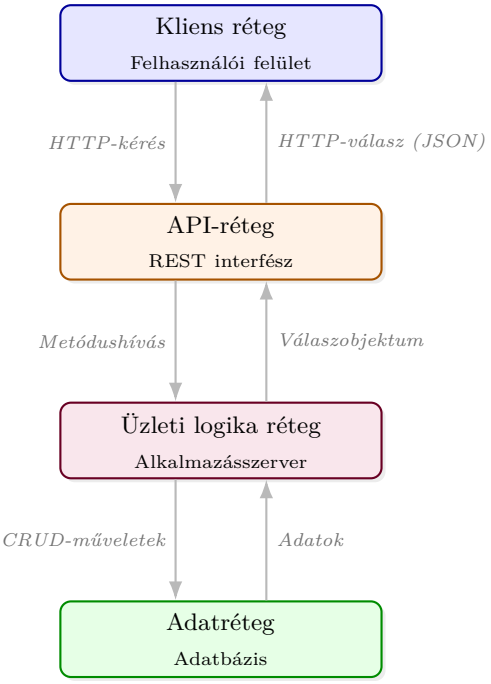
3. **Gyorsítótárazhatóság**
    - A válaszoknak jelezniük kell, hogy gyorsítótárazhatók-e.
    - Csökkenti a szerver terhelését, valamint javítja a skálázhatóságot és a teljesítményt.
4. **Egységes interfész**
    - Szabványosított kommunikáció a komponensek között.
    - Egyedi erőforrás-azonosítók (URI-k).
    - Erőforrások manipulálása reprezentációkon keresztül.
    - Önleíró üzenetek.
    - HATEOAS (Hypermedia as the Engine of Application State).
5. **Réteges rendszer**
    - A komponensek hierarchikus rétegekbe szervezhetők (lásd a 7.2. ábrát).
    - Minden réteg csak a közvetlen szomszédaival kommunikál.
    - Javítja a modularitást és a skálázhatóságot.
6. **Code on Demand (opcionális)**
    - A kliens funkcionalitása bővíthető szerverről letöltött kóddal (pl. JavaScript).
    - Ez az egyetlen opcionális megszorítás.

7.2.2. Erőforrások és reprezentációk

A REST központi fogalma az erőforrás. Minden, amit az API-n keresztül el szeretnénk érni vagy kezelni szeretnénk, erőforrásnak tekinthető.

Erőforrások jellemzői

- Minden erőforrásnak egyedi azonosítója (URI) van.
- Az erőforrások különböző reprezentációkkal rendelkezhetnek.
- Egy erőforrás lehet:
    - Dokumentum (pl. egy felhasználó adatai);
    - Gyűjtemény (pl. felhasználók listája);
    - Szolgáltatás (pl. keresés).



7.2. ábra. *A REST-architektúra réteges működése*

Példa URI-k:

URI	Erőforrás
`/users`	Felhasználók gyűjteménye
`/users/123`	Egy konkrét felhasználó
`/users/123/orders`	Egy felhasználó rendelései
`/products?category=books`	Könyv kategóriájú termékek

Reprezentációk

Az erőforrások különböző formátumokban reprezentálhatók. A leggyakoribb reprezentációs formátumok:

- JSON (JavaScript Object Notation);
- XML (eXtensible Markup Language);
- HTML (Hypertext Markup Language);
- YAML (YAML Ain't Markup Language).

Példa egy felhasználó JSON-reprezentációjára:

```json
{
    "id": 123,
    "username": "nagy_janos",
    "email": "nagy.janos@example.com",
    "created_at": "2024-01-15T10:30:00Z",
    "links": {
        "self": "/users/123",
        "orders": "/users/123/orders",
        "profile": "/users/123/profile"
    }
}
```

7.1. kódpélda. *Erőforrás JSON-reprezentációja*

Ugyanez az erőforrás XML-reprezentációban:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<user>
    <id>123</id>
    <username>nagy_janos</username>
    <email>nagy.janos@example.com</email>
    <created_at>2024-01-15T10:30:00Z</created_at>
    <links>
        <self>/users/123</self>
        <orders>/users/123/orders</orders>
        <profile>/users/123/profile</profile>
    </links>
</user>
```

7.2. kódpélda. *Ugyanaz az erőforrás XML-reprezentációban*

A kliens az `Accept` HTTP-fejléc segítségével kérheti az erőforrás kívánt reprezentációját:

```
Accept: application/json
Accept: application/xml
Accept: text/html
```

A szerver a `Content-Type` fejlécben jelzi a visszaküldött reprezentáció típusát:

```
Content-Type: application/json; charset=utf-8
Content-Type: application/xml; charset=utf-8
Content-Type: text/html; charset=utf-8
```

7.3. API-végpontok tervezése

A jól megtervezett API-végpontok (*endpoint*ok) intuitívak, konzisztensek és könnyen használhatók. Kialakításukkor figyelembe kell venni az erőforrások azonosítását, a végpontok elnevezését, a hierarchia meghatározását és a műveletek definiálását.

7.3.1. Erőforrás-alapú URL-tervezés

Az erőforrások a rendszerben kezelt entitások (például felhasználók, termékek és rendelések), amelyeket egyedi URL-lel azonosítunk.

URL-konvenciók

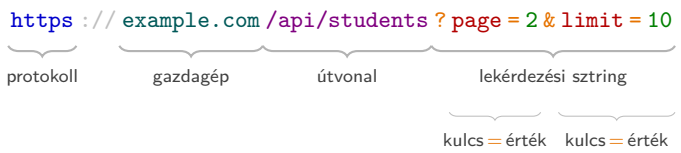
A konzisztens és érthető URL-struktúra kialakításához az alábbi konvenciókat érdemes követni:

- **Főnevek használata:** az URL-ekben főneveket használunk az erőforrások azonosítására. Az igéket kerüljük, mivel a műveletek típusát a HTTP-metódusok jelzik.
  - **Helyes:** `/users`, `/products`, `/orders`.
  - **Helytelen:** `/getUsers`, `/createProduct`, `/deleteOrder`.
- **Többes szám használata:** az erőforrásokat következetesen többes számban nevezzük meg, mivel általában erőforrás-gyűjteményekkel dolgozunk.
  - **Helyes:** `/users`, `/categories`.
  - **Helytelen:** `/user`, `/category`.
- **Hierarchikus kapcsolatok:** az erőforrások közötti kapcsolatokat az URL-struktúrában jelenítjük meg:
  - `/users/{id}/orders` – egy felhasználó rendelései;
  - `/orders/{id}/items` – egy rendelés tételei;
  - `/products/{id}/reviews` – egy termék értékelései.

7.3.2. Lekérdezési paraméterek

A lekérdezési paraméterek (*query parameters*) lehetővé teszik az API rugalmas használatát anélkül, hogy új végpontokat kellene létrehozni. Különösen hasznosak listázó végpontoknál, ahol az eredményeket szűrni, rendezni vagy lapozni szeretnénk.

A lekérdezési sztring az URL azon része, amely a „?” karakter után következik, és `kulcs=érték` párokat tartalmaz „&” elválasztóval. A 7.3. ábra az URL felépítését szemlélteti.



7.3. ábra. *Az URL felépítése és a lekérdezési sztring összetevői*

Szűrési paraméterek

A szűrési paraméterek lehetővé teszik az eredményhalmaz szűkítését megadott feltételek alapján:

Paraméter	Leírás
`/products?category=electronics`	Kategória szerinti szűrés
`/products?price_min=100`	Minimum ár megadása
`/products?price_max=500`	Maximum ár megadása
`/products?brand=samsung`	Gyártó szerinti szűrés
`/users?status=active`	Státusz szerinti szűrés
`/products?brand=samsung&price_max=500`	Több feltétel szerinti szűrés

Rendezési paraméterek

A rendezési paraméterek meghatározzák az eredmények sorrendjét:

Paraméter	Leírás
`/products?sort=price_desc`	Ár szerint csökkenő sorrend
`/products?sort=name_asc`	Név szerint növekvő sorrend
`/users?sort=created_at_desc`	Létrehozás dátuma szerint csökkenő sorrend
`/products?sort=category_asc,price_desc`	Rendezés több mező alapján

Lapozási paraméterek

A lapozás elengedhetetlen nagy adathalmazok kezelésénél. Az offsetalapú lapozásnál az `offset` a kihagyott elemek számát, a `limit` pedig az oldalon visszaadott elemek számát jelöli. Az oldal alapú lapozásnál a `page` az aktuális oldal sorszámát, a `per_page` pedig az oldalméretet jelöli.

Példa	Típus	Leírás
`/products?offset=40&limit=20`	offsetalapú	A 41–60. elemek lekérése
`/products?page=2&per_page=20`	oldalalapú	A 2. oldal lekérése, oldalanként 20 elemmel

A válaszban célszerű visszaadni a lapozással kapcsolatos metaadatokat:

```
{
    "data": [...],
    "pagination": {
        "total_items": 100,
        "total_pages": 5,
        "current_page": 2,
        "per_page": 20,
        "next_page": 3,
        "prev_page": 1
    }
}
```

7.4. HTTP-metódusok és státuszkódok

A modern webes API-k a HTTP-protokollra épülnek. A protokoll két fő komponense a HTTP-metódusok, amelyek meghatározzák a kért műveletet, valamint a státuszkódok, amelyek a művelet eredményét jelzik.

7.4.1. HTTP-metódusok

A HTTP-metódusok (más néven HTTP-igék) meghatározzák a szerver által végrehajtandó műveletet. Az egyes metódusokat két fontos tulajdonsággal jellemezhetjük: *idempotens* és *biztonságos* (lásd a 7.2. táblázatot).

7.2. táblázat. *HTTP-metódusok tulajdonságai*

Metódus	Idempotens	Biztonságos
GET	Igen	Igen
POST	Nem	Nem
PUT	Igen	Nem
PATCH	Nem feltétlenül	Nem
DELETE	Igen	Nem

GET-metódus

A GET-metódus erőforrások lekérdezésére szolgál, a REST-architektúrában az „olvasás” műveletet valósítja meg. Idempotens és biztonságos: nem módosítja a szerver állapotát, ezért ismételt hívása sem változtat rajta. A paramétereket az URL-ben adjuk át, a válaszok pedig gyorsítótárazhatók. Tipikus felhasználási területe az erőforrások listázása, egyes elemek lekérdezése, szűrés és keresés.

```
# Termékek listázása
GET /api/products

# Egy konkrét termék lekérdezése
GET /api/products/123

# Szűrés és rendezés
GET /api/products?category=electronics&sort=price_desc

# Lapozás
GET /api/products?page=2&per_page=20

# Keresés
GET /api/products/search?q=laptop
```

POST-metódus

A POST-metódus új erőforrások létrehozására és összetett műveletek végrehajtására szolgál, a REST-architektúrában a „létrehozás” műveletet valósítja meg. Nem idempotens és nem biztonságos: módosítja az erőforrás állapotát, és ismételt hívás esetén több erőforrást is létrehozhat. Az adatokat a kérés törzsében (request body) adjuk át. Gyakran használjuk új erőforrások létrehozására, fájlfeltöltésre és hitelesítési műveletekre.

```
# Új termék létrehozása
POST /api/products
Content-Type: application/json

{
    "name": "Új laptop",
    "price": 999.99,
    "category": "electronics"
}

# Fájl feltöltés
POST /api/products/123/images
Content-Type: multipart/form-data

# Bejelentkezés
POST /api/auth/login
Content-Type: application/json

{
    "email": "user@example.com",
    "password": "secret123"
}
```

PUT-metódus

A PUT-metódus egy erőforrás teljes állapotának módosítására vagy létrehozására szolgál. A meglévő állapotot teljes egészében lecseréli, ezért az erőforrás minden mezőjét meg kell adni. Ha az erőforrás nem létezik, a szerver létrehozhatja.

```
# Termék teljes módosítása
PUT /api/products/123
Content-Type: application/json

{
    "name": "Módosított laptop",
    "price": 1299.99,
    "category": "electronics",
    "description": "Új leírás",
    "in_stock": true,
    "specifications": {
        "cpu": "Intel i7",
        "ram": "16GB",
        "storage": "512GB SSD"
    }
}

# Felhasználói profil frissítése
PUT /api/users/456/profile
Content-Type: application/json

{
    "full_name": "Nagy János",
    "email": "nagy.janos@example.com",
    "address": "Fő utca 12.",
    "phone": "+40XXXXXXXXX",
    "preferences": {
        "newsletter": true,
        "language": "en",
        "timezone": "UTC+1"
    }
}
```

PATCH-metódus

A PATCH-metódus egy erőforrás részleges módosítására szolgál: csak a módosítani kívánt mezőket kell küldeni, a többi változatlan marad. Általában kisebb adatforgalommal jár, mint a PUT-metódus használata, és rugalmasabb megoldást biztosít, mivel különböző módosítási formátumokat támogat. Nem minden esetben idempotens: a módosítás jellegétől függően ismételt hívás eltérő eredményt adhat.

```
# Egyszerű mezőmódosítás
PATCH /api/products/123
Content-Type: application/json

{
    "price": 1199.99,
    "in_stock": false
}

# JSON Patch formátum használata
PATCH /api/products/123
Content-Type: application/json-patch+json

[
    { "op": "replace", "path": "/price", "value": 1199.99 },
    { "op": "add", "path": "/tags",
      "value": ["sale", "featured"] },
    { "op": "remove", "path": "/old_field" }
]

# Beágyazott objektumok módosítása
PATCH /api/users/456/profile
Content-Type: application/json

{
    "preferences": {
        "newsletter": false
    },
    "address": {
        "city": "Csíkszereda"
    }
}
```

DELETE-metódus

A DELETE-metódus egy erőforrás törlésére szolgál. Idempotens: az erőforrás törlése után az ismételt kérés is ugyanazt az állapotot eredményezi. Sikeres törlés esetén a szerver általában `204 No Content` választ küld. Fontos figyelembe venni a kapcsolódó erőforrásokat (referenciális integritás), a jogosultságkezelést, valamint kritikus adatok esetén a *soft delete* megközelítés alkalmazását (logikai törlés: az adat nem törlődik fizikailag, csupán töröltként jelölődik meg).

```
# Egyszerű törlés
DELETE /api/products/123
```

```
# Törlés megerősítési tokennel
DELETE /api/products/123
Authorization: Bearer <token>
X-Confirmation-Token: abc123

# Többszörös törlés
DELETE /api/products?ids=123,456,789

# Soft delete jelzése
DELETE /api/users/789
X-Delete-Type: soft
```

A példákban szereplő `Authorization: Bearer <token>` fejléc a kérést küldő fél azonosítására szolgál. A tokenalapú hitelesítést a 7.7. alfejezetben mutatjuk be részletesen.

7.4.2. HTTP-státuszkódok

A HTTP-státuszkódok háromjegyű számok, amelyek a kérés feldolgozásának eredményét jelzik. A státuszkódokat öt fő kategóriába (1xx–5xx) soroljuk. Az alábbiakban a gyakorlatban leggyakoribb négyet mutatjuk be.

2xx – Sikeres válaszok

Kód	Név	Leírás
200	OK	Sikeres kérés; a válasz törzse tartalmazza az erőforrás adatait
201	Created	Új erőforrás sikeresen létrehozva; a válasz általában tartalmazza az új erőforrást
204	No Content	Sikeres művelet válasz tartalom nélkül; gyakori DELETE vagy PUT/PATCH után

3xx – Átirányítási válaszok

Kód	Név	Leírás
301	Moved Permanently	Az erőforrás véglegesen áthelyezve; az új URL a `Location` fejlécben található
302	Found	Ideiglenes átirányítás; az eredeti URL továbbra is használható
304	Not Modified	A kliens gyorsítótárazott verziója továbbra is érvényes; az erőforrás nem változott

4xx – Kliensoldali hibák

Kód	Név	Leírás
400	Bad Request	Érvénytelen kérés; hibás JSON vagy hiányzó paraméterek
401	Unauthorized	Hiányzó vagy érvénytelen hitelesítés
403	Forbidden	A kliens hitelesített, de nincs megfelelő jogosultsága
404	Not Found	A kért erőforrás nem található
409	Conflict	A kérés ütközik az erőforrás aktuális állapotával
422	Unprocessable Entity	Validációs hiba; a kérés szintaktikailag helyes, de tartalmilag hibás
429	Too Many Requests	Túl sok kérés rövid idő alatt; a `Retry-After` fejléc jelezheti a várakozási időt

5xx – Szerveroldali hibák

Kód	Név	Leírás
500	Internal Server Error	Általános szerveroldali hiba; például kezeletlen kivétel
502	Bad Gateway	A proxy vagy gateway érvénytelen választ kapott egy másik szolgáltatástól
503	Service Unavailable	A szolgáltatás átmenetileg nem elérhető
504	Gateway Timeout	Egy háttérszolgáltatás nem válaszolt időben

7.5. JSON-válaszok

7.5.1. JSON-alapok

A JSON (JavaScript Object Notation)[1] egy szövegalapú adatcsereformátum, amelyet emberek és gépek egyaránt könnyen olvashatnak. Eredetileg a JavaScript szintaxisára épült, mára azonban nyelvfüggetlen formátummá vált, és REST API-k esetén a legelterjedtebb adatátviteli formátum. Hat alaptípust támogat (`number`, `string`, `boolean`, `null`, `object`, `array`), amelyeket a 7.3. táblázat foglal össze.

[1] https://tools.ietf.org/html/rfc8259

7.3. táblázat. *JSON-adattípusok*

Típus	Értékek	Megjegyzés
`number`	`42`, `3.14`, `1.2e-10`	Egész és lebegőpontos számok
`string`	`"Hello"`, `"Line 1\nLine 2"`	Idézőjelek között, escape karakterekkel
`boolean`	`true`, `false`	Mindig kisbetűvel írva
`null`	`null`	Hiányzó vagy üres érték jelölésére
`object`	`{"key": "value"}`	Kapcsos zárójelek között megadott név–érték párok, ahol a nevek mindig idézőjelbe tett sztringek
`array`	`[1, "two", true]`	Szögletes zárójelek közötti elemek, akár vegyes típusokkal

7.5.2. JSON feldolgozása PHP-ben

`json_encode()`

A `json_encode()` függvény PHP-értékeket alakít át JSON-formátumú szöveggé.

Szintaxis

```
json_encode(
    mixed $value,
    int   $flags = 0,
    int   $depth = 512
): string|false
```

Paraméterek:

- `$value`: a konvertálandó PHP-érték (tömb, objektum vagy skalár);
- `$flags`: opcionális jelzők (flagek), amelyek a `|` operátorral kombinálhatók:
  - `JSON_PRETTY_PRINT`: formázott, behúzott kimenetet generál;
  - `JSON_UNESCAPED_UNICODE`: a nem ASCII-karaktereket változatlanul hagyja. Enélkül minden ilyen karakter `\u` és négy hexadecimális számjegy alakban jelenik meg, például az `á` (U+00E1) karakter `"\u00e1"` formában (ezt nevezik itt escape-elt alaknak);

  - `JSON_UNESCAPED_SLASHES`: a perjeleket (`/`) változatlanul hagyja;
  - `JSON_THROW_ON_ERROR`: hiba esetén kivételt dob `false` helyett.

- `$depth`: maximális beágyazási mélység (alapértelmezett: 512).

Visszatérési érték: JSON-kódolt sztring, hiba esetén `false`.

```php
$data = ["name" => "Kovács János", "age" => 30];
// Alapértelmezés: az ékezetes karakterek \uXXXX alakban
    jelennek meg
echo json_encode($data);
// {"name":"Kov\u00e1cs J\u00e1nos","age":30}
// Ékezetek megtartása és formázott kimenet
echo json_encode($data, JSON_UNESCAPED_UNICODE |
    JSON_PRETTY_PRINT);
// {
//     "name": "Kovács János",
//     "age": 30
// }
```

7.3. kódpélda. *`json_encode()` alapvető használata*

`json_decode()`

A `json_decode()` függvény JSON-formátumú szöveget alakít át PHP-értékké. Alapértelmezés szerint objektumot ad vissza, de kérhetünk asszociatív tömböt is.

Szintaxis

```php
json_decode(
    string $json,
    bool   $associative = false,
    int    $depth = 512,
    int    $flags = 0
): mixed
```

Paraméterek:

- `$json`: a dekódolandó JSON-sztring;
- `$associative`: ha `true`, asszociatív tömböt ad vissza objektum helyett;
- `$depth`: maximális beágyazási mélység (alapértelmezett: 512);
- `$flags`: dekódolási opciók, például `JSON_THROW_ON_ERROR`.

Visszatérési érték: a dekódolt PHP-érték, hiba esetén `null`.

```php
$json = '{"name":"Kovács János","age":30}';

// Objektumként (alapértelmezett)
$obj = json_decode($json);
echo $obj->name; // Kovács János

// Asszociatív tömbként
$arr = json_decode($json, true);
echo $arr['name']; // Kovács János
```

7.4. kódpélda. *`json_decode()` objektumként és tömbként*

Hibakezelés

A `JSON_THROW_ON_ERROR` jelző hatására a `json_decode()` és a `json_encode()` függvény hibás bemenet esetén kivételt dob, amelyet `try-catch` blokkal kezelhetünk:

```php
try {
    $data = json_decode($json, true, 512,
        JSON_THROW_ON_ERROR);
} catch (JsonException $e) {
    echo "JSON hiba: " . $e->getMessage();
}
```

7.5. kódpélda. *JSON-hibakezelés kivétellel*

7.5.3. JSON használata API-válaszokban

Az API-válaszok esetében fontos a konzisztens és jól strukturált formátum. Elterjedt megközelítés az úgynevezett *wrapper* (envelope) minta, amelyben a tényleges adat egy `data` mezőbe kerül, míg a többi mező metaadatokat tartalmaz:

```json
{
    "status": "success",
    "data": {
        "id": 123,
        "name": "Product Name",
        "price": 99.99
    }
}
```

Hiba esetén a `status` mező értéke `"error"`, a `message` mező pedig leírja a probléma okát:

```
{
    "status": "error",
    "code": 400,
    "message": "Invalid request parameters"
}
```

7.6. REST API-implementáció PHP-ben

Egy egyszerű hallgatókezelő API-n keresztül bemutatjuk a projekt felépítését, a routing-logikát és a CRUD-műveleteket. A hitelesítésről és a CORS-kezelésről a 7.7. alfejezet szól részletesen.

7.6.1. Alapvető API-szerkezet

Projektstruktúra

Egy jól szervezett REST API-projektstruktúra megkönnyíti a kód karbantartását és bővítését. Az alábbi példa egy lehetséges mappaszerkezetet mutat be:

student-api/

```
|-- api/
|   |-- index.php       # Front controller
|   |-- config.php      # Adatbázis-konfiguráció
|   |-- helpers.php     # Segédfüggvények
|-- database.sql        # Adatbázis-séma
|-- setup.php           # Telepítő szkript
+-- .htaccess           # URL-átírás (opcionális)
```

A projekt fő elemei:

- **index.php**: az API egyetlen belépési pontja (*front controller* minta);
- **config.php**: adatbázis-kapcsolat és -konfiguráció;
- **helpers.php**: gyakran használt segédfüggvények (JSON-válasz küldése, hibakezelés);
- **setup.php**: a `database.sql`-lel azonos sémát és mintaadatokat hozza létre böngészőből futtatva;
- **.htaccess**: URL-ek tisztítása (opcionális, de ajánlott).

Adatbázisséma

Az API-példában egy egyszerű hallgatókezelő rendszert valósítunk meg. Az adatbázis szerkezete:

```sql
CREATE DATABASE IF NOT EXISTS student_api
    CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci;

USE student_api;

CREATE TABLE students (
    id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    email VARCHAR(100) NOT NULL UNIQUE,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
               ON UPDATE CURRENT_TIMESTAMP
);

-- Példa adatok
INSERT INTO students (name, email) VALUES
('Nagy János', 'nagy.janos@example.com'),
('Kiss Anna', 'kiss.anna@example.com'),
('Kovács Péter', 'kovacs.peter@example.com');
```

7.6. kódpélda. *Adatbázisséma (database.sql)*

Adatbázis-konfiguráció

Az adatbázis-kapcsolatot PDO segítségével valósítjuk meg, amely biztonságos és korszerű megközelítést biztosít:

```php
<?php
// Adatbázis kapcsolat konfigurációja
define('DB_HOST', 'localhost');
define('DB_NAME', 'student_api');
define('DB_USER', 'root');
define('DB_PASS', '');
define('DB_CHARSET', 'utf8mb4');

// PDO kapcsolat létrehozása
function getDbConnection(): PDO
{
    static $pdo = null;

    if ($pdo === null) {
        $dsn = sprintf(
            "mysql:host=%s;dbname=%s;charset=%s",
```

```
            DB_HOST,
            DB_NAME,
            DB_CHARSET
        );

        $options = [
            PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION,
            PDO::ATTR_DEFAULT_FETCH_MODE => PDO::FETCH_ASSOC,
            PDO::ATTR_EMULATE_PREPARES => false,
        ];

        try {
            $pdo = new PDO($dsn, DB_USER, DB_PASS, $options);
        } catch (PDOException $e) {
            // Éles környezetben ne adjunk ki részletes
               hibaüzenetet!
            http_response_code(500);
            echo json_encode([
                'status' => 'error',
                'message' => 'Adatbázis kapcsolódási hiba'
            ]);
            exit;
        }
    }

    return $pdo;
}
```

7.7. kódpélda. *Adatbázis-kapcsolat (config.php)*

A `static $pdo = null` deklaráció hatására a változó értéke megőrződik a függvény egymást követő hívásai között. Az első hívásnál a `$pdo` még `null`, ezért létrejön a PDO-kapcsolat. Minden további hívásnál a `$pdo` már tartalmazza az egyszer létrehozott objektumot, így nem keletkezik felesleges új kapcsolat. Ez az egyszerű *singleton* minta (Freeman–Robson 2021).

Segédfüggvények

A segédfüggvények egységesen kezelik az API gyakori műveleteit, például a JSON-válaszok küldését és a hibakezelést:

```
<?php
// JSON válasz küldése
function sendJson(mixed $data, int $statusCode = 200): void
{
    http_response_code($statusCode);
    header('Content-Type: application/json; charset=UTF-8');
```

```
    echo json_encode($data, JSON_UNESCAPED_UNICODE |
        JSON_UNESCAPED_SLASHES | JSON_THROW_ON_ERROR);
    exit;
}

// Sikeres válasz küldése
function sendSuccess(mixed $data, string $message =
    'Sikeres', int $statusCode = 200): void
{
    sendJson(['status' => 'success', 'message' => $message,
        'data' => $data], $statusCode);
}

// Hibaválasz küldése
function sendError(string $message, int $statusCode = 400,
    ?array $errors = null): void
{
    $response = ['status' => 'error', 'message' => $message];
    if ($errors !== null) {
        $response['errors'] = $errors;
    }
    sendJson($response, $statusCode);
}

// HTTP metódus ellenőrzése
function requireMethod(string $method): void
{
    if ($_SERVER['REQUEST_METHOD'] !== $method) {
        sendError("A(z) $method metódus nem engedélyezett",
            405);
    }
}

// JSON input beolvasása és dekódolása
function getJsonInput(): array
{
    $input = file_get_contents('php://input');
    if (empty($input)) return [];
    try {
        $data = json_decode($input, true, 512,
            JSON_THROW_ON_ERROR);
        return is_array($data) ? $data : [];
    } catch (JsonException $e) {
        sendError('Érvénytelen JSON formátum', 400);
    }
}
```

7.8. kódpélda. *Segédfüggvények (helpers.php)*

Megjegyzés

A `sendJson()`, `sendSuccess()` és `sendError()` függvények egységes válaszformátumot biztosítanak az API-ban, ami megkönnyíti a kliensoldali feldolgozást.
A `getJsonInput()` függvény által használt `file_get_contents('php://input')` a HTTP-kérés nyers törzsét olvassa be. A `$_POST` szuperglobálissal ellentétben ez tetszőleges `Content-Type` esetén működik, ezért REST API-kban ez az ajánlott módszer JSON-adatok beolvasására.

7.6.2. Routing és kérések kezelése

Egyszerű routing-implementáció

A *routing* feladata, hogy a beérkező HTTP-kéréseket a megfelelő műveletekhez irányítsa. REST API-k esetében az URL-struktúra és a HTTP-metódus együtt határozza meg a végrehajtandó műveletet.

```php
<?php
// Fájlok betöltése
require_once 'config.php';
require_once 'helpers.php';

// CORS fejlécek beállítása (később részletesen)
header('Access-Control-Allow-Origin: *');
header('Access-Control-Allow-Methods: GET, POST, PUT,
    DELETE, OPTIONS');
header('Access-Control-Allow-Headers: Content-Type,
    Authorization');

// Preflight kérés kezelése
if ($_SERVER['REQUEST_METHOD'] === 'OPTIONS') {
    http_response_code(204);
    exit;
}

// HTTP metódus és URI lekérése
$method = $_SERVER['REQUEST_METHOD'];
$uri = parse_url($_SERVER['REQUEST_URI'], PHP_URL_PATH);

// API prefix eltávolítása: /api/students helyett /students
$uri = str_replace('/api', '', $uri);
$uri = trim($uri, '/');

// URI szegmensekre bontása
$segments = explode('/', $uri);

// Routing logika
```

```php
$resource = $segments[0] ?? '';
$id = $segments[1] ?? null;

// Adatbázis kapcsolat
$pdo = getDbConnection();

// Routing táblázat
switch ($resource) {
    case 'students':
        handleStudents($pdo, $method, $id);
        break;

    case '':
        // API info endpoint
        sendSuccess([
            'name' => 'Student API',
            'version' => '1.0.0',
            'endpoints' => [
                'GET /api/students' => 'Hallgatók listája',
                'GET /api/students/{id}' => 'Hallgató
                    lekérése azonosító alapján',
                'POST /api/students' => 'Új hallgató
                    létrehozása',
                'PUT /api/students/{id}' => 'Hallgató
                    módosítása',
                'DELETE /api/students/{id}' => 'Hallgató
                    törlése'
            ]
        ]);
        break;

    default:
        sendError('Az erőforrás nem található', 404);
}
```

7.9. kódpélda. *Front controller (index.php)*

Megjegyzés

A `parse_url()` függvény biztonságosan kezeli az URL-t, és csak az útvonal részt adja vissza, a „query string” paraméterek nélkül. Ez segít a routing-logika egyszerűsítésében és a végpontok egyértelmű kezelésében.

A `$_SERVER` szuperglobális változó

A `$_SERVER` tömb a szerver és az aktuális HTTP-kérés környezeti információit tartalmazza. REST API-k fejlesztése során különösen fontos néhány

kulcs használata, amelyeket a 7.4. táblázat foglal össze.

7.4. táblázat. *A `$_SERVER` tömb fontosabb elemei*

Változó	Leírás
`REQUEST_METHOD`	A HTTP-metódus neve (GET, POST, PUT, DELETE stb.)
`REQUEST_URI`	A teljes kérés URI-ja (pl. `/api/students?page=2`)
`CONTENT_TYPE`	A kérés tartalomtípusa (pl. `application/json`)
`HTTP_AUTHORIZATION`	Az Authorization fejléc értéke hitelesítéshez
`QUERY_STRING`	Az URL lekérdezési paraméterei (pl. `page=2&limit=10`)
`REMOTE_ADDR`	A kliens IP-címe

HTTP-metódus-alapú műveletek

Az egyes HTTP-metódusok a REST-konvenció szerint különböző műveletekhez rendelődnek:

```
function handleStudents(PDO $pdo, string $method, ?string
    $id): void
{
    switch ($method) {
        case 'GET':
            if ($id) {
                // GET /api/students/{id} - Egyedi hallgató
                    lekérése
                getStudent($pdo, $id);
            } else {
                // GET /api/students - Lista lekérése
                getStudents($pdo);
            }
            break;

        case 'POST':
            // POST /api/students - Új hallgató létrehozása
            if ($id) {
                sendError('Létrehozáshoz nem szükséges
                    azonosító', 400);
            }
            createStudent($pdo);
            break;

        case 'PUT':
            // PUT /api/students/{id} - Hallgató módosítása
```

```
            if (!$id) {
                sendError('Módosításhoz azonosító
                    szükséges', 400);
            }
            updateStudent($pdo, $id);
            break;

        case 'DELETE':
            // DELETE /api/students/{id} - Hallgató törlése
            if (!$id) {
                sendError('Töröléshez azonosító szükséges',
                    400);
            }
            deleteStudent($pdo, $id);
            break;

        default:
            sendError("A(z) $method metódus nem
                engedélyezett", 405);
    }
}
```

7.10. kódpélda. *Metódusalapú routing*

Megjegyzés

Nagyobb API-k esetében érdemes dedikált routerkönyvtárat használni (pl. `nikic/fast-route` vagy `Slim Framework`), mivel ezek hatékonyabb és könnyebben karbantartható routingmegoldást biztosítanak.

Query-paraméterek kezelése

A GET-kéréseknél gyakran használunk query-paramétereket szűréshez, rendezéshez és lapozáshoz:

```
function getStudents(PDO $pdo): void
{
    $page  = max(1, (int)($_GET['page']  ?? 1));
    $limit = min(100, max(1, (int)($_GET['limit'] ?? 10)));
    $sort  = $_GET['sort']  ?? 'name';
    $order = strtoupper($_GET['order'] ?? 'ASC');

    $allowedSorts = ['name', 'email', 'created_at'];
    if (!in_array($sort, $allowedSorts)) $sort = 'name';
    if (!in_array($order, ['ASC', 'DESC'])) $order = 'ASC';
```

```
    $offset = ($page - 1) * $limit;

    try {
        $stmt = $pdo->prepare(
            "SELECT * FROM students ORDER BY $sort $order
               LIMIT :limit OFFSET :offset"
        );
        $stmt->bindValue(':limit',  $limit,  PDO::PARAM_INT);
        $stmt->bindValue(':offset', $offset, PDO::PARAM_INT);
        $stmt->execute();
        $students = $stmt->fetchAll();

        $total = (int)$pdo->query("SELECT COUNT(*) FROM
            students")->fetchColumn();

        sendSuccess([
            'students'   => $students,
            'pagination' => [
                'page'  => $page,
                'limit' => $limit,
                'total' => $total,
                'pages' => (int)ceil($total / $limit)
            ]
        ]);
    } catch (PDOException $e) {
        sendError('Adatbázis hiba', 500);
    }
}
```

7.11. kódpélda. *Query-paraméterek feldolgozása*

A `$sort` és `$order` változók közvetlenül bekerülnek az SQL-szövegbe (nem helyőrzőként), mert az adatbázis-motor nem fogadhatja el paraméterként az oszlopneveket és a rendezési irányt. Ez önmagában SQL-injection-kockázatot jelentene, ezért a kód előbb mindkét értéket whitelisttel ellenőrzi: csak az `$allowedSorts` tömbben szereplő mezőneveket, illetve az `ASC`/`DESC`-értékeket engedi át. Ha az érték nem szerepel a listában, a rendszer egy biztonságos alapértéket alkalmaz.

7.6.3. Teljes CRUD-példa

A következő alfejezetekben megvalósítjuk a négy alapvető CRUD-műveletet (Create, Read, Update, Delete) a hallgatói adatok kezelésére.

Create – Új hallgató létrehozása

```php
function createStudent(PDO $pdo): void
{
    $input = getJsonInput();

    if (empty($input['name']) || empty($input['email'])) {
        sendError('Hiányzó kötelező mezők: name, email',
            400);
    }

    $name  = trim($input['name']);
    $email = trim($input['email']);

    $errors = [];
    if (mb_strlen($name) < 2 || mb_strlen($name) > 100) {
        $errors['name'] = 'A névnek 2-100 karakter között
            kell lennie';
    }
    if (!filter_var($email, FILTER_VALIDATE_EMAIL)) {
        $errors['email'] = 'Érvénytelen e-mail formátum';
    }
    if (!empty($errors)) {
        sendError('Validációs hiba', 422, $errors);
    }

    try {
        $stmt = $pdo->prepare("INSERT INTO students (name,
            email) VALUES (:name, :email)");
        $stmt->execute([':name' => $name, ':email' =>
            $email]);

        $id   = $pdo->lastInsertId();
        $stmt = $pdo->prepare("SELECT * FROM students WHERE
            id = :id");
        $stmt->execute([':id' => $id]);

        sendSuccess($stmt->fetch(), 'Hallgató sikeresen
            létrehozva', 201);
    } catch (PDOException $e) {
        if ($e->getCode() === '23000') {
            sendError('Ez az e-mail cím már foglalt', 409);
        }
        sendError('Adatbázis hiba', 500);
    }
}
```

7.12. kódpélda. *Hallgató létrehozása*

POST-kérés példa:

```
POST /api/students
Content-Type: application/json

{
    "name": "Szabó Gábor",
    "email": "szabo.gabor@example.com"
}

Valasz (201 Created):
{
    "status": "success",
    "message": "Hallgató sikeresen létrehozva",
    "data": {
        "id": 4,
        "name": "Szabó Gábor",
        "email": "szabo.gabor@example.com",
        "created_at": "2024-01-22 10:30:00",
        "updated_at": "2024-01-22 10:30:00"
    }
}
```

Read – Hallgató lekérése ID alapján

```php
function getStudent(PDO $pdo, string $id): void
{
    // ID validálása
    if (!is_numeric($id) || $id < 1) {
        sendError('Érvénytelen azonosító formátum', 400);
    }
    try {
        $sql = "SELECT * FROM students WHERE id = :id";
        $stmt = $pdo->prepare($sql);
        $stmt->execute([':id' => $id]);
        $student = $stmt->fetch();
        if (!$student) {
            sendError('A hallgató nem található', 404);
        }
        sendSuccess($student);
    } catch (PDOException $e) {
        sendError('Adatbázis hiba', 500);
    }
}
```

7.13. kódpélda. *Hallgató lekérése*

GET-kérés példa:

```
GET /api/students/1

Valasz (200 OK):
{
    "status": "success",
    "message": "Sikeres",
    "data": {
        "id": 1,
        "name": "Nagy János",
        "email": "nagy.janos@example.com",
        "created_at": "2024-01-20 08:15:00",
        "updated_at": "2024-01-20 08:15:00"
    }
}
```

Update – Hallgató módosítása

```php
function updateStudent(PDO $pdo, string $id): void
{
    if (!is_numeric($id) || $id < 1) {
        sendError('Érvénytelen azonosító formátum', 400);
    }

    $checkStmt = $pdo->prepare("SELECT id FROM students
        WHERE id = :id");
    $checkStmt->execute([':id' => $id]);
    if (!$checkStmt->fetch()) {
        sendError('A hallgató nem található', 404);
    }

    $input = getJsonInput();
    if (empty($input)) {
        sendError('Nem érkezett frissítendő adat', 400);
    }

    $allowedFields = ['name', 'email'];
    $updateFields  = [];
    $params        = [':id' => $id];

    foreach ($input as $field => $value) {
        if (!in_array($field, $allowedFields)) continue;
        $updateFields[] = "$field = :$field";
        $params[":$field"] = $value;
    }
```

```
    if (empty($updateFields)) {
        sendError('Nincs érvényes mező a frissítéshez', 400);
    }

    try {
        $sql = "UPDATE students SET " . implode(', ',
            $updateFields) . " WHERE id = :id";
        $pdo->prepare($sql)->execute($params);

        $stmt = $pdo->prepare("SELECT * FROM students WHERE
            id = :id");
        $stmt->execute([':id' => $id]);

        sendSuccess($stmt->fetch(), 'Hallgató sikeresen
            módosítva');
    } catch (PDOException $e) {
        if ($e->getCode() === '23000') {
            sendError('Ez az e-mail cím már foglalt', 409);
        }
        sendError('Adatbázis hiba', 500);
    }
}
```

7.14. kódpélda. *Hallgató módosítása*

A függvény csak azokat a mezőket frissíti, amelyeket a kliens ténylegesen elküldött. A ciklus minden bemeneti mezőt ellenőriz az `$allowedFields` whitelist alapján, a jóváhagyott mezőkhöz `"field = :field"` alakú kifejezést fűz az `$updateFields` tömbhöz, és a hozzá tartozó értéket behelyezi a `$params` tömbbe. Az `implode(', ', $updateFields)` ezekből egyetlen `SET`-utasításrészt állít össze.

PUT-kérés példa:

```
PUT /api/students/1
Content-Type: application/json

{
    "email": "nagy.janos.uj@example.com"
}

Valasz (200 OK):
{
    "status": "success",
    "message": "Hallgató sikeresen módosítva",
    "data": {
        "id": 1,
```

```
        "name": "Nagy János",
        "email": "nagy.janos.uj@example.com",
        "created_at": "2024-01-20 08:15:00",
        "updated_at": "2024-01-22 11:45:00"
    }
}
```

Delete – Hallgató törlése

```php
function deleteStudent(PDO $pdo, string $id): void
{
    // ID validálása
    if (!is_numeric($id) || $id < 1) {
        sendError('Érvénytelen azonosító formátum', 400);
    }

    try {
        // Ellenőrizzük, létezik-e a hallgató
        $checkSql = "SELECT * FROM students WHERE id = :id";
        $checkStmt = $pdo->prepare($checkSql);
        $checkStmt->execute([':id' => $id]);
        $student = $checkStmt->fetch();

        if (!$student) {
            sendError('A hallgató nem található', 404);
        }

        // Törlés végrehajtása
        $deleteSql = "DELETE FROM students WHERE id = :id";
        $deleteStmt = $pdo->prepare($deleteSql);
        $deleteStmt->execute([':id' => $id]);

        // Visszaadjuk a törölt hallgató adatait
        sendSuccess(
            $student,
            'Hallgató sikeresen törölve'
        );

    } catch (PDOException $e) {
        sendError('Adatbázis hiba', 500);
    }
}
```

7.15. kódpélda. *Hallgató törlése*

DELETE-kérés példa:

```
DELETE /api/students/1

Valasz (200 OK):
{
    "status": "success",
    "message": "Hallgató sikeresen törölve",
    "data": {
        "id": 1,
        "name": "Nagy János",
        "email": "nagy.janos@example.com",
        "created_at": "2024-01-20 08:15:00",
        "updated_at": "2024-01-20 08:15:00"
    }
}
```

7.7. Biztonság REST API-kban

A REST API-k biztonságos működéséhez elengedhetetlen a hitelesítés és a különböző források közötti (Cross-Origin) kérések megfelelő kezelése.

7.7.1. Bearer token hitelesítés

A REST API-k állapotmentesek, ezért a hagyományos munkamenet-alapú hitelesítés nem ideális megoldás. A tokenalapú hitelesítés az egyik legelterjedtebb módszer, amely során a kliens minden kérésével együtt elküldi a hitelesítő tokent.

Tokenalapú hitelesítés működése

A hitelesítési folyamat lépéseit a 7.4. ábra szemlélteti. A folyamat négy lépésből áll: a kliens elküldi a bejelentkezési adatokat, a szerver visszaadja a tokent, majd a kliens minden további kéréshez csatolja a tokent, a szerver pedig ellenőrzi azt és visszaküldi az adatokat.

Megjegyzés

Miért nem munkamenet-alapú (session) hitelesítés? A munkamenet-alapú megoldás szerveroldali állapotot tárol, ami sérti a REST állapotmentességének elvét. Több szerver használata esetén nehezebben skálázható, és mobilalkalmazásoknál is problémákat okozhat. A tokenalapú hitelesítés ezzel szemben skálázhatóbb és megőrzi az állapotmentes működést.



7.4. ábra. *Tokenalapú hitelesítési folyamat*

Egyszerű tokengenerálás és -validálás

Az alábbi példa egy egyszerű tokenrendszert mutat be oktatási célokra. A token két részből áll: a felhasználó adataiból (user ID és időbélyeg), valamint egy HMAC (Hash-based Message Authentication Code) aláírásból. Az aláírás segítségével észlelhető, ha a token tartalma módosult. A validálás ellenőrzi az aláírást és a token lejárati idejét.

```php
<?php
// A titkos kulcs éles környezetben .env fájlból érkezik
    (lásd a 8. fejezetet)
$_ENV['SECRET_KEY'] ??=
    'csereljuk-le-egy-hosszu-veletlen-kulcsra';

// Egyszerű token generálás - FIGYELEM: Éles környezetben
    használjunk JWT-t!
function generateToken(int $userId): string
{
    $timestamp = time();
    $data = "$userId|$timestamp";

    // Token = base64(data) . '.' . base64(hmac(data))
    $signature = hash_hmac('sha256', $data,
        $_ENV['SECRET_KEY'], true);

    return base64_encode($data) . '.' .
        base64_encode($signature);
}

// Token validálás - visszaadja a user ID-t, vagy null-t ha
    érvénytelen
```

```php
function validateToken(string $token): ?int
{
    $parts = explode('.', $token);

    if (count($parts) !== 2) {
        return null;
    }

    [$data, $signature] = $parts;
    $data = base64_decode($data);
    $signature = base64_decode($signature);

    // Signature ellenőrzés
    $expectedSignature = hash_hmac('sha256', $data,
        $_ENV['SECRET_KEY'], true);

    if (!hash_equals($expectedSignature, $signature)) {
        return null;
    }

    // Adatok kinyerése
    $parts = explode('|', $data);
    if (count($parts) !== 2) {
        return null;
    }

    [$userId, $timestamp] = $parts;

    // Token lejárat ellenőrzése (24 óra)
    if (time() - $timestamp > 86400) {
        return null;
    }

    return (int)$userId;
}

// Authorization header beolvasása, Bearer token kinyerése
function getBearerToken(): ?string
{
    // getallheaders() Apache alatt működik; Nginx/PHP-FPM
        esetén a $_SERVER a megbízható
    $authHeader = getallheaders()['Authorization'] ??
        $_SERVER['HTTP_AUTHORIZATION'] ?? '';

    if (empty($authHeader)) {
        return null;
```

```
    }

    // Elvárt formátum: "Bearer abc123xyz..."
    if (!preg_match('/^Bearer\s+(.+)$/i', $authHeader,
        $matches)) {
        return null;
    }

    return $matches[1];
}

// Hitelesítés megkövetelése - visszaadja a user ID-t, vagy
    hibával leáll
function requireAuth(): int
{
    $token = getBearerToken();

    if (!$token) {
        sendError('Hitelesítési token szükséges', 401);
    }

    $userId = validateToken($token);

    if (!$userId) {
        sendError('Érvénytelen vagy lejárt token', 401);
    }

    return $userId;
}
```

7.16. kódpélda. *Tokenkezelés (helpers.php)*

A `base64_encode()` függvény tetszőleges bináris adatot csak betűkből, számjegyekből és néhány írásjelből álló szöveggé alakít (Base64 kódolás). Erre azért van szükség, mert az aláírás (`$signature`) nyers bináris bájtsorozat, amely nem helyezhető el biztonságosan egy HTTP-fejlécben. A Base64 nem titkosítás: a kódolt adat bárki által visszaalakítható, a token biztonságát kizárólag a HMAC-aláírás adja.

Login végpont implementálása

A login végpont fogadja a felhasználó hitelesítő adatait (e-mail-cím és jelszó), ellenőrzi azokat az adatbázisban, és sikeres hitelesítés esetén visszaad egy tokent. A példában egyszerűsítve csak az e-mail-cím alapján keresünk, valós alkalmazásban természetesen a megadott jelszó ellenőrzése a tárolt jelszóhash alapján is szükséges (`password_verify()` függvény).

```php
// index.php-hez hozzáadva a routing táblázatban
case 'auth':
    handleAuth($pdo, $method);
    break;

function handleAuth(PDO $pdo, string $method): void
{
    if ($method !== 'POST') {
        sendError('A metódus nem engedélyezett', 405);
    }

    login($pdo);
}

function login(PDO $pdo): void
{
    $input = getJsonInput();

    // Kötelező mezők
    if (!isset($input['email']) ||
        !isset($input['password'])) {
        sendError('E-mail cím és jelszó megadása kötelező',
            400);
    }

    $email = trim($input['email']);
    $password = $input['password'];

    try {
        // Felhasználó keresése (példa - normál esetben
            users táblából)
        // Demonstrációs célból a students táblát használjuk
        $sql = "SELECT id, name, email FROM students WHERE
            email = :email LIMIT 1";
        $stmt = $pdo->prepare($sql);
        $stmt->execute([':email' => $email]);
        $user = $stmt->fetch();

        if (!$user) {
            // Biztonsági okokból ugyanazt a hibaüzenetet
                adjuk
            sendError('Hibás e-mail cím vagy jelszó', 401);
        }

        // Token generálása
        $token = generateToken($user['id']);
```

```
        sendSuccess([
            'token' => $token,
            'user' => [
                'id' => $user['id'],
                'name' => $user['name'],
                'email' => $user['email']
            ]
        ], 'Sikeres bejelentkezés');

    } catch (PDOException $e) {
        sendError('Adatbázis hiba', 500);
    }
}
```

7.17. kódpélda. *Token generálása*

Login példa:

```
POST /api/auth
Content-Type: application/json

{
    "email": "nagy.janos@example.com",
    "password": "password123"
}

Válasz (200 OK):
{
    "status": "success",
    "message": "Sikeres bejelentkezés",
    "data": {
        "token": "MXwxNzA2MDgwMDAw.abc123...",
        "user": {
            "id": 1,
            "name": "Nagy János",
            "email": "nagy.janos@example.com"
        }
    }
}
```

Védett végpontok

A hallgató műveleteket úgy módosítjuk, hogy csak hitelesített felhasználók érjék el:

```
function handleStudents(PDO $pdo, string $method, ?string
    $id): void
{
    switch ($method) {
        case 'GET':
            // GET művelet publikus marad
            if ($id) {
                getStudent($pdo, $id);
            } else {
                getStudents($pdo);
            }
            break;
        case 'POST':
            // Létrehozás csak hitelesített felhasználónak
            $userId = requireAuth();
            createStudent($pdo);
            break;
        case 'PUT':
            // Módosítás csak hitelesített felhasználónak
            $userId = requireAuth();
            if (!$id) {
                sendError('Módosításhoz azonosító
                    szükséges', 400);
            }
            updateStudent($pdo, $id);
            break;
        case 'DELETE':
            // Törlés csak hitelesített felhasználónak
            $userId = requireAuth();
            if (!$id) {
                sendError('Törléshez azonosító szükséges',
                    400);
            }
            deleteStudent($pdo, $id);
            break;
        default:
            sendError("A(z) $method metódus nem
                engedélyezett", 405);
    }
}
```

7.18. kódpélda. *Védett végpont példa*

Védett kérés példa:

```
POST /api/students
Content-Type: application/json
Authorization: Bearer MXwxNzA2MDgwMDAw.abc123...

{
    "name": "Új hallgató",
    "email": "uj.hallgato@example.com"
}

Sikeres válasz (201 Created) - ha token érvényes:
{
    "status": "success",
    "message": "Hallgató sikeresen létrehozva",
    "data": {
        "id": 5,
        "name": "Új hallgató",
        ...
    }
}

Hibaválasz (401), ha a token hiányzik vagy érvénytelen:
{
    "status": "error",
    "message": "Hitelesítési token szükséges"
}
```

Megjegyzés

JWT (JSON Web Token) használata éles környezetben:
A fenti, saját készítésű token kizárólag a működési elv bemutatására szolgál. Éles környezetben mindenképpen szabványos JWT-tokent használjunk, amelyet jól bevált, karbantartott könyvtár állít elő és ellenőriz (PHP-ben például a `firebase/php-jwt` csomag). A JWT az RFC 7519 szabványon alapuló, digitálisan aláírt token, amely tartalmazza a felhasználó adatait és a lejárati időt. A hozzá kapcsolódó frissítő token (*refresh token*) mechanizmus hosszabb munkameneteket tesz lehetővé anélkül, hogy a felhasználónak újra be kellene jelentkeznie.

7.7.2. A CORS kezelése

A CORS (Cross-Origin Resource Sharing) egy böngésző által alkalmazott biztonsági mechanizmus, amely szabályozza, hogy egy webalkalmazás milyen feltételekkel férhet hozzá más domainen található erőforrásokhoz. Alapértelme-

zetten a böngészők a *Same-Origin Policy* szabályt alkalmazzák, amely tiltja az eltérő domainről, protokollról vagy portról érkező JavaScript-kéréseket.

CORS-ra akkor van szükség, ha a frontend alkalmazás és az API különböző forrásból működik, például:

- Frontend: `http://localhost:3000`, API: `http://localhost:8080`;
- SPA-alkalmazás (React, Vue, Angular) REST API-t használ;
- Eltérő domainek közötti kommunikáció (`myapp.com` → `api.myapp.com`).

Megjegyzés

A CORS kizárólag böngészőoldali korlátozás. Szerveroldali kliensek (pl. PHP cURL, Postman vagy backend szolgáltatások) nem alkalmazzák ezt a szabályt.

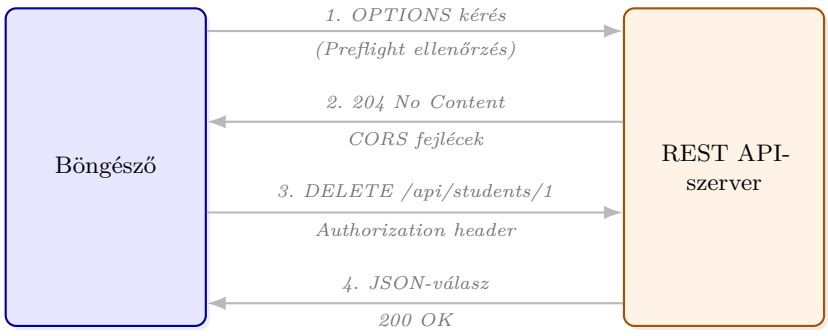
Preflight kérések

Bizonyos HTTP-kérések végrehajtása előtt a böngésző automatikusan egy úgynevezett *preflight* kérést küld `OPTIONS`-metódussal. Ennek célja annak ellenőrzése, hogy a szerver engedélyezi-e a tényleges kérés végrehajtását.

Preflight kérés jellemzően az alábbi esetekben történik:

- `PUT`-, `PATCH`- vagy `DELETE`-metódus használata esetén;
- egyedi HTTP-fejlécek küldésekor (pl. `Authorization`);
- `application/json` típusú kérés esetén.

A 7.5. ábra a preflight folyamat négy lépését mutatja be: a böngésző `OPTIONS`-kérést küld, a szerver `204 No Content` válasszal engedélyezi a kérést, majd a böngésző elküldi a tényleges kérést.



7.5. ábra. *CORS preflight folyamat*

CORS-implementáció PHP-ben

1. Alapvető CORS-konfiguráció fejlesztési környezetben

Fejlesztési környezetben gyakori megoldás az összes origin engedélyezése. Ez leegyszerűsíti a frontend és backend közötti kommunikáció tesztelését.

```php
<?php
// CORS fejlécek beállítása
header('Access-Control-Allow-Origin: *');
header('Access-Control-Allow-Methods: GET, POST, PUT,
    DELETE, OPTIONS');
header('Access-Control-Allow-Headers: Content-Type,
    Authorization');

// Preflight kérés kezelése
if ($_SERVER['REQUEST_METHOD'] === 'OPTIONS') {
    http_response_code(204);
    exit;
}

// ... további API logika
```

7.19. kódpélda. *Egyszerű CORS-konfiguráció (index.php)*

> **Megjegyzés**
>
> Az `Access-Control-Allow-Origin: *` használata kizárólag fejlesztési környezetben ajánlott. Éles rendszerben célszerű explicit módon megadni az engedélyezett domaineket, különösen hitelesített API-k esetén.

2. Szigorúbb CORS-konfiguráció (éles környezet)

Éles környezetben célszerű *whitelist*-alapú megközelítést alkalmazni, ahol kizárólag a megbízható domainekről érkező kérések engedélyezettek. A szerver az `Origin` fejléc értékét ellenőrzi, és csak az engedélyezett források esetén küldi vissza a megfelelő CORS-fejléceket.

```php
<?php
// CORS fejlécek beállítása whitelist alapján
function setCorsHeaders(array $allowedOrigins = []): void
{
    // Alapértelmezett engedélyezett domainek
    if (empty($allowedOrigins)) {
        $allowedOrigins = [
            'http://localhost:3000',
            'http://localhost:8080',
```

```
            'https://myapp.com',
            'https://www.myapp.com'
        ];
    }

    // Origin fejléc ellenőrzése
    $origin = $_SERVER['HTTP_ORIGIN'] ?? '';

    if (in_array($origin, $allowedOrigins, true)) {
        header("Access-Control-Allow-Origin: $origin");
    }

    // További CORS fejlécek
    header('Access-Control-Allow-Methods: GET, POST, PUT,
        DELETE, OPTIONS');
    header('Access-Control-Allow-Headers: Content-Type,
        Authorization, X-Requested-With');
    header('Access-Control-Max-Age: 86400'); // 24 óra

    // Hitelesített kérések esetén:
    // header('Access-Control-Allow-Credentials: true');

    // Preflight kérés kezelése
    if ($_SERVER['REQUEST_METHOD'] === 'OPTIONS') {
        http_response_code(204);
        exit;
    }
}

// index.php:
setCorsHeaders([
    'http://localhost:3000',
    'https://myapp.com'
]);
```

7.20. kódpélda. *Biztonságos CORS-konfiguráció (helpers.php)*

Megjegyzés

Ha az `Access-Control-Allow-Credentials: true` fejléc engedélyezett, akkor az `Access-Control-Allow-Origin` nem lehet `*`, hanem konkrét domaint kell megadni.

7.7.3. Alapvető biztonsági elvek

A Bearer tokenek és a CORS megfelelő kezelése mellett számos általános biztonsági elvet is érdemes figyelembe venni REST API-k fejlesztése során.

- Éles környezetben ajánlott HTTPS használata, mivel HTTP-kapcsolaton keresztül a tokenek és az adatok titkosítatlanul továbbítódnak, így lehallgathatók vagy módosíthatók.
- A kliens által küldött adatokban nem szabad megbízni, ezért minden bemenetet szerveroldalon validálni kell.
- Az adatbázis-műveletek során ajánlott előkészített utasításokat használni az SQL-injection-támadások megelőzése érdekében.

Megjegyzés

Rate limiting: Éles környezetben célszerű korlátozni az egy kliensről vagy IP-címről érkező kérések számát (például 100 kérés/perc), ezzel csökkentve a brute-force és túlterheléses támadások kockázatát. A korlátozás megvalósítható webszerver szinten (pl. Nginx), reverse proxy segítségével vagy dedikált könyvtárakkal (pl. `symfony/rate-limiter`).

7.8. REST API-tesztelés

A REST API-k tesztelése eltér a hagyományos webalkalmazások tesztelésétől, mivel nem csupán HTML-űrlapokkal kommunikálunk. Egy klasszikus HTML-űrlap alapértelmezetten csak GET- és POST-metódusokat támogat, míg REST API-k esetén gyakran szükség van PUT-, PATCH- és DELETE-kérések küldésére is. Emiatt speciális tesztelőeszközök használata szükséges.

7.8.1. cURL használata PHP-ben

A cURL egy URL-alapú adatátvitelt támogató parancssori eszköz és programkönyvtár, amely számos hálózati protokoll kezelésére alkalmas. PHP beépített cURL-támogatással rendelkezik, így HTTP-kérések egyszerűen küldhetők az API felé.

A fejezet cURL-példái a GitHub-tároló `curl_test.php` fájljában találhatók. Futtatásuk előtt el kell indítani az API-szervert a projekt gyökérkönyvtárából:

```
php -S localhost:8080 api/index.php
```

A PHP beépített webszerverének paraméterként megadott fájl úgynevezett *router* szkriptként működik: a szerver minden beérkező HTTP-kérésre ezt a fájlt futtatja le, így az API front controllere minden kérést megkap.

A legfontosabb PHP cURL függvények:

- `curl_init()`: cURL munkamenet inicializálása;
- `curl_setopt()`: cURL opciók beállítása;
- `curl_exec()`: a kérés végrehajtása;
- `curl_getinfo()`: információk lekérése a kérésről;
- `curl_close()`: a munkamenet lezárása. PHP 8.0 óta a cURL munkamenet objektum, amelyet a PHP automatikusan felszabadít, ezért a függvény hívása felesleges, PHP 8.5-től pedig elavultnak (*deprecated*) minősül.

Példa cURL használatára POST-kérés esetén:

```php
<?php
// Inicializálás
$ch = curl_init();

// Küldendő adatok
$data = [
    "name" => "Nagy Péter",
    "email" => "nagy.peter@example.com"
];

// cURL opciók beállítása
curl_setopt($ch, CURLOPT_URL,
    "http://localhost:8080/api/students");
curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
curl_setopt($ch, CURLOPT_POST, true);
curl_setopt($ch, CURLOPT_POSTFIELDS, json_encode($data));
curl_setopt($ch, CURLOPT_HTTPHEADER, [
    'Content-Type: application/json',
    'Authorization: Bearer abc123xyz...'
]);

// Kérés végrehajtása
$response = curl_exec($ch);
$httpCode = curl_getinfo($ch, CURLINFO_HTTP_CODE);

// Hibakezelés
if (curl_errno($ch)) {
```

```
    echo 'cURL hiba: ' . curl_error($ch);
} else {
    echo "HTTP kód: $httpCode\n";
    $result = json_decode($response, true);
    print_r($result);
}
```

7.21. kódpélda. *cURL POST-kérés JSON-adatokkal*

7.8.2. Grafikus API-tesztelő eszközök

A grafikus felületű API-tesztelő eszközök megkönnyítik a REST API-k manuális tesztelését, hibakeresését és fejlesztés közbeni ellenőrzését.

Postman

A Postman[2] az egyik legismertebb API-fejlesztő és -tesztelő eszköz, amely REST API-k tesztelésére, automatizálására és dokumentálására is alkalmas.

Főbb funkciók:

- Grafikus felületű API-kliens Windows-, macOS- és Linux-rendszerekre;
- Az összes elterjedt HTTP-metódus támogatása (GET, POST, PUT, DELETE, PATCH stb.);
- Kérések mentése gyűjteményekbe (Collections);
- Környezetek kezelése (development, testing, production);
- Változók használata (pl. `{{base_url}}`, `{{token}}`);
- Automatizált tesztek készítése JavaScript segítségével;
- API-dokumentáció generálása;
- Kérések és gyűjtemények megosztása csapattagok között.

Alapvető használat:

1. A kívánt HTTP-metódus kiválasztása (GET, POST stb.);
2. Az API-végpont címének megadása (pl. `http://localhost:8080/api/students`);
3. A `Headers` fülön a megfelelő fejlécek beállítása, például `Content-Type: application/json`;

[2] https://www.postman.com

4. A `Body` fülön a `raw` és `JSON` opció kiválasztása;
5. A küldendő JSON-adatok megadása;
6. A kérés elküldése a `Send` gombbal;
7. A válasz ellenőrzése az alsó panelen (státuszkód, fejlécek, válasz törzse).

A kéréseket gyűjteménybe (*Collection*) is menthetjük: `New` → `Collection`, majd a kéréseket a `Save` gombbal adjuk hozzá. A kész gyűjtemény az `Export` menüponttal JSON-fájlba menthető, és egy másik gépen az `Import` menüponttal tölthető be. Az ismétlődő értékeket, például az API címét vagy a tokent, a gyűjtemény változóiban (`{{base_url}}`, `{{token}}`) érdemes tárolni.

Insomnia

Az Insomnia[3] egy modern API-kliens, amely egyszerűbb és letisztultabb alternatívát kínál a Postman mellett.

Főbb funkciók:

- Minimalista és könnyen áttekinthető felület;
- REST és GraphQL API-k támogatása;
- Nyílt forráskódú és ingyenesen használható;
- Kódrészletek generálása különböző programozási nyelvekre (PHP, JavaScript, Python stb.);
- Környezetek és változók kezelése.

7.9. Gyakorlófeladatok

A következő feladatok egy részvénykatalógus API köré épülnek. A részvény (*stock*) egy tőzsdén jegyzett vállalat tulajdonrészét megtestesítő értékpapír, amelyet egy rövid betűkód, a *ticker* azonosít (például `AAPL` az Apple esetében). A szektor (*sector*) a vállalatok tevékenységi kör szerinti csoportja, például technológia vagy pénzügy. A feladatok adatbázisa:

```
CREATE TABLE sectors (
    id   INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(100) NOT NULL UNIQUE
);
```

[3] https://insomnia.rest/

```
CREATE TABLE stocks (
    id           INT AUTO_INCREMENT PRIMARY KEY,
    ticker       VARCHAR(10)   NOT NULL UNIQUE,
    company_name VARCHAR(100)  NOT NULL,
    sector_id    INT,
    price        DECIMAL(10,2) NOT NULL,
    created_at   DATETIME      NOT NULL DEFAULT
       CURRENT_TIMESTAMP,
    FOREIGN KEY (sector_id) REFERENCES sectors(id)
);
INSERT INTO sectors (name) VALUES
    ('Technology'), ('Finance'), ('Healthcare'), ('Energy');
INSERT INTO stocks (ticker, company_name, sector_id, price)
    VALUES
    ('AAPL',  'Apple Inc.',        1, 189.30),
    ('MSFT',  'Microsoft Corp.',   1, 415.50),
    ('GOOGL', 'Alphabet Inc.',     1, 175.20),
    ('JPM',   'JPMorgan Chase',    2, 198.40),
    ('JNJ',   'Johnson & Johnson', 3, 147.80);
```

Minden feladatnál javasolt az előkészített utasítások használata és az egységes JSON-válaszformátum alkalmazása (`status`, `data`, `message` mezőkkel).

1. Tervezzük meg egy részvénykatalógus API-végpontjait. Készítsünk egy táblázatot, amely minden végponthoz megadja a HTTP-metódust, az URL-t, a várható státuszkódokat (sikeres és hibás esetekben), valamint egy rövid leírást. A rendszer kezelje a részvényeket (`/api/stocks`) és a szektorokat (`/api/sectors`). A módosító műveletek legyenek hitelesítéshez kötve.

2. Hozzunk létre egy `index.php` fájlt, amely adatbázis nélkül, hardkódolt tömbből szolgálja ki a `GET /api/stocks` és `GET /api/stocks/{id}` végpontokat. A válaszokat `json_encode()` segítségével küldjük el, a `Content-Type` fejlécet állítsuk `application/json`-ra, és használjunk helyes státuszkódokat: sikeres válasznál 200-at, illetve 404-et, ha az adott azonosítóhoz nem tartozik rekord.

3. Csatlakoztassuk az API-t a fenti adatbázishoz. Hozzuk létre a sémát, töltsük be a mintaadatokat, majd implementáljuk a CRUD-műveleteket:

   - `GET /api/stocks` és `GET /api/stocks/{id}` – listázás és egyedi lekérés;
   - `POST /api/stocks` – új részvény létrehozása (201). Validáljuk a tickert, a nevet és az árat. Érvénytelen adatok esetén 422-es választ adjunk, mezőnként lebontott hibaüzenetekkel.

- `PUT /api/stocks/{id}` – teljes rekord frissítése;
- `PATCH /api/stocks/{id}` – csak az ár (`price`) frissítése;
- `DELETE /api/stocks/{id}` – törlés, 204-es válasszal.

A kérés törzsét a `php://input` adatfolyamból olvassuk be, majd dekódoljuk JSON-formátumban.

4. Bővítsük a `GET /api/stocks` végpontot szűréssel, rendezéssel és lapozással:

   - `$_GET['sector']` – szektor szerinti szűrés;
   - `$_GET['sort']` – ár szerint rendezés (`price_asc` / `price_desc`);
   - `$_GET['page']`, `$_GET['limit']` – lapozás.

   A válasz tartalmazza a lapozáshoz szükséges metaadatokat is: aktuális oldal, oldalméret, összes rekord száma, valamint az oldalak száma.

5. Adjunk hozzá Bearer token-alapú hitelesítést. Valósítsuk meg a `POST /api/auth` végpontot, amely érvényes hitelesítő adatok esetén tokent ad vissza. A `GET`-végpontok maradjanak nyilvánosak; a `POST`-, `PUT`-, `PATCH`- és `DELETE`-végpontok csak érvényes `Authorization: Bearer {token}` fejléc esetén legyenek elérhetők, egyébként 401-es választ adjanak.

6. Végezzük el az alábbi két feladatot:

   - Állítsuk be a CORS-fejléceket úgy, hogy csak a `http://localhost:3000` forrásból érkező kérések legyenek engedélyezve, és kezeljük a preflight `OPTIONS`-kéréseket 204-es válasszal.
   - Készítsünk egy Postman-gyűjteményt, amely tartalmazza az összes végpontot. A módosító kérésekhez adjuk meg az `Authorization` fejlécet is.

8. FEJEZET

COMPOSER ÉS CSOMAGKEZELÉS

A Composer a PHP-ökoszisztéma szabványos függőségkezelője, amely lehetővé teszi külső könyvtárak és csomagok egyszerű telepítését, frissítését és kezelését (Composer Contributors 2026). A modern PHP-fejlesztés gyakorlatilag elképzelhetetlen Composer nélkül, mivel a legtöbb keretrendszer és könyvtár ezen keresztül telepíthető és frissíthető. A Composer az autoloadingot, a verziókezelést és a projekt reprodukálhatóságát is egységes keretbe foglalja.

8.1. Composer-alapok

8.1.1. Csomagok keresése a Packagist oldalon

A legtöbb nyilvános PHP-csomag a *Packagist* oldalon érhető el: https://packagist.org. A csomagok kereshetők név, kulcsszó vagy kategória alapján. Egy csomag adatlapján megtalálható:

- az elérhető verziók és a kiadási előzmények;
- a telepítési parancs (`composer require vendor/package`);
- a csomag függőségei;
- a letöltési statisztikák és a GitHub-hivatkozás.

A csomag kiválasztásánál érdemes figyelni a letöltések számára, a karbantartottságra (utolsó frissítés dátuma) és arra, hogy van-e aktív fejlesztői közösség mögötte.

8.1.2. Telepítés

Windows rendszeren a legegyszerűbb módszer a hivatalos telepítő használata:

1. A telepítőt letöltjük a hivatalos oldalról;[1]

[1] https://getcomposer.org

2. A telepítő futtatása után megadjuk a PHP futtatható fájl elérési útját;
3. A telepítő automatikusan beállítja a PATH környezeti változót.

A telepítés után a Composer elérhetővé válik a parancssorból, amit a verzióellenőrzéssel tudunk igazolni.

Megjegyzés

XAMPP esetén a telepítőnek a `C:\xampp\php\php.exe` fájlra kell mutatnia. Linux- és macOS-rendszereken a telepítés parancssorból történik (`curl -sS https://getcomposer.org/installer | php`). Részletes útmutató: https://getcomposer.org.

A telepítés sikerességét a következő paranccsal ellenőrizzük, amely kiírja a Composer verziószámát:

```
composer --version
```

8.2. Projekt inicializálása

8.2.1. Új projekt létrehozása

Egy új PHP-projekt Composerrel történő inicializálása a következő lépésekben történik:

```
mkdir my-project
cd my-project
composer init
```

A `composer init` parancs interaktív módon végigvezet a projekt alapvető beállításain:

- **Package name**: a csomag neve (pl. `vendor/project-name`);
- **Description**: rövid leírás;
- **Author**: szerző neve és e-mail-címe;
- **Minimum Stability**: minimális stabilitási szint (dev, alpha, beta, RC, stable);
- **Package Type**: csomag típusa (library, project stb.);
- **License**: licenc típusa (MIT, GPL stb.).

8.2.2. A composer.json fájl

A `composer.json` a projekt központi konfigurációs fájlja, amely a függőségeket és a projekt beállításait tartalmazza:

```
{
    "name": "myvendor/my-project",
    "description": "Egy minta PHP projekt",
    "type": "project",
    "require": {
        "php": ">=8.1",
        "monolog/monolog": "^3.0"
    },
    "require-dev": {
        "phpunit/phpunit": "^10.0"
    },
    "autoload": {
        "psr-4": {
            "App\\": "src/"
        }
    }
}
```

8.1. kódpélda. *composer.json*

A `composer.json` fájl több fontos metaadatot tartalmaz, amelyek a projekt azonosítását és működését határozzák meg. A leggyakoribb mezők a következők:

- `require`: az éles (production) környezetben szükséges függőségek;
- `require-dev`: fejlesztéshez szükséges függőségek (pl. tesztelés, debug eszközök);
- `autoload`: az automatikus osztálybetöltés konfigurációja.

Ezek a mezők együtt írják le a projekt alapvető metaadatait és függőségi rendszerét.

8.2.3. A composer.lock fájl

A `composer.lock` a Composer működésének egyik legfontosabb eleme, mivel a reprodukálhatóságot biztosítja a különböző környezetek között. Ez a fájl rögzíti a telepített csomagok pontos verzióit:

- Automatikusan generálódik a `composer install` vagy `composer update` parancsok futtatásakor;

- Biztosítja, hogy minden fejlesztői környezet és az éles rendszer ugyanazokat a csomagverziókat használja;
- Alkalmazások esetén a verziókövetés része.

Megjegyzés

- **Alkalmazások** esetén a `composer.lock` fájlt verziókövetésbe tesszük. Ez biztosítja, hogy a fejlesztői és az éles környezet pontosan ugyanazokat a csomagverziókat használja.
- **Könyvtárak** (library típusú csomagok) esetén a `composer.lock` fájl általában nem kerül verziókövetésbe, mivel a függőségek pontos verzióját a könyvtárat használó projekt határozza meg.

8.3. Csomagok kezelése

8.3.1. Csomag telepítése

Új csomag hozzáadása a projekthez a következő paranccsal történik:

```
composer require monolog/monolog
```

A `composer require` parancs nemcsak egy csomagot telepít, hanem automatikusan kezeli annak teljes függőségi gráfját is. A folyamat során a Composer a következő lépéseket hajtja végre:

1. Letölti a csomagot és annak függőségeit a `vendor` mappába;
2. Frissíti a `composer.json` fájlt;
3. Frissíti a `composer.lock` fájlt;
4. Újragenerálja az autoloadert.

Fejlesztői függőségek telepítése

Fejlesztéshez szükséges csomagok (például tesztelési keretrendszerek) a következő módon adhatók hozzá:

```
composer require --dev phpunit/phpunit
```

8.3.2. Verziójelölések

A Composer a szemantikus verziózást (Semantic Versioning) használja.[2] A verziószám három részből áll: `MAJOR.MINOR.PATCH`, ahol a MAJOR visszafelé inkompatibilis változásokat, a MINOR új funkciókat, a PATCH pedig hibajavításokat jelöl.

A függőségek verziókövetelményeit különböző jelölésekkel adhatjuk meg:

- `^3.0`: kompatibilis a 3.x verzióval (>=3.0.0, <4.0.0);
- `~3.5`: minor szintű frissítéseket enged (>=3.5.0, <4.0.0). Ha három komponenst adunk meg (`~3.5.0`), akkor csak patch szintű frissítéseket enged (>=3.5.0, <3.6.0);
- `3.0.5`: pontosan ez a verzió;
- `>=3.0`: legalább 3.0-s verzió;
- `*`: tetszőleges verzió. Használata általában kerülendő.

8.3.3. Csomagok frissítése

A telepített csomagok frissítése vagy újratelepítése a következő parancsokkal történik:

```
# Összes csomag frissítése
composer update

# Egy konkrét csomag frissítése
composer update monolog/monolog

# Telepítés a lock fájl alapján (éles környezetben)
composer install
```

A két parancs között lényeges különbség van: a `composer update` frissíti a csomagokat és újraírja a `composer.lock` fájlt, míg a `composer install` a meglévő lock fájl alapján telepít. Éles környezetben mindig ez utóbbit célszerű használni.

8.3.4. Csomag eltávolítása

Egy csomag eltávolítása a `composer remove` paranccsal történik:

```
composer remove monolog/monolog
```

[2] https://semver.org/

8.4. Automatikus osztálybetöltés

Az automatikus osztálybetöltés (*autoloading*) lehetővé teszi, hogy a PHP automatikusan betöltse az osztályokat, így nincs szükség minden fájlhoz külön `require` utasításra.

8.4.1. PSR-4 autoloading

A PSR-4 a PHP szabványos autoloading specifikációja (PHP-FIG 2013). A helyes működéshez elengedhetetlen a névterek és a könyvtárstruktúra konzisztens használata. A konfiguráció a `composer.json` fájlban történik:

```
{
    "autoload": {
        "psr-4": {
            "App\\": "src/"
        }
    }
}
```

8.2. kódpélda. *PSR-4 autoload konfiguráció*

Ennek alapján a következő megfeleltetés érvényes:

- `App\User` → `src/User.php`;
- `App\Controllers\HomeController` → `src/Controllers/HomeController.php`;
- `App\Models\Product` → `src/Models/Product.php`.

8.4.2. Autoloader használata

Az autoloader használatához egyetlen `require` utasítás szükséges:

```
<?php
require __DIR__ . '/vendor/autoload.php';

// Ezt követően az osztályok automatikusan betöltődnek
use App\Controllers\HomeController;
use App\Models\User;

$controller = new HomeController();
$user = new User();
```

8.3. kódpélda. *Autoloader betöltése*

Amennyiben az autoload beállítások módosulnak a `composer.json` fájlban, az autoloader újragenerálása szükséges:

```
composer dump-autoload
```

Éles környezetben az `--optimize` kapcsoló használatával optimalizált, classmapalapú autoloader generálható. Ez egy előre elkészített osztálynév–fájlnév megfeleltetési táblázatból keresi ki, hogy egy osztály definíciója melyik fájlban található, így nem kell minden betöltéskor a PSR-4-szabály alapján az útvonalat kiszámítani és a fájl létezését ellenőrizni:

```
composer dump-autoload --optimize
```

8.5. Composer-szkriptek

A Composer lehetővé teszi saját parancsok definiálását a `scripts` kulcs alatt a `composer.json` fájlban. Ezek a parancsok a `composer <nev>` alakban futtathatók, és kényelmes rövidítésként szolgálnak a fejlesztési munkafolyamatban.

```
"scripts": {
    "start": "php -S localhost:8000 -t public",
    "test":  "phpunit --testdox"
}
```

8.4. kódpélda. *Composer-szkriptek konfigurációja*

A `start` szkript a beépített PHP-fejlesztői szervert indítja el a `public/` mappából kiszolgálva, a `test` szkript pedig a PHPUnit tesztelőt hívja meg. Használatuk:

```
composer start
composer test
```

8.6. Projektstruktúra

Egy Composer-alapú PHP-projekt általában jól elkülönített könyvtárstruktúrát követ, amely támogatja a skálázhatóságot és az átláthatóságot. A következőkben egy tipikus projektfelépítés látható:

my-project/

```
|-- src/
|   |-- Controllers/
|   |   +-- HomeController.php
|   |-- Models/
|   |   +-- User.php
|   +-- Services/
|       +-- EmailService.php
|-- tests/
|   +-- UserTest.php
|-- logs/
|   +-- app.log          (alkalmazás hozza létre)
|-- vendor/           (automatikusan generált)
|-- public/
|   +-- index.php     (belépési pont)
|-- composer.json
|-- composer.lock
+-- .gitignore
```

A konkrét könyvtárszerkezet projektenként eltérhet, de a Composer-alapú alkalmazások általában hasonló felépítést követnek. A `logs/` mappa például nem minden projektben szerepel, mivel azt az alkalmazás futás közben hozza létre.

8.6.1. A vendor mappa

A `vendor` mappa tartalmazza az összes telepített függőséget, valamint az automatikusan generált autoloadert (`vendor/autoload.php`).

Mivel a mappa a Composer működéséből automatikusan újragenerálható, szükség esetén törölhető, majd a `composer install` paranccsal visszaállítható.

A `vendor` mappát nem tesszük verziókövetésbe. Ehelyett a `.gitignore` fájlban kizárjuk, mivel a függőségek a `composer.lock` alapján bármikor újratelepíthetők.

8.6.2. Környezeti változók és `.env` fájlok

Az adatbázis-jelszavak, API-kulcsok és SMTP-hitelesítő adatok érzékeny konfigurációs adatok. Ezeket nem szabad közvetlenül a forráskódban tárolni, mert így bekerülnek a verziókövető rendszerbe, és minden fejlesztő (vagy rosszabb esetben nyilvános repository esetén bárki) hozzáférhet hozzájuk.

Modern PHP-alkalmazásokban elterjedt megoldás a `.env` fájl használata, amely a projekt gyökérkönyvtárában helyezkedik el, és a környezetfüggő

konfigurációs értékeket tartalmazza:

```
DB_HOST=localhost
DB_NAME=myapp
DB_USER=root
DB_PASS=secret

SMTP_HOST=smtp.example.com
SMTP_USER=user@example.com
SMTP_PASS=password
```

8.5. kódpélda. *.env fájl példa*

A `.env` fájlt fel kell venni a `.gitignore` fájlba, hogy ne kerüljön verziókövetésbe. Ehelyett egy `.env.example` fájlt érdemes verziókövetésbe tenni, amely a szükséges változókat tartalmazza értékek nélkül:

```
DB_HOST=
DB_NAME=
DB_USER=
DB_PASS=
```

8.6. kódpélda. *.env.example fájl*

A `.env` fájl beolvasása PHP-ben

A `.env` fájl tartalmát a `vlucas/phpdotenv` Composer-csomag tölti be automatikusan:

```
composer require vlucas/phpdotenv
```

A csomag betöltése és inicializálása az alkalmazás belépési pontján (pl. `index.php`) történik:

```
<?php

require_once __DIR__ . '/vendor/autoload.php';

$dotenv = Dotenv\Dotenv::createImmutable(__DIR__);
$dotenv->load();
```

8.7. kódpélda. *.env betöltése phpdotenv segítségével*

Ezt követően a változók elérhetők a `$_ENV` vagy a `$_SERVER` szuperglobális tömbön keresztül. A `getenv()` függvény csak a `createUnsafeImmutable()` használatakor látja őket:

```
$dbHost = $_ENV['DB_HOST'];
$dbName = $_SERVER['DB_NAME'];
```

8.8. kódpélda. *Környezeti változók olvasása*

A PDO-kapcsolat így már nem tartalmaz érzékeny adatot a forráskódban:

```
<?php

$pdo = new PDO(
    dsn: "mysql:host={$_ENV['DB_HOST']};dbname={$_ENV['
        DB_NAME']};charset=utf8mb4",
    username: $_ENV['DB_USER'],
    password: $_ENV['DB_PASS'],
    options: [
        PDO::ATTR_ERRMODE            => PDO::
            ERRMODE_EXCEPTION,
        PDO::ATTR_DEFAULT_FETCH_MODE => PDO::FETCH_ASSOC,
        PDO::ATTR_EMULATE_PREPARES   => false,
    ]
);
```

8.9. kódpélda. *PDO-kapcsolat .env alapján*

Megjegyzés

Fejlesztői és éles környezet: fejlesztői gépen a `.env` fájlt manuálisan hozzuk létre a `.env.example` alapján. Éles (production) szerveren a környezeti változókat általában nem `.env` fájlból, hanem közvetlenül a szerver vagy a konténer konfigurációjából töltik be (pl. Docker, rendszer környezeti változók), mivel a `.env` fájl az éles szerveren is biztonsági kockázatot jelenthet, ha véletlenül elérhetővé válik.

8.7. Gyakran használt parancsok

A 8.1. táblázat összefoglalja a fejlesztés során leggyakrabban előforduló Composer-parancsokat és azok leírását.

8.1. táblázat. *Composer-parancsok*

Parancs	Leírás
`composer init`	Új projekt inicializálása
`composer require vendor/package`	Csomag telepítése
`composer require --dev vendor/package`	Fejlesztői függőség telepítése
`composer remove vendor/package`	Csomag eltávolítása
`composer install`	Telepítés a lock fájl alapján (éles környezetben: `--no-dev` és `--optimize-autoloader`)
`composer update`	Csomagok frissítése
`composer dump-autoload`	Autoloader újragenerálása
`composer show`	Telepített csomagok listázása
`composer outdated`	Elavult csomagok megjelenítése
`composer audit`	Biztonsági sérülékenységek ellenőrzése
`composer self-update`	A Composer frissítése

8.8. Tipikus hibák

Az alábbi hibák különösen gyakran fordulnak elő kezdő fejlesztők körében, és hosszabb távon komoly problémákhoz vezethetnek a projekt karbantarthatóságában és reprodukálhatóságában.

- A `vendor` mappa verziókövetésbe helyezése. Ez felesleges, és jelentősen növeli a repository méretét, miközben a tartalma bármikor újragenerálható.
- A `composer.lock` fájl kihagyása alkalmazások esetén. Ennek hiányában különböző környezetek eltérő függőségverziókat használhatnak.
- A `composer update` használata éles környezetben. Ez nem a lock fájl alapján telepít, hanem frissíti a függőségeket, ami váratlan verzióváltozásokat okozhat.
- Hibás PSR-4-megfeleltetés (névtér és mappastruktúra eltérése). Ilyenkor az autoloader nem tudja betölteni az osztályokat, ami futási hibákhoz vezet.

8.9. Gyakorlati példák

Ez az alfejezet két konkrét könyvtáron keresztül mutatja be a Composer használatát: az első egy naplózó alkalmazást épít fel a Monolog könyvtárral, a második e-mail-küldést valósít meg a PHPMailer segítségével.

8.9.1. Naplózás Monolog könyvtárral

Ebben a példában egy egyszerű projektet hozunk létre, amely a Monolog könyvtárat használja naplózásra. A cél egy minimális, de jól strukturált Composer-alapú alkalmazás felépítésének bemutatása.

Projekt létrehozása

```
mkdir logger-demo
cd logger-demo
composer init --name="myvendor/logger-demo" --type="project" -n
composer require monolog/monolog
```

Autoload beállítása

A `composer.json` fájlban beállítjuk az autoloadingot:

```
{
    "name": "myvendor/logger-demo",
    "type": "project",
    "require": {
        "monolog/monolog": "^3.0"
    },
    "autoload": {
        "psr-4": {
            "App\\": "src/"
        }
    }
}
```

8.10. kódpélda. *composer.json (autoload beállítással)*

Alkalmazáskód

Az alkalmazás két fájlból áll: a `Logger` egy csomagoló osztály (*wrapper class*), amely egyszerűbb felületet ad a Mononloghoz, a`public/index.php` belépési pont pedig ezt használja a naplóbejegyzések írásához.

```php
<?php

namespace App;

use Monolog\Logger as MonologLogger;
use Monolog\Handler\StreamHandler;
use Monolog\Level;

class Logger
{
    private MonologLogger $logger;

    public function __construct(string $name = 'app')
    {
        $this->logger = new MonologLogger($name);
        $this->logger->pushHandler(
            new StreamHandler(__DIR__ . '/../logs/app.log',
                Level::Debug)
        );
    }

    public function info(string $message): void
    {
        $this->logger->info($message);
    }

    public function error(string $message): void
    {
        $this->logger->error($message);
    }
}
```

8.11. kódpélda. *Logger osztály*

A konstruktor egy `StreamHandler` kezelőt csatol a naplózóhoz, amely a bejegyzéseket a `logs/app.log` fájlba írja `Debug` szinttől kezdve. Az `info()` és `error()` metódusok az azonos nevű Monolog-metódusokra delegálnak – további szintekhez (`warning`, `critical` stb.) hasonló metódusok adhatók hozzá.

```php
<?php

require __DIR__ . '/../vendor/autoload.php';

use App\Logger;
```

```php
// A logs mappa létrehozása, ha még nem létezik
if (!is_dir(__DIR__ . '/../logs')) {
    mkdir(__DIR__ . '/../logs', 0755, true);
}

$logger = new Logger();
$logger->info('Az alkalmazás elindult');
$logger->error('Példa hibaüzenet');

echo "Naplózás sikeres. Ellenőrizd a logs/app.log fájlt.";
```

8.12. kódpélda. *Belépési pont (public/index.php)*

A `php public/index.php` parancs futtatása után a `logs/app.log` fájl tartalmazza a naplóbejegyzéseket.

8.9.2. E-mail-küldés PHPMailerrel

Az e-mail-küldés az egyik leggyakoribb Composer-csomag használati esete. A PHP beépített `mail()` függvénye egyszerű esetekben elegendő lehet, de megbízható SMTP-alapú küldéshez a `phpmailer/phpmailer` csomag az egyik legelterjedtebb megoldás.

```
composer require phpmailer/phpmailer
```

A telepítés után a PHPMailer az autoloaderen keresztül érhető el. A következő példa SMTP-n keresztül küld e-mailt: megadja a szerver beállításait, a feladót és a címzettet, majd elküldi az üzenetet.

```php
use PHPMailer\PHPMailer\PHPMailer;
use PHPMailer\PHPMailer\Exception;

$mail = new PHPMailer(true);
$mail->CharSet    = PHPMailer::CHARSET_UTF8;
$mail->isSMTP();
$mail->Host       = 'smtp.example.com';
$mail->SMTPAuth   = true;
$mail->Username   = 'user@example.com';
$mail->Password   = 'secret';
$mail->SMTPSecure = PHPMailer::ENCRYPTION_STARTTLS;
$mail->Port       = 587;

$mail->setFrom('noreply@example.com', 'Weboldal');
$mail->addAddress('recipient@example.com');
$mail->Subject = 'Tárgy';
$mail->Body    = 'Az üzenet szövege.';
```

```php
try {
    $mail->send();
} catch (Exception $e) {
    echo "Hiba az e-mail küldésekor: " . $mail->ErrorInfo;
}
```

8.13. kódpélda. *E-mail-küldés PHPMailerrel*

Az SMTP-beállítások érzékeny adatokat tartalmaznak, ezért ezeket ajánlott a forráskódtól elkülönítve, például `.env` fájlban tárolni.

8.10. Gyakorlófeladatok

A következő feladatok egyetlen Composer-alapú projekt köré épülnek, amelyet fokozatosan bővítünk. A feladatok egymásra épülnek, ezért célszerű sorrendben megoldani őket.

1. Hozzunk létre egy új Composer-projektet a `composer init` paranccsal, majd telepítsük a `nesbot/carbon` csomagot. Készítsünk egy `public/index.php` szkriptet, amely:

   - kiírja a mai dátumot és a következő nap dátumát;
   - megjeleníti, hány nap van hátra az év végéig;
   - formázott dátumot jelenít meg magyar nyelvi beállításokkal.

   Végül adjuk hozzá a `vendor` mappát a `.gitignore` fájlhoz, hogy ne kerüljön verziókövetésbe.

2. Bővítsük a projektet saját osztályokkal. Ehhez konfiguráljuk a PSR-4-autoloadingot az `App\` névtérrel, amely a `src/` mappára mutat. Hozzuk létre a következő osztályokat:

   - `src/Models/Event.php` – esemény osztály (cím, helyszín, dátum `Carbon`-objektumként);
   - `src/Services/Calendar.php` – naptárkezelő, amely eseményeket tárol és közelgő eseményeket listáz.

   A `public/index.php` fájlban hozzunk létre néhány teszteseményt, majd jelenítsük meg azokat dátum szerint rendezve. Az autoload konfiguráció módosítása után futtassuk a `composer dump-autoload` parancsot.

3. Telepítsük fejlesztői függőségként a `symfony/var-dumper` csomagot, majd módosítsuk az `index.php` fájlt úgy, hogy az eseménylista tartalmát a `dump()` függvénnyel jelenítse meg.

Ezt követően vizsgáljuk meg a fejlesztői és éles telepítés közötti különbséget:

- töröljük a `vendor` mappát;
- futtassuk a `composer install --no-dev` parancsot.

Figyeljük meg, hogy a `var-dumper` csomag ebben az esetben nem kerül telepítésre. Ez jól szemlélteti a `require` és `require-dev` mezők, valamint a `composer install` és `composer update` parancsok közötti különbséget.

4. Egészítsük ki a `composer.json` fájlt egy `scripts` mezővel, amely elindítja a beépített PHP-fejlesztői webszervert:

```
"scripts": {
    "start": "php -S localhost:8000 -t public"
}
```

Indítsuk el a projektet a `composer start` paranccsal, majd ellenőrizzük a működést a böngészőben. Ezt követően futtassuk a `composer audit` parancsot, és értelmezzük az eredményét.

5. Egészítsük ki a projektet e-mail-küldési funkcionalitással:

- Telepítsük a `phpmailer/phpmailer` csomagot;
- Hozzunk létre egy `.env` fájlt az SMTP-beállításokkal (host, felhasználónév, jelszó), és adjuk hozzá a `.gitignore` fájlhoz;
- Írjunk egy `src/Services/Mailer.php` osztályt, amely a PHPMailert becsomagolja, és egyetlen `send(string $to, string $subject, string $body): void` metódust kínál;
- A `public/index.php` fájlból küldjük el a naptár következő eseményét e-mailben.

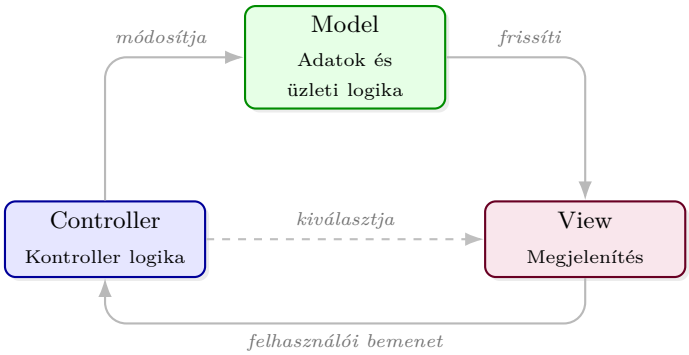
9. FEJEZET

MVC TÍPUSÚ ALKALMAZÁSOK

9.1. Az MVC-architektúra alapelvei

A Model–View–Controller (MVC) egy szoftvertervezési minta, amely az alkalmazás felépítését három fő komponensre bontja (Fowler 2002). Ez a szétválasztás javítja a kód karbantarthatóságát, tesztelhetőségét és újrafelhasználhatóságát.

9.1.1. Az MVC komponensei



9.1. ábra. *MVC-architektúra*

- **Model** (modell): az alkalmazás adatait és üzleti logikáját kezeli. Felelős az adatbázis-műveletekért (CRUD), valamint az üzleti szabályokért és az adatok validálásáért. A modell nem végez megjelenítési feladatokat.
- **View** (nézet): a megjelenítésért felelős réteg. Általában HTML-sablonokból áll. Egyszerű megjelenítési logika (pl. ciklusok, feltételek) megengedett, de üzleti logika nem kerülhet ide.
- **Controller** (kontroller): összeköti a modellt és a nézetet. Fogadja a felhasználói kéréseket, meghívja a megfelelő modell-metódusokat, majd

kiválasztja és betölti a megfelelő nézetet. A kontroller csak koordinációs logikát tartalmaz: az üzleti logika túlnyomó részét a modellbe kell helyezni.

Megjegyzés

Nagyobb alkalmazásokban a kontroller és a modell közé gyakran bekerül egy *Service réteg* is. A Service osztályok tartalmazzák az összetettebb üzleti logikát (pl. e-mail-küldés, komplex számítások), így a kontroller valóban csak koordinál, a modell pedig elsősorban az adatkezelésre koncentrál.

9.1.2. Miért van szükség az MVC-re?

Vegyünk egy egyszerű hallgató-nyilvántartó alkalmazást, ahol kezdetben minden kód egyetlen fájlban található:

```php
<?php
// adatbázis kapcsolat
$pdo = new PDO("mysql:host=localhost;dbname=students",
    "root", "");

// lekérdezés
$stmt = $pdo->query("SELECT * FROM students");

// HTML megjelenítés
echo "<html><body>";
echo "<table>";
while ($row = $stmt->fetch(PDO::FETCH_ASSOC)) {
    echo "<tr><td>{$row['name']}</td></tr>";
}
echo "</table>";
echo "</body></html>";
```

9.1. kódpélda. *Helytelen példa: minden logika egy fájlban*

Problémák ezzel a megközelítéssel:

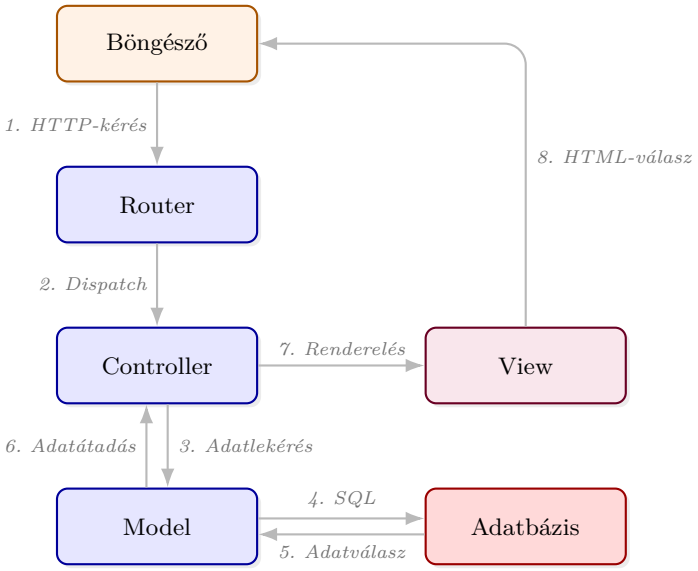
- **Karbantarthatóság**: ha módosítani akarjuk a HTML-struktúrát, az adatbázis-logika között kell keresgélnünk;
- **Újrafelhasználás**: az adatbázis-logikát nem tudjuk más nézetekben használni;
- **Tesztelés**: nem tudjuk tesztelni az adatbázis-logikát a HTML nélkül;

- **Csapatmunka**: a designer és a backend fejlesztő nem tud egyszerre dolgozni.

Az MVC-minta ezeket a problémákat a felelősségek szétválasztásával oldja meg.

9.1.3. HTTP-kérés életciklusa

A 9.2. ábra bemutatja, hogyan halad végig egy HTTP-kérés az MVC-alkalmazás rétegein.



9.2. ábra. *MVC-alkalmazás kérés–válasz folyamata*

1. **HTTP-kérés**: a böngésző HTTP-kérést küld a szerver felé.
2. **Dispatch** (kérésirányítás): a Router az URL alapján meghatározza, melyik kontroller metódust kell meghívni, és továbbítja a kérést.
3. **Adatlekérés**: a kontroller meghívja a modell megfelelő metódusát az adatok lekéréséhez.
4. **SQL**: a modell végrehajtja az adatbázis-műveletet (SELECT, INSERT, UPDATE, DELETE).
5. **Adatválasz**: az adatbázis visszaadja az eredményt a modellnek.

6. **Adatátadás**: a modell visszaadja az adatokat a kontrollernek.
7. **Renderelés**: a kontroller átadja az adatokat a nézetnek, amely rendereli a HTML-választ.
8. **HTML-válasz**: a generált tartalom visszakerül a böngészőhöz.

Megjegyzés

A 9.2. ábra GET-kérést szemléltet (adatok lekérése), de az MVC-architektúra POST-, PUT- és DELETE-kéréseket is ugyanilyen módon kezel. A kontroller dönti el, hogy melyik modell-metódust kell meghívni.

9.2. Egyszerű MVC-keretrendszer készítése

Ebben az alfejezetben lépésről lépésre felépítünk egy egyszerű MVC-alapú alkalmazást, amelyet fokozatosan bővítünk.

9.2.1. Projektstruktúra

Először hozzuk létre a projektünk alapvető könyvtárstruktúráját:

student-mvc/

```
|-- src/
|   |-- Controller/
|   |   +-- StudentController.php
|   |-- Model/
|   |   +-- Student.php
|   +-- Core/
|       |-- Database.php
|       +-- Router.php
|-- views/
|   +-- student/
|       |-- index.php
|       |-- create.php
|       +-- show.php
|-- public/
|   +-- index.php     (belépési pont)
|-- vendor/           (Composer - automatikusan generált)
|-- composer.json
```

A könyvtárak szerepe:

- `src/Controller/`: kontrollerek, amelyek összekapcsolják a modellt és a nézetet;

- `src/Model/`: modellek, amelyek az adatbázis-műveleteket végzik;
- `src/Core/`: a keretrendszer alaposztályai (pl. Router);
- `views/`: nézetek, amelyek a HTML-megjelenítést kezelik;
- `public/index.php`: Front Controller, az alkalmazás egyetlen belépési pontja;
- `vendor/`: Composer által telepített függőségek és autoloader.

9.2.2. Adatbázis-konfiguráció

Az adatbázis-kapcsolat létrehozásáért a `Database` osztály felelős, amely a PDO konfigurációját és példányosítását végzi. A belépési pontban egyszer kerül példányosításra, a PDO-kapcsolat pedig függőségként adódik át a modelleknek.

```php
<?php
namespace App\Core;

use PDO;

define('DB_HOST', 'localhost');
define('DB_USER', 'root');
define('DB_PASS', '');
define('DB_NAME', 'student_mvc');

class Database
{
    private PDO $connection;

    public function __construct()
    {
        $dsn = sprintf('mysql:host=%s;dbname=%s;charset=
            utf8mb4', DB_HOST, DB_NAME);
        $this->connection = new PDO($dsn, DB_USER, DB_PASS, [
            PDO::ATTR_ERRMODE            => PDO::
                ERRMODE_EXCEPTION,
            PDO::ATTR_DEFAULT_FETCH_MODE => PDO::FETCH_ASSOC,
            PDO::ATTR_EMULATE_PREPARES   => false,
        ]);
    }

    public function getConnection(): PDO
    {
        return $this->connection;
    }
}
```

9.2. kódpélda. *Adatbázis-kapcsolat (Database.php)*

9.2.3. Modell osztály

A modell az adatkezelésért felelős réteg. A Student modell alapvető CRUD-műveleteket biztosít:

```php
namespace App\Model;

class Student
{
    private string $table = 'students';

    public function __construct(private \PDO $db) {}

    public function getAll(): array
    {
        return $this->db->query(
            "SELECT * FROM {$this->table} ORDER BY name ASC"
        )->fetchAll();
    }

    public function getById(int $id): ?array
    {
        $stmt = $this->db->prepare(
            "SELECT * FROM {$this->table} WHERE id = :id"
        );
        $stmt->execute(['id' => $id]);
        return $stmt->fetch() ?: null;
    }

    public function create(string $name, string $email): bool
    {
        if (!filter_var($email, FILTER_VALIDATE_EMAIL)) {
            return false;
        }

        $stmt = $this->db->prepare(
            "INSERT INTO {$this->table} (name, email) VALUES
                (:name, :email)"
        );
        return $stmt->execute(['name' => $name, 'email' =>
            $email]);
    }

    // update() és delete() metódusok hasonlóan...
}
```

9.3. kódpélda. *Student modell osztály*

9.2.4. Kontroller osztály

A kontroller összekapcsolja a modellt és a nézetet:

```php
namespace App\Controller;

use App\Model\Student;

class StudentController
{
    public function __construct(private Student
        $studentModel) {}

    public function index(): void
    {
        $students = $this->studentModel->getAll();
        include __DIR__ . '/../../views/student/index.php';
    }

    public function show(): void
    {
        $id = (int)($_GET['id'] ?? 0);
        $student = $this->studentModel->getById($id);

        if (!$student) {
            header('Location: index.php?action=index');
            exit;
        }

        include __DIR__ . '/../../views/student/show.php';
    }

    public function create(): void
    {
        if ($_SERVER['REQUEST_METHOD'] === 'POST') {
            $name = $_POST['name'] ?? '';
            $email = $_POST['email'] ?? '';

            if ($this->studentModel->create($name, $email)) {
                header('Location:
                    index.php?action=index&msg=created');
                exit;
            }
        }

        include __DIR__ . '/../../views/student/create.php';
    }

    // edit() és delete() metódusok hasonlóan...
}
```

9.4. kódpélda. *StudentController osztály*

A kontroller feladatai:

- Fogadja a HTTP-kéréseket;
- Meghívja a modell megfelelő metódusait;
- Betölti a megfelelő nézetet és átadja neki az adatokat;
- Átirányítást végez szükség esetén.

Megjegyzés

Fat Model, Skinny Controller: Az üzleti logika (számítások, validáció, adatbázis-műveletek) a modellbe kerüljön. A kontroller maradjon „vékony”: csak a kérések irányítására koncentráljon.

9.2.5. Front Controller

Az `index.php` az alkalmazás egyetlen belépési pontja:

```php
<?php
require __DIR__ . '/../vendor/autoload.php';

use App\Core\Database;
use App\Model\Student;
use App\Controller\StudentController;

$db = (new Database())->getConnection();
$action = $_GET['action'] ?? 'index';

$studentModel = new Student($db);
$controller = new StudentController($studentModel);

match ($action) {
    'index'  => $controller->index(),
    'show'   => $controller->show(),
    'create' => $controller->create(),
    'edit'   => $controller->edit(),
    'delete' => $controller->delete(),
    default  => header('Location: index.php?action=index'),
};
exit;
```

9.5. kódpélda. *Front Controller (index.php)*

Az `index.php` feladatai:

- Az URL-paraméterek alapján dönti el, hogy melyik műveletet kell végrehajtani;
- Példányosítja a modellt és a kontrollert;
- A `match` kifejezéssel kezeli az elágazásokat.

9.2.6. View példa

A nézet felelős a HTML-megjelenítésért:

```
<!DOCTYPE html>
<html lang="hu">
<head>
    <meta charset="UTF-8">
    <title>Hallgatók listája</title>
</head>
<body>
    <h1>Hallgatók</h1>
    <a href="index.php?action=create">Új hallgató</a>
    <table>
        <thead>
            <tr><th>Név</th><th>Email</th><th>Műveletek</th
                ></tr>
        </thead>
        <tbody>
            <?php foreach ($students as $row): ?>
            <tr>
                <td><?= htmlspecialchars($row['name']) ?></td
                    >
                <td><?= htmlspecialchars($row['email']) ?></
                    td>
                <td>
                    <a href="index.php?action=show&id=<?=
                        $row['id'] ?>">
                        Megtekintés
                    </a>
                </td>
            </tr>
            <?php endforeach; ?>
        </tbody>
    </table>
</body>
</html>
```

9.6. kódpélda. *Hallgatók listája (index.php)*

A `htmlspecialchars()` hívások XSS-védelmet biztosítanak: az adatbázisból érkező értékeket (például a hallgató nevét) a böngésző nem értelmezi HTML-elemként, ha speciális karaktereik (pl. `<`, `>`, `"`) kódolva vannak. Ez különösen fontos felhasználói bevitelből származó adatoknál.

9.2.7. Az alapvető struktúra problémái

Az első verzió működik, de több problémát is felvet:

- **Routinglogika**: a `match` kifejezés minden új actionnel bővül;
- **Kódismétlődés**: a kontrollerekben ismétlődik az `include` és a `header`;
- **Több entitás**: több entitásnál a routing még bonyolultabb lesz;
- **Modell-duplikáció**: ugyanazok a CRUD-műveletek minden modellben.

Ezeket a problémákat a következő fejlesztési lépésekben oldjuk meg.

9.3. Routing

A routing (útvonalválasztás) az MVC-alkalmazások egyik legfontosabb része. Feladata, hogy a beérkező HTTP-kéréseket a megfelelő kontroller metódusokra továbbítsa.

9.3.1. Miért van szükség Router osztályra?

Az előző példában látott egyszerű `match()` alapú routing nehezen karbantartható több entitás esetén:

```php
match ($action) {
    'list' => $controller->list(),
    'show' => $controller->show(),
    'create' => $controller->create(),
    'edit' => $controller->edit(),
    'delete' => $controller->delete(),
    // Most jönnek a további entitások...
    'other_list' => $otherController->list(),
    'other_show' => $otherController->show(),
    // ... és így tovább, egyre bonyolultabb
    default => header('Location: index.php'),
};
exit;
```

9.7. kódpélda. *Problémás routing több erőforrással*

Az egyszerű megközelítés korlátai:

- Minden új action külön sorban jelenik meg az `index.php`-ben;
- Nehezen áttekinthető, hogy mely action melyik kontrollerhez tartozik;
- A kontroller- és modell-példányosítás keveredik a routing-logikával.

9.3.2. A Router osztály implementációja

A Router feladata az URL-ben érkező action értékek és a hozzájuk tartozó kontroller metódusok összerendelése, valamint a kérés továbbítása a megfelelő kontrollernek (dispatch).

Az `add()` metódussal regisztráljuk a route-okat, míg a `dispatch()` metódus végrehajtja a teljes kérésfeldolgozási folyamatot.

Az implementáció egyszerűsége miatt a kontroller és modell példányosítását is a Router végzi, így egy egyszerű factory jellegű megoldásként értelmezhető.

```php
namespace App\Core;

use PDO;

class Router
{
    private array $routes = [];
    private PDO $database;

    public function __construct(PDO $database)
    {
        $this->database = $database;
    }

    public function add(string $action,
                        string $controllerClass,
                        string $method,
                        string $modelClass): void
    {
        $this->routes[$action] = [
            'controller' => $controllerClass,
            'method' => $method,
            'model' => $modelClass
        ];
    }

    public function dispatch(?string $action = null): void
    {
```

```
        $action = $action ?? $_GET['action'] ?? 'index';

        if (!isset($this->routes[$action])) {
            $this->redirect404();
            return;
        }

        $route = $this->routes[$action];

        $model = new $route['model']($this->database);

        $controller = new $route['controller']($model);

        $method = $route['method'];
        $controller->$method();
    }

    private function redirect404(): void
    {
        http_response_code(404);
        echo "A kért oldal nem található.";
        exit;
    }
}
```

9.8. kódpélda. *Router osztály*

9.3.3. Router használata

A Router használatával az `index.php` lényegesen rövidebb lesz: minden erőforráshoz (Student, Course stb.) elegendő néhány `add()` hívás a korábbi, terjedelmes `match()` szerkezet helyett.

```
<?php
require __DIR__ . '/../vendor/autoload.php';

use App\Core\Database;
use App\Core\Router;
use App\Model\Student;
use App\Controller\StudentController;

$db = (new Database())->getConnection();
$router = new Router($db);

// Student routes
$router->add('index', StudentController::class,
```

```
            'index', Student::class);
$router->add('show', StudentController::class,
            'show', Student::class);
$router->add('create', StudentController::class,
            'create', Student::class);
// ... további routes

// Route végrehajtás
$router->dispatch();
```

9.9. kódpélda. *Front Controller Routerrel (index.php)*

9.3.4. A Router működése

Amikor egy HTTP-kérés érkezik (pl. `?action=create`), a Router a következő lépéseket hajtja végre:

1. **Route keresése**: megkeresi a `routes` tömbben a megfelelő actionhöz tartozó konfigurációt;
2. **Modell példányosítása**: létrehozza a szükséges modell objektumot (pl. `Student`);
3. **Kontroller példányosítása**: létrehozza a kontroller objektumot, és átadja neki a modellt;
4. **Metódus meghívása**: meghívja a kontroller megfelelő metódusát (pl. `create()`).

Így az `index.php`-ban csak az új route-okat kell regisztrálni, a példányosítási logikát a Router végzi.

A Router előnyei:

- **Tisztább struktúra**: a routing konfiguráció elkülönül a végrehajtási logikától;
- **Bővíthetőség**: új route-ok könnyen hozzáadhatók;
- **Központosított felelősség**: minden routing egy helyen kezelhető;
- **Automatikus példányosítás**: a Router kezeli az alapfüggőségek injektálását.

Megjegyzés

Ez a Router maga példányosítja a modelleket és a kontrollereket (factory jellegű megoldás), ezért szorosan csatolt. Ez egy átmeneti megoldás, amelyet a Dependency Injection Container (lásd a 9.6. alfejezetet) vált fel. Ebben a megközelítésben az objektumok a függőségeiket kívülről kapják meg.

9.4. Kontrollerek és modellek

Ebben az alfejezetben az öröklődést alkalmazzuk a kontrollerek és modellek közös funkcionalitásának kiszervezésére, ezáltal csökkentve a kódismétlődést és növelve az újrafelhasználhatóságot.

9.4.1. A probléma: kódismétlődés

A kontrollerekben a következő ismétlődések figyelhetők meg:

```
// StudentController.php
public function index(): void
{
    $students = $this->model->getAll();
    include 'views/student/index.php';  // Ismétlődés!
}

public function create(): void
{
    if ($_SERVER['REQUEST_METHOD'] === 'POST') {
        // ... validáció
        if ($this->model->create($name, $email)) {
            header('Location: index.php?action=index');  //
                Ismétlődés!
            exit;
        }
    }
    include 'views/student/create.php';  // Ismétlődés!
}
```

9.10. kódpélda. *Kódismétlődés a kontrollerekben*

A modell osztályokban is hasonló kódismétlődés figyelhető meg.

9.4.2. BaseController implementációja

Ezeket a közös műveleteket, vagyis a modell konstruktoron keresztüli átadását, a nézetbetöltést (render) és az átirányítást (redirect), egy absztrakt

`BaseController` osztályba szervezzük ki. Ez az osztály közvetlenül nem példányosítható, csak örökléssel használható.

```php
namespace App\Core;

abstract class BaseController
{
    public function __construct(protected BaseModel $model)
        {}

    protected function render(string $view,
                              array $data = [],
                              bool $layout = true): void
    {
        $viewFile = __DIR__ . "/../../views/{$view}.php";

        if (!file_exists($viewFile)) {
            throw new \RuntimeException("Nézet nem
                található: {$viewFile}");
        }

        // Output buffering - view tartalmát elkapjuk
        // Az extract() a $data tömb kulcsait változóként
            elérhetővé teszi a nézetben
        ob_start();
        extract($data);
        include $viewFile;
        $content = ob_get_clean();

        // Layout használata
        if ($layout) {
            include __DIR__ . '/../../views/layout.php';
        } else {
            echo $content;
        }
    }

    protected function redirect(string $action,
                                ?string $message = null,
                                array $params = []): void
    {
        $url = "index.php?action={$action}";

        if ($message !== null) {
            $url .= '&message=' . urlencode($message);
        }
```

```
        foreach ($params as $key => $value) {
            $url .= '&' . urlencode((string)$key) . '=' .
                urlencode((string)$value);
        }

        header("Location: {$url}");
        exit;
    }
}
```

9.11. kódpélda. *BaseController osztály*

A `BaseController` két fontos metódust biztosít:

- `render()`: egy nézet betöltése adatokkal, opcionálisan layout használatával;
- `redirect()`: átirányítás, opcionális üzenettel és paraméterekkel.

9.4.3. BaseModel implementációja

A modell osztályokban szintén számottevő kódismétlődés figyelhető meg. Minden entitáshoz (Student, Course stb.) ugyanazokat az alapvető CRUD-műveleteket (Create, Read, Update, Delete) implementáljuk, például a `getAll()`, `getById()` és `delete()` metódusokat.

Egy absztrakt `BaseModel` osztály segítségével ezek a közös műveletek egyszer kerülnek implementálásra, és minden konkrét modell automatikusan örökli őket. Az egyetlen változó elem a tábla neve, amelyet a konstruktorban paraméterként adunk át. Például a `Student` modell konstruktorában elegendő a `parent::__construct($db, 'students')` hívás, így az osztály automatikusan eléri a `students` táblára vonatkozó alapvető adatbázis-műveleteket.

Ez a megoldás egy egyszerűsített adathozzáférési rétegként értelmezhető, amely az adatbázis-műveletek egy részét központosítja.

```
namespace App\Core;

use PDO;

abstract class BaseModel
{
    protected PDO $db;
    protected string $table;

    public function __construct(PDO $db, string $table)
    {
        $this->db = $db;
```

```
        $this->table = $table;
    }

    public function getAll(): array
    {
        return $this->db->query(
            "SELECT * FROM {$this->table} ORDER BY id ASC"
        )->fetchAll();
    }

    public function getById(int $id): ?array
    {
        $sql = "SELECT * FROM {$this->table} WHERE id = :id";
        $stmt = $this->db->prepare($sql);
        $stmt->execute(['id' => $id]);
        return $stmt->fetch() ?: null;
    }

    public function delete(int $id): bool
    {
        $sql = "DELETE FROM {$this->table} WHERE id = :id";
        $stmt = $this->db->prepare($sql);
        return $stmt->execute(['id' => $id]);
    }
}
```

9.12. kódpélda. *BaseModel osztály*

9.4.4. Az öröklődés alkalmazása a gyakorlatban

Most a konkrét kontrollerek és modellek sokkal egyszerűbbek:

```
namespace App\Controller;

use App\Core\BaseController;

class StudentController extends BaseController
{
    public function index(): void
    {
        $students = $this->model->getAll();
        $this->render('student/index', [
            'students' => $students,
            'message'  => $_GET['message'] ?? null,
        ]);
    }
```

```
    public function create(): void
    {
        if ($_SERVER['REQUEST_METHOD'] === 'POST') {
            $name = $_POST['name'] ?? '';
            $email = $_POST['email'] ?? '';

            if ($this->model->create($name, $email)) {
                $this->redirect('index', 'created');
                return;
            }
        }

        $this->render('student/create');
    }
}
```

9.13. kódpélda. *StudentController osztály*

```
namespace App\Model;
use App\Core\BaseModel;
class Student extends BaseModel
{
    public function __construct(\PDO $db)
    {
        parent::__construct($db, 'students');
    }
    // Csak a specifikus metódusokat kell implementálni
    public function create(string $name, string $email): bool
    {
        if (!filter_var($email, FILTER_VALIDATE_EMAIL)) {
            return false;
        }
        $stmt = $this->db->prepare(
            "INSERT INTO {$this->table} (name, email) VALUES
                (:name, :email)"
        );
        return $stmt->execute(['name' => $name, 'email' =>
            $email]);
    }
    // A getAll(), getById(), delete()
    // már a BaseModel-ben van!
}
```

9.14. kódpélda. *Student modell osztály*

9.4.5. Osztályhierarchia

Az öröklődés eredményeként a következő osztályhierarchia alakul ki:
Kontrollerhierarchia:

- `BaseController` (abstract): közös `render()` és `redirect()` metódusok
  - `StudentController extends BaseController`.
  - további kontroller osztályok.

Modellhierarchia:

- `BaseModel` (abstract): közös CRUD-műveletek
  - `Student extends BaseModel`.
  - további modell osztályok.

Ez a megoldás a DRY (Don't Repeat Yourself) elv gyakorlati alkalmazása: a közös funkcionalitást egyszer definiáljuk, majd azt minden leszármazott osztály automatikusan örökli és újrafelhasználja.

9.5. Nézetek és sablonrendszerek

A nézetréteg a felhasználói felület megjelenítéséért felelős. A modern webes alkalmazásokban sablonrendszerek (template systems) segítségével strukturáljuk a HTML-szerkezetet.

A sablonrendszer fő elemei:

- **Layout**: a közös oldalváz (fejléc, lábléc, navigáció);
- **Részsablonok (partials)**: újrafelhasználható sablonrészletek (például űrlapmezők);
- **Nézetek**: az egyes oldalak egyedi tartalma.

Ez a megközelítés lehetővé teszi a HTML-szerkezet újrafelhasználását, valamint a megjelenés egyszerűbb karbantartását és módosítását. A bonyolultabb alkalmazásokban gyakran használnak teljes értékű sablonmotorokat (például Twig vagy Blade), azonban kisebb alkalmazások esetén a PHP beépített lehetőségei is elegendőek.

9.5.1. Layout rendszer

Eddig minden nézet teljes HTML-oldalt tartalmazott, ami jelentős kódismétlődést eredményezett. A layout-megközelítés ezt a problémát egy központi sablon (layout) bevezetésével oldja meg, amelybe a konkrét oldal tartalma kerül beillesztésre:

```
<!DOCTYPE html>
<html lang="hu">
<head>
    <meta charset="UTF-8">
    <title><?= htmlspecialchars($title ?? 'Student MVC')
        ?></title>
    <link rel="stylesheet" href="assets/style.css">
</head>
<body>
    <?php include __DIR__ . '/partials/nav.php'; ?>

    <main class="container">
        <?php if (isset($message)): ?>
            <div class="alert">
                <?= htmlspecialchars($message) ?>
            </div>
        <?php endif; ?>

        <?= $content ?? '' ?>
    </main>

    <?php include __DIR__ . '/partials/footer.php'; ?>
</body>
</html>
```

9.15. kódpélda. *Fő layout (layout.php)*

9.5.2. ViewHelper osztály

A nézetrétegben gyakran előforduló műveleteket segédfüggvényekbe (helper metódusokba) szervezhetjük, ezzel csökkentve a kódismétlődést és növelve a kód újrafelhasználhatóságát.

```
namespace App\Helpers;

class ViewHelper
{
    /**
     * HTML-kódolás XSS elleni védelemhez
     */
```

```
    public static function escape(mixed $var): string
    {
        if (is_null($var)) {
            return '';
        }
        return htmlspecialchars((string)$var,
                                ENT_QUOTES, 'UTF-8');
    }

    /**
     * URL generálás
     */
    public static function url(string $action,
                               array $params = []): string
    {
        $url = 'index.php?action=' . urlencode($action);

        foreach ($params as $key => $value) {
            $url .= '&' . urlencode((string)$key) .
                    '=' . urlencode((string)$value);
        }

        return $url;
    }

    // További helper metódusok...
}
```

9.16. kódpélda. *ViewHelper segédosztály*

9.5.3. Űrlaprészletek újrafelhasználása

Az újrafelhasználható űrlapelemeket külön sablonfájlokba szervezhetjük, amelyeket részsablonoknak (partials) nevezünk.

```
<?php use App\Helpers\ViewHelper; ?>

<div class="form-group">
    <label for="name">Név *</label>
    <input
        type="text"
        id="name"
        name="name"
        value="<?= ViewHelper::escape($student['name'] ??
            '') ?>"
        required
    >
```

```
</div>

<div class="form-group">
    <label for="email">Email cím *</label>
    <input
        type="email"
        id="email"
        name="email"
        value="<?= ViewHelper::escape($student['email'] ??
            '') ?>"
        required
    >
</div>

<!-- További mezők... -->
```

9.17. kódpélda. *Újrafelhasználható form részlet (_student_form.php)*

9.5.4. Nézetek használata

Most a konkrét nézetek sokkal egyszerűbbek:

```
<?php use App\Helpers\ViewHelper; ?>

<div class="form-container">
    <h1>Új hallgató létrehozása</h1>

    <form method="POST" action="">
        <?php include __DIR__ .
            '/../partials/_student_form.php'; ?>

        <div class="d-flex gap-2">
            <button type="submit" class="btn btn-primary">
                Létrehozás
            </button>
            <a href="<?= ViewHelper::url('index') ?>"
               class="btn btn-secondary">
                Mégse
            </a>
        </div>
    </form>
</div>
```

9.18. kódpélda. *Létrehozó nézet (create.php)*

9.5.5. A `BaseController::render()` metódus működése

A layout rendszer használatához a `BaseController::render()` metódus az output buffering technikát alkalmazza:

1. Először betölti a konkrét nézetet (például `student/create.php`);
2. A nézet kimenetét elkapja az output buffering segítségével;
3. A generált tartalmat átadja a layout sablonnak a `$content` változóban;
4. A layout beilleszti ezt a tartalmat a teljes HTML-szerkezetbe.

Az output buffering (`ob_start()`) lehetővé teszi, hogy a PHP ne azonnal küldje ki a HTML-kimenetet a böngészőnek, hanem azt először egy belső pufferben gyűjtse. Az `ob_get_clean()` a puffer tartalmát egy változóba menti (`$content`), amelyet a layout sablon a `<?= $content ?? " ?>` kifejezéssel illeszt be a megfelelő helyre.

9.6. Dependency Injection alapok

Az eddig megismert megoldásokban a Router példányosította a modelleket és a kontrollereket. Ez működőképes, de szorosan csatolt és nehezen bővíthető megoldás. A Dependency Injection (DI) egy fejlettebb megközelítés, amely lehetővé teszi az objektumok függőségeinek központi és rugalmas kezelését.

9.6.1. Mi a Dependency Injection?

A Dependency Injection (Fowler 2004) egy tervezési minta, amely szerint egy objektum nem maga hozza létre a függőségeit, hanem azokat kívülről kapja meg. Ez az elv a modern PHP-keretrendszerekben (például Laravel, Stauffer 2023) is alapvető építőelem, és a laza csatolás megvalósításának egyik fontos eszköze.

```php
class StudentController
{
    private Student $model;

    public function __construct()
    {
        // Rossz megoldás: szoros csatolás (hardcoding)
        $db = (new Database())->getConnection();
        $this->model = new Student($db);
    }
}
```

9.19. kódpélda. *Hardcoding (nem ajánlott megoldás)*

```php
class StudentController
{
    // PHP 8.0+: constructor property promotion
    public function __construct(private Student $model) {}
}
```

9.20. kódpélda. *Dependency Injection (ajánlott megközelítés)*

A Dependency Injection előnyei:

- **Tesztelhetőség**: a függőségek könnyen helyettesíthetők tesztelés során (például mock objektumokkal);
- **Rugalmasság**: az implementációk egyszerűen cserélhetők;
- **Laza csatolás**: az osztályok nem függenek konkrét implementációktól;
- **Jobb karbantarthatóság**: a függőségek kezelése központilag, külsőleg történik.

9.6.2. DI Container implementációja

A Dependency Injection Container (DI Container vagy IoC Container) egy központi komponens, amely az alkalmazás objektumainak létrehozását és függőségeinek összerendelését kezeli. Működési elve, hogy regisztráljuk benne, hogyan kell egy adott osztályt példányosítani, majd a Container ezt igény szerint elvégzi. A jelen implementáció egy egyszerű gyorsítótárazási mechanizmust is alkalmaz: ugyanazt a service-t többször kérve mindig ugyanazt a példányt adja vissza.

```php
namespace App\Core;

class Container
{
    private array $services = [];
    private array $instances = [];

    // Service regisztrálása closure-rel
    public function set(string $id, callable $resolver): void
    {
        $this->services[$id] = $resolver;
        unset($this->instances[$id]);
    }

    // Service lekérése (caching)
    public function get(string $id): mixed
```

```
    {
        // Ha már van instance, adjuk vissza azt
        if (isset($this->instances[$id])) {
            return $this->instances[$id];
        }

        if (!isset($this->services[$id])) {
            throw new \RuntimeException(
                "Service '{$id}' nem regisztrálva."
            );
        }

        // Service factory meghívása és caching
        $this->instances[$id] =
            ($this->services[$id])($this);

        return $this->instances[$id];
    }
}
```

9.21. kódpélda. *DI Container osztály*

9.6.3. Container használata

A Container használatával az `index.php` központi konfigurációs és alkalmazásindítási (bootstrap) ponttá válik:

```
<?php
require __DIR__ . '/../vendor/autoload.php';

use App\Core\Container;
use App\Core\Database;
use App\Core\Router;
use App\Model\Student;
use App\Controller\StudentController;

// Container inicializálása
$container = new Container();

// Database service
$container->set('database', function () {
    return (new Database())->getConnection();
});

// Model definíciók
$container->set(Student::class, function (Container $c) {
    return new Student($c->get('database'));
```

```php
});

// Controller definíciók
$container->set(StudentController::class, function
    (Container $c) {
    return new StudentController($c->get(Student::class));
});

// Router inicializálása Containerrel
$router = new Router($container);

// Routes regisztrálása
$router->add('index', StudentController::class, 'index');
// ... további routes

// Request dispatch
$action = $_GET['action'] ?? 'index';
$router->dispatch($action, 'index');
```

9.22. kódpélda. *Front Controller DI Containerrel (index.php)*

9.6.4. Router módosítása Container használathoz

A Router mostantól nem közvetlenül példányosítja a kontrollereket, hanem a Containerből kéri le azokat:

```php
namespace App\Core;

class Router
{
    private array $routes = [];
    private Container $container;

    public function __construct(Container $container)
    {
        $this->container = $container;
    }

    public function add(string $action,
                        string $controllerClass,
                        string $method): void
    {
        $this->routes[$action] = [
            'controller' => $controllerClass,
            'method'     => $method,
        ];
    }
```

```
    public function dispatch(?string $action = null,
                             string $defaultAction =
                                 'index'): void
    {
        $action = $action ?? $_GET['action'] ??
            $defaultAction;

        if (!isset($this->routes[$action])) {
            http_response_code(404);
            echo "A kért oldal nem található.";
            exit;
        }

        $route = $this->routes[$action];
        $controllerClass = $route['controller'];
        $controller =
            $this->container->get($controllerClass);
        $method = $route['method'];
        $controller->$method();
    }
}
```

9.23. kódpélda. *Router osztály*

9.7. Gyakorlófeladatok

A következő feladatok egyetlen filmkatalógus-alkalmazás köré épülnek, és a fejezet szerkezetét követve fokozatosan bővítik az alkalmazást.

1. Hozzunk létre egy `film-mvc` nevű Composer-projektet a fejezetben bemutatott könyvtárstruktúrával. Készítsük el a `films` táblát az alábbi mezőkkel: cím, rendező, megjelenési év, műfaj. Implementáljuk a `Film` modellt teljes CRUD-műveletekkel, és a `FilmController` osztályt a következő műveletekkel:

    - `index`: az összes film listázása;
    - `show`: egy film részleteinek megjelenítése;
    - `create`: új film létrehozása (GET-kérés esetén az űrlap megjelenítése, POST-kérés esetén mentés);
    - `edit`: meglévő film szerkesztése (GET-kérés esetén előtöltött űrlap, POST-kérés esetén frissítés);
    - `delete`: film törlése és visszairányítás a listára.

A `public/index.php` fájlban `match`-alapú routinggal kezeljük a műveleteket, és készítsük el a megfelelő nézetfájlokat.

2. Szervezzük át az alkalmazást a `Router` osztály bevezetésével:

   - Cseréljük le a `match`-alapú routingot, és regisztráljuk a route-okat a `Router::add()` metódussal;
   - Egészítsük ki a `Film` modellt egy `search()` metódussal, amely cím vagy rendező alapján szűr (`LIKE`-operátorral);
   - A `FilmController` `index` metódusa olvassa be a `?search=` paramétert, és adja át a szűrt eredményeket a nézetnek;
   - A nézet jelenítse meg a keresőmezőt és az aktív szűrőt.

3. Vezessünk be öröklésen alapuló kódmegosztást és layout rendszert:

   - Hozzuk létre a `BaseController` és `BaseModel` osztályokat;
   - Alakítsuk át a `Film` modellt és a `FilmController` osztályt úgy, hogy ezekből örököljenek;
   - A `BaseController::render()` metódus output buffering segítségével illessze be az oldalak tartalmát egy közös `views/layout.php` sablonba;
   - Készítsünk újrafelhasználható `views/partials/_film_form.php` részsablont, amelyet a létrehozó és a szerkesztő nézet egyaránt használ.

4. Alakítsuk át az alkalmazást DI Container használatára:

   - Implementáljuk a `Container` osztályt;
   - Regisztráljuk benne az adatbázis-kapcsolatot, a `Film` modellt és a `FilmController` osztályt;
   - Módosítsuk a `Router` osztályt úgy, hogy a kontrollereket a Container segítségével hozza létre;
   - Az `index.php` legyen az alkalmazás egyetlen belépési pontja;
   - Adjunk hozzá egy új modellt (`Genre`: műfajok kezelése), regisztráljuk a Containerben, és vegyük fel a megfelelő route-okat.

10. FEJEZET

TESZTELÉS ÉS VERZIÓKÖVETÉS

A modern webfejlesztésben elengedhetetlen a kód minőségének ellenőrzése és a változások nyomon követése. Ez a fejezet két fontos témát mutat be: az automatikus tesztelést és a verziókövetést, amelyek a modern szoftverfejlesztés alapvető minőségbiztosítási eszközei.

10.1. Egységtesztelés PHPUnittal

10.1.1. Az egységtesztelés szerepe

Az egységtesztelés (unit testing) a szoftver legkisebb, önállóan tesztelhető egységeinek, például függvényeknek és metódusoknak az ellenőrzését jelenti (Bergmann 2026). Célja annak biztosítása, hogy minden komponens a várt módon működjön.

Az egységtesztelés előnyei:

- **Korai hibafeltárás**: a problémákat a fejlesztés korai szakaszában azonosíthatjuk, amikor még könnyen javíthatók.
- **Refaktorálás biztonsága**: a tesztek biztosítják, hogy a kód átalakítása során a meglévő funkcionalitás ne sérüljön.
- **Dokumentáció**: a tesztek példákon keresztül mutatják be az egyes komponensek használatát.
- **Fejlesztési hatékonyság**: a hibák korai felismerése csökkenti a későbbi javítások költségét, és hosszú távon gyorsítja a fejlesztési folyamatot.

10.1.2. PHPUnit telepítése

A PHPUnit a PHP egyik legelterjedtebb egységtesztelő keretrendszere. Széles körben használják modern PHP-projektekben és keretrendszerekben, például Laravel- vagy Symfony-alapú alkalmazásokban. Jól integrálható CI/CD-rendszerekkel is. Telepítése Composer segítségével történik:

```
composer require --dev phpunit/phpunit ^10
```

A `--dev` kapcsoló jelzi, hogy a PHPUnit csak fejlesztési függőségként kerül telepítésre, ezért éles (production) környezetben nem szükséges.

A telepítést követően a PHPUnit az alábbi paranccsal ellenőrizhető:

```
vendor/bin/phpunit --version
```

10.1.3. Projektstruktúra

Egy tipikus PHPUnit-alapú PHP-projekt könyvtárstruktúrája a következő:

my-project/

```
|-- src/
|   |-- Calculator.php
|   |-- User.php
|   +-- Product.php
|-- tests/
|   |-- CalculatorTest.php
|   |-- UserTest.php
|   +-- ProductTest.php
|-- vendor/
|-- composer.json
+-- phpunit.xml
```

A `src` könyvtár tartalmazza az alkalmazás forráskódját, míg a `tests` könyvtár a hozzá tartozó tesztosztályokat. Általános konvenció, hogy minden forrásosztályhoz külön tesztfájl tartozik, amelynek neve az eredeti osztály nevéből és a `Test` utótagból áll.

10.1.4. Alapkonfiguráció

A projekt gyökérkönyvtárában hozzuk létre a `phpunit.xml` konfigurációs fájlt. Ez a fájl határozza meg a PHPUnit működésének alapbeállításait, például a tesztek elérési útját és a futtatási opciókat.

```
<?xml version="1.0" encoding="UTF-8"?>
<phpunit
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:noNamespaceSchemaLocation=
              "https://schema.phpunit.de/10.0/phpunit.xsd"
          bootstrap="vendor/autoload.php">
     <testsuites>
         <testsuite name="Application Test Suite">
             <directory>tests</directory>
```

```
        </testsuite>
    </testsuites>
</phpunit>
```

10.1. kódpélda. *PHPUnit konfiguráció (phpunit.xml)*

A `tests` könyvtár tartalmazza a tesztfájlokat. A konfiguráció megadja, hogy a PHPUnit ebből a könyvtárból gyűjtse be a teszteket. PHPUnit 10-ben egyes korábbi XML-beállítások megszűntek (például a `verbose`). Ezek helyett parancssori kapcsolók használhatók, például a `--display-warnings`.

10.1.5. Tesztosztályok és tesztmetódusok

Egy egyszerű osztály tesztelése jól szemlélteti a PHPUnit használatát. A `Calculator` osztály alapvető aritmetikai műveleteket valósít meg: összeadás, kivonás, szorzás és osztás. Az osztás esetén külön kezelni kell a nullával való osztást, amely kivételt eredményez.

```php
<?php
class Calculator
{
    public function add(int $a, int $b): int
    {
        return $a + $b;
    }
    public function subtract(int $a, int $b): int
    {
        return $a - $b;
    }
    public function multiply(int $a, int $b): int
    {
        return $a * $b;
    }
    public function divide(int $a, int $b): float
    {
        if ($b === 0) {
            throw new \InvalidArgumentException(
                'Nullával való osztás nem megengedett.'
            );
        }
        return $a / $b;
    }
}
```

10.2. kódpélda. *Calculator osztály*

A `Calculator` osztály teszteléséhez hozzunk létre egy tesztosztályt. A konvenció szerint a tesztosztály neve az eredeti osztály nevéből és a `Test` utótagból áll. A `setUp()` metódus minden teszt előtt lefut, és előkészíti a tesztkörnyezetet, például létrehozza a tesztelendő objektumot. Az egyes tesztmetódusok egy adott funkcionalitás működését ellenőrzik, beleértve a kivételkezelési eseteket is.

A jól megírt tesztek az AAA (Arrange, Act, Assert) mintát követik. Először előkészítjük a tesztkörnyezetet (*Arrange*), majd meghívjuk a tesztelendő kódot (*Act*), végül ellenőrizzük az eredményt (*Assert*).

```php
<?php

use PHPUnit\Framework\TestCase;

class CalculatorTest extends TestCase
{
    private Calculator $calculator;

    protected function setUp(): void
    {
        $this->calculator = new Calculator();
    }

    public function testAdd(): void
    {
        $result = $this->calculator->add(5, 3);
        $this->assertEquals(8, $result);
    }

    public function testSubtract(): void
    {
        $result = $this->calculator->subtract(10, 4);
        $this->assertEquals(6, $result);
    }

    public function testMultiply(): void
    {
        $result = $this->calculator->multiply(4, 7);
        $this->assertEquals(28, $result);
    }

    public function testDivide(): void
    {
        $result = $this->calculator->divide(15, 3);
        $this->assertEquals(5.0, $result);
    }
```

```
    public function testDivideByZeroThrowsException(): void
    {
        $this->expectException(InvalidArgumentException::
            class);
        $this->calculator->divide(10, 0);
    }
}
```

10.3. kódpélda. *CalculatorTest osztály*

A tesztek futtatása:

```
vendor/bin/phpunit tests/CalculatorTest.php
```

10.1.6. Assertion metódusok

Az *assertion* (állítás) metódusok azt ellenőrzik, hogy a tényleges eredmény megfelel-e a várt értéknek. A PHPUnit számos ilyen metódust biztosít a leggyakoribb ellenőrzési feladatokhoz.

Egyenlőség és azonosság

Az `assertEquals()` metódus az értékek tartalmi egyezését vizsgálja, míg az `assertSame()` a típus- és értékazonosságot is ellenőrzi. Például az `assertEquals(42, '42')` igaz lesz, mivel a PHP automatikusan típuskényszerítést alkalmaz. Ezzel szemben az `assertSame(42, '42')` hamis, mert a két érték típusa eltér.

```
public function testEquality(): void
{
    // Egyenlőség ellenőrzése (tartalom)
    $this->assertEquals(42, 42);
    $this->assertEquals('hello', 'hello');

    // Azonosság ellenőrzése (típus és érték)
    $this->assertSame(42, 42);
    $this->assertNotSame(42, '42');  // különböző típusok
}
```

10.4. kódpélda. *Egyenlőség és azonosság ellenőrzése*

Logikai értékek

Az `assertTrue()` és `assertFalse()` metódusok logikai (boolean) értékek ellenőrzésére használhatók.

```php
public function testBooleans(): void
{
    $userIsAdmin = true;
    $this->assertTrue($userIsAdmin);
    $accountIsLocked = false;
    $this->assertFalse($accountIsLocked);
}
```

10.5. kódpélda. *Logikai értékek ellenőrzése*

Null értékek

Az `assertNull()` és `assertNotNull()` metódusokkal `null` értékeket ellenőrizhetünk.

```php
public function testNull(): void
{
    $result = findUser(999);
    $this->assertNull($result);  // nem talált felhasználót
    $result = findUser(1);
    $this->assertNotNull($result);  // talált felhasználót
}
```

10.6. kódpélda. *Null értékek ellenőrzése*

Tömbök ellenőrzése

Tömbök esetén gyakran szükség van annak ellenőrzésére, hogy tartalmaznak-e egy adott elemet, hány elemük van, illetve hogy üresek-e. Ezek az assertion metódusok gyors és egyszerű ellenőrzést biztosítanak a tömbök állapotának vizsgálatához.

```php
public function testArrays(): void
{
    $roles = ['admin', 'editor', 'viewer'];
    // Tömb tartalmazását ellenőrzi
    $this->assertContains('editor', $roles);
    // Tömb elemszámát ellenőrzi
    $this->assertCount(3, $roles);
    // Üres tömb ellenőrzése
    $emptyList = [];
    $this->assertEmpty($emptyList);
    $this->assertNotEmpty($roles);
}
```

10.7. kódpélda. *Tömbök ellenőrzése*

Sztring-műveletek

Sztringértékek esetén ellenőrizhető, hogy tartalmaznak-e egy adott részszöveget, illetve hogy megfelelnek-e egy adott mintának. Erre a következő metódusok használhatók:

- részszöveg keresése: `assertStringContainsString()`;
- mintaillesztés: `assertMatchesRegularExpression()`.

```php
public function testStrings(): void
{
    $message = 'Error: Invalid email address';
    // String tartalmaz részstringet
    $this->assertStringContainsString('email', $message);
    // String mintaillesztés
    $this->assertMatchesRegularExpression('/^Error:/',
        $message);
}
```

10.8. kódpélda. *Sztringek ellenőrzése*

Típusellenőrzések

A PHP-ben használt típusok ellenőrzésére számos assertion metódus áll rendelkezésre.

```php
public function testTypes(): void
{
    $age = 25;
    $this->assertIsInt($age);
    $name = 'John';
    $this->assertIsString($name);
    $scores = [85, 90, 78];
    $this->assertIsArray($scores);
    $isActive = true;
    $this->assertIsBool($isActive);
    $price = 99.99;
    $this->assertIsFloat($price);
}
```

10.9. kódpélda. *Típusellenőrzések*

10.1.7. Tesztelhető kód írásának alapelvei

Az egységtesztek megírásának előfeltétele a tesztelhető kód. Az előző fejezetben megismert Dependency Injection tervezési minta ebben is kulcsszerepet játszik: mivel a függőségeket kívülről adjuk át az osztálynak, tesztelés során valódi erőforrások, például adatbázis vagy külső API helyett „hamis”, úgynevezett *mock* objektumokat is használhatunk.

Tesztelés mock objektummal

A *mock* egy valódi osztály tesztelési célú helyettesítője, amely előre meghatározott válaszokat ad vissza. Ennek köszönhetően a tesztek futtatásához nincs szükség valós adatbázis-kapcsolatra vagy hálózati kommunikációra. A PHPUnit beépített `createMock()` metódusával mock objektumok hozhatók létre.

```php
<?php
use PHPUnit\Framework\TestCase;
class UserServiceTest extends TestCase
{
    public function testGetUserReturnsUserData(): void
    {
        // Mock PDO objektum létrehozása
        $pdoMock = $this->createMock(PDO::class);
        $stmtMock = $this->createMock(PDOStatement::class);
        // A mock viselkedését definiáljuk
        $stmtMock->method('execute')
                 ->willReturn(true);
        $stmtMock->method('fetch')
                 ->willReturn(['id' => 1, 'name' => 'Test
                     User']);
        $pdoMock->method('prepare')
                ->willReturn($stmtMock);
        // A UserService most a mock PDO-t kapja meg
        $userService = new UserService($pdoMock);
        // A teszt gyors, nincs valódi adatbázis művelet
        $result = $userService->getUser(1);
        $this->assertNotNull($result);
        $this->assertEquals('Test User', $result['name']);
    }
}
```

10.10. kódpélda. *Tesztelés mock PDO-val*

A teszt gyorsan végrehajtható, mivel nem igényel valós adatbázis-kapcsolatot vagy külső erőforrást.

További alapelvek

- **Kis, fókuszált metódusok**: egy metódus egyetlen, jól definiált feladatot végezzen. Ezáltal a kód könnyebben tesztelhető és átlátható.
- **Globális állapot kerülése**: kerülni kell a globális változókat és a statikus, belső állapotot tároló metódusokat, mivel ezek megnehezítik a tesztelést és a kiszámíthatóságot.
- **Tiszta függvények előnyben részesítése**: ahol lehetséges, olyan metódusokat érdemes írni, amelyek ugyanarra a bemenetre mindig ugyanazt az eredményt adják, mellékhatások nélkül.

10.1.8. API-tesztelés

A REST API-végpontok tesztelése biztosítja, hogy az alkalmazás helyesen kezeli a HTTP-kéréseket, és a megfelelő válaszokat adja vissza. Az API-tesztelés fő szempontjai:

- **HTTP-státuszkódok**: ellenőrizni kell, hogy a szerver a megfelelő státuszkódot adja vissza, például 200, 201, 400, 404 vagy 500;
- **JSON-válaszstruktúra**: biztosítani kell, hogy a válasz tartalmazza a szükséges mezőket és azok helyes formátumúak legyenek;
- **Adatellenőrzés**: tesztelni kell a hibás vagy hiányos bemeneti adatokra adott válaszokat;
- **HTTP-metódusok kezelése**: külön kell tesztelni a GET-, POST-, PUT- és DELETE-kérések helyes működését.

Az API-végpontok teszteléséhez HTTP-kéréseket küldünk a szervernek. Az alábbi példában egy segédmetódus végzi ezt a feladatot, amely cURL segítségével hajtja végre a kérést, és a választ strukturált tömb formájában adja vissza.

Az alábbi példa egy REST API-végpont integrációs tesztjét mutatja be:

```
<?php

use PHPUnit\Framework\TestCase;

class UserApiTest extends TestCase
{
    private string $baseUrl = 'http://localhost:8080/api';

    public function testGetUsersReturnsSuccess(): void
```

```
    {
        $response = $this->makeHttpRequest('GET',
            '/students');

        // Státuszkód ellenőrzése
        $this->assertEquals(200, $response['status']);

        // JSON struktúra ellenőrzése
        $this->assertArrayHasKey('data', $response['body']);
        $this->assertIsArray($response['body']['data']);
    }

    public function testCreateUserWithValidData(): void
    {
        $newUser = [
            'name' => 'Test User',
            'email' => uniqid('test') . '@example.com'
        ];

        $response = $this->makeHttpRequest('POST',
            '/students', $newUser);

        // 201 Created státuszkód
        $this->assertEquals(201, $response['status']);

        // Új felhasználó adatai visszaérkeznek
        $this->assertArrayHasKey('id',
            $response['body']['data']);
    }

    public function testCreateUserWithInvalidEmail(): void
    {
        $invalidUser = [
            'name' => 'Test',
            'email' => 'invalid-email'
        ];

        $response = $this->makeHttpRequest('POST',
            '/students', $invalidUser);

        // 422 Unprocessable Entity, mezőnkénti
            hibaüzenetekkel
        $this->assertEquals(422, $response['status']);
        $this->assertArrayHasKey('errors',
            $response['body']);
    }
```

```
    // Segédmetódus HTTP-kérésekhez
    private function makeHttpRequest(
        string $method,
        string $endpoint,
        ?array $data = null
    ): array {
        // cURL használata a HTTP-kérés végrehajtásához
        // Visszaadja: ['status' => ..., 'body' => ...]
    }
}
```

10.11. kódpélda. *API-tesztelés*

Ez már nem egységteszt, hanem *integrációs teszt*, mivel a rendszer több komponensének együttműködését vizsgálja, valódi HTTP-kérések küldésével a szerver felé.

10.1.9. Code coverage

A code coverage (kódlefedettség) azt mutatja meg, hogy a tesztek a forráskód mekkora részét hajtják végre. A PHPUnit az Xdebug bővítménnyel[1] együttműködve képes kódlefedettségi riportok generálására, amelyek HTML-formátumban is elérhetők:

```
vendor/bin/phpunit --coverage-html coverage/
```

A lefedettségi riport a `coverage` könyvtárban jelenik meg. A 80% feletti lefedettség gyakran elfogadott irányadó érték, azonban önmagában nem garantálja a hibamentességet. Fontosabb szempont a tesztek minősége: az összetett üzleti logikát, az elágazásokat és a határeseteket célszerű előnyben részesíteni a triviális getter és setter metódusokkal szemben.

10.1.10. Test-Driven Development (TDD)

A TDD (tesztvezérelt fejlesztés) egy szoftverfejlesztési módszertan, amelyben a fejlesztési folyamat iteratív módon a tesztekből indul ki: először a tesztet írjuk meg, majd az azt kielégítő implementáció következik (Beck 2002). A folyamat három lépésből áll:

1. **Red**: olyan teszt írása, amely egy még nem létező funkcionalitást ellenőriz, ezért a teszt sikertelen lesz.
2. **Green**: a minimális implementáció elkészítése, amely a tesztet sikeressé teszi.

[1] https://xdebug.org/

3. **Refactor**: a kód átalakítása úgy, hogy a tesztek továbbra is sikeresek maradjanak.

A TDD segít a fejlesztés előtti API-tervezésben, valamint elősegíti a magasabb szintű kódlefedettség természetes kialakulását.

10.1.11. Tesztek futtatása és eredmények értelmezése

A tesztek futtatása egyszerű parancsokkal történik:

```
# Összes teszt futtatása
vendor/bin/phpunit

# Konkrét tesztfájl futtatása
vendor/bin/phpunit tests/CalculatorTest.php

# Konkrét tesztmetódus futtatása
vendor/bin/phpunit --filter testAdd tests/CalculatorTest.php
```

A PHPUnit kimenete színkódolt formában jeleníti meg a teszteredményeket. Sikeres futtatás esetén például az alábbi kimenet látható:

```
PHPUnit by Sebastian Bergmann and contributors.

.....                                                               5 / 5
    (100%)

Time: 00:00.023, Memory: 6.00 MB

OK (5 tests, 5 assertions)
```

A pontok (`.`) az egyes sikeresen lefutott teszteket jelölik. Sikertelen teszt esetén a PHPUnit részletes információt ad a hiba típusáról, helyéről és az érintett assertionről.

10.2. Git-alapok

10.2.1. A verziókövetés célja

A verziókövetés olyan rendszer, amely rögzíti a fájlok időbeli változásait, lehetővé téve a korábbi állapotok visszaállítását és a módosítások nyomon követését.

A verziókövetés előnyei:

- **Változások követése**: minden módosítás rögzítésre kerül, így visszakereshető, hogy ki mikor és mit változtatott;

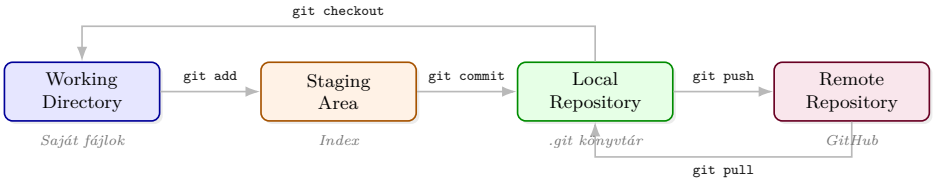
- **Együttműködés**: több fejlesztő párhuzamosan dolgozhat ugyanazon a projekten anélkül, hogy egymás munkáját felülírnák;
- **Biztonsági mentés**: a korábbi verziók bármikor visszaállíthatók hibás módosítás esetén;
- **Kísérletezés támogatása**: új funkciók fejlesztése elkülönítve történhet a stabil kódbázistól, branchek használatával;
- **Dokumentáció**: a commit üzenetek jól követhetően dokumentálják a fejlesztés menetét.

A Git (Chacon–Straub 2014) az egyik legszélesebb körben használt elosztott verziókövető rendszer, amelyet Linus Torvalds fejlesztett a Linux-kernel számára.

10.2.2. Alapfogalmak

A Git működésének megértéséhez érdemes tisztában lenni néhány alapfogalommal, amelyeket a 10.1. ábra is szemléltet.

- **Repository** (**tároló**): a projekt teljes verziótörténetét és metaadatait tartalmazó adattár, amelyet a Git a `.git` könyvtárban tárol. Két típusa van:
  - **Local repository** (**helyi tároló**): a fejlesztő saját gépén található példány;
  - **Remote repository** (**távoli tároló**): szerveren, például GitHubon tárolt központi adattár, amelyet több fejlesztő is elér.
- **Commit**: a projekt fájljainak egy adott állapotáról készült pillanatfelvétel, amely egyedi azonosítót (hash), szerzőt, dátumot és leíró üzenetet tartalmaz.
- **Branch (ág)**: a fejlesztés elkülönített vonala, amely lehetővé teszi új funkciók fejlesztését a stabil kódbázis módosítása nélkül. Az új tárolók fő ágának neve a GitHubon `main`. A `git init` beállítástól függően `master` vagy `main` nevű ágat hoz létre.
- **Working directory (munkakönyvtár)**: a fejlesztő helyi környezete, ahol a fájlok tényleges szerkesztése történik.
- **Staging area (előkészítési terület)**: átmeneti terület, ahol a következő commitba kerülő módosításokat előkészítjük.



10.1. ábra. *A Git fázisai*

10.2.3. Git telepítése és konfigurálása

A Git Windows-telepítőjét a hivatalos oldalról tölthetjük le: https://git-scm.com.

A letöltött telepítőt általában az alapértelmezett beállításokkal érdemes futtatni, mivel ezek a legtöbb fejlesztői környezetben megfelelőek.

Telepítés után nyissunk egy új Command Prompt vagy PowerShell ablakot, és ellenőrizzük a telepítés sikerességét:

```
git --version
```

A Git használatának megkezdése előtt be kell állítani a felhasználói adatokat:

```
git config --global user.name "Név"
git config --global user.email "email@example.com"
git config --global init.defaultBranch main
```

Ezek az adatok minden commithoz hozzárendelődnek, így később azonosítható, hogy ki készítette az adott módosítást. A harmadik parancs beállítja, hogy az új tárolók fő ága `main` nevű legyen. A konfiguráció ellenőrzése a következő paranccsal történhet:

```
git config --list
```

10.2.4. Repository létrehozása

Egy új projekt esetén a Git-repository inicializálása a következő lépésekben történik:

```
mkdir my-project
cd my-project
git init
```

Ez a parancs létrehoz egy rejtett `.git` könyvtárat, amely a verziókövetéshez szükséges összes metaadatot tartalmazza.

10.2.5. Alapvető Git-parancsok

git status

A *working directory* állapotát mutatja meg: jelzi, mely fájlok módosultak, melyek kerültek az előkészítési területre, és mely fájlok nincsenek még követve.

```
git status
```

Példa kimenet (új fájl hozzáadása előtt):

```
On branch main
No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        index.php

nothing added to commit but untracked files present
```

git add

A fájlok hozzáadása az előkészítési területhez, azaz előkészítés commithoz:

```
# Egy fájl hozzáadása
git add index.php

# Több fájl hozzáadása
git add index.php style.css

# Az aktuális könyvtár módosításainak hozzáadása
git add .

# A teljes repository összes változásának hozzáadása
git add -A
```

A `git add .` csak az aktuális könyvtárból indul ki, míg a `git add -A` a teljes repository területén figyelembe veszi az új fájlokat, módosításokat és törléseket is.

git commit

Az előkészítési terület tartalmának mentése a repositoryba:

```
# Commit üzenettel
git commit -m "Kezdeti commit: index.php létrehozása"

# Interaktív commit üzenet szerkesztéssel
git commit
```

Megjegyzés

Jó commit üzenetek írása:

- Legyen rövid és tömör, ideálisan legfeljebb 50 karakter;
- Angol nyelvű, parancsoló módban (például `Add feature`, nem `Added feature`);
- Részletesebb leírás esetén egy üres sor után következhet a magyarázat.

Sokan a *Conventional Commits* konvenciót használják, amely típuselőtagokat alkalmaz (például `feat:`, `fix:`, `docs:`, `refactor:`). Ez megkönnyíti a változásnaplók (changelog) automatikus generálását.

git log

A committörténet megtekintésére szolgáló parancs. Minden commit azonosítót, szerzőt, dátumot és üzenetet tartalmaz.

```
# Teljes történet
git log

# Egyszerűsített, egysoros formátum
git log --oneline

# Grafikus, ágstruktúrát is mutató megjelenítés
git log --graph --oneline --all
```

Példa kimenet (két legutóbbi commit):

```
commit a3f5d2b (HEAD -> main)
Author: Kovacs Bela <bela@example.com>
Date:   Mon Jan 6 10:30:00 2026 +0100

    Add login form validation

commit b8c9e1a
Author: Kovacs Bela <bela@example.com>
Date:   Mon Jan 6 09:15:00 2026 +0100

    Create database connection class
```

git diff

A módosítások összehasonlítására szolgáló parancs.

```
# Working directory és staging area közötti különbség
git diff

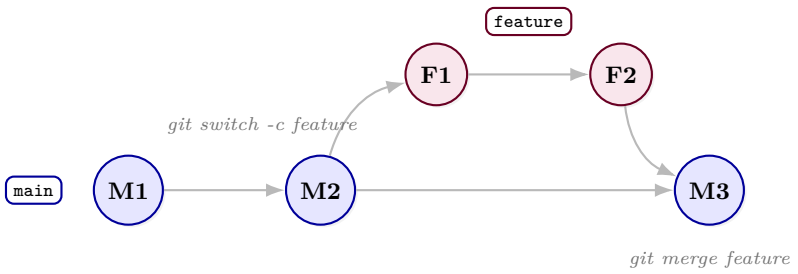
# Staging area és az utolsó commit közötti különbség
git diff --staged

# Két commit összehasonlítása
git diff commit1 commit2
```

10.2.6. Branchek használata

A branchek lehetővé teszik a párhuzamos fejlesztést anélkül, hogy a stabil kódbázist érintenénk.

A 10.2. ábra egy új funkció fejlesztésének tipikus menetét mutatja be branchek használatával.



10.2. ábra. *Branchalapú fejlesztési folyamat*

A branchekkel való munka alapvető parancsai a következők:

```
# Új branch létrehozása
git branch feature-login

# Branch-ek listázása
git branch

# Váltás egy meglévő branchre (ajánlott parancs)
git switch feature-login

# Régebbi parancs ugyanarra a célra
git checkout feature-login

# Új branch létrehozása és azonnali váltás
git switch -c feature-registration

# Branch törlése
git branch -d feature-login
```

Tipikus munkafolyamat:

```
# 1. Új feature branch létrehozása
git switch -c feature-user-profile

# 2. Módosítások elvégzése és commit készítése
git add profile.php
git commit -m "Add user profile page"

# 3. Visszaváltás a main branchre
git switch main

# 4. Feature branch beolvasztása
git merge feature-user-profile

# 5. Feature branch törlése
git branch -d feature-user-profile
```

10.2.7. .gitignore fájl

A `.gitignore` fájl határozza meg, hogy a Git mely fájlokat vagy könyvtárakat ne kövessen nyomon. Bizonyos állományokat nem célszerű verziókövetés alá venni, például automatikusan generált fájlokat, függőségeket, jelszavakat tartalmazó konfigurációkat vagy IDE-specifikus beállításokat.

PHP-projektekben a `.gitignore` fájl tipikusan a következő bejegyzéseket tartalmazza:

```
# Composer függőségek
/vendor/

# Környezeti változók (jelszavak, API kulcsok)
.env

# IDE-specifikus fájlok
.idea/
.vscode/

# Operációs rendszer fájlok
.DS_Store
Thumbs.db

# Log fájlok
*.log

# Ideiglenes fájlok
tmp/
cache/

# PHPUnit gyorsítótár
.phpunit.result.cache
```

Megjegyzés

Bevált gyakorlatok:

- Készítsünk gyakori, kisebb commitokat logikus egységek mentén;
- Írjunk egyértelmű, leíró commit üzeneteket;
- Használjunk brancheket új funkciók fejlesztéséhez;
- A `main` branch mindig legyen működőképes állapotban;
- Commit előtt ellenőrizzük a módosításokat (`git diff`).

10.3. GitHub-munkafolyamat

10.3.1. A GitHub szerepe

A GitHub egy webalapú Git-szolgáltatás, amely távoli repositoryk tárolását és a csapatmunka támogatását teszi lehetővé. A Git funkcióit kiegészítve számos együttműködési és automatizálási eszközt biztosít.

- **Távoli repositoryk tárolása**: biztonságos, központi tárhely a forráskód számára.
- **Csapatmunka támogatása**: pull requestek, code review és feladat- és hibakövetés.
- **Dokumentáció kezelése**: README-fájlok, wiki és GitHub Pages.
- **Automatizálás**: CI/CD-folyamatok és automatikus tesztelés GitHub Actions segítségével.
- **Nyílt forráskód támogatása**: közösségi projektek és külső hozzájárulások kezelése.

10.3.2. Repository létrehozása GitHubon

1. Jelentkezzünk be a GitHub-felületre (`https://github.com`).
2. A jobb felső sarokban kattintsunk a plusz ikonra, majd válasszuk a *New repository* lehetőséget.
3. Adjunk nevet a repositorynak, és opcionálisan adjunk hozzá leírást.
4. Válasszuk ki, hogy a repository publikus vagy privát legyen.
5. Kattintsunk a *Create repository* gombra.

A GitHub ezután útmutatást ad a helyi repository összekapcsolásához.

10.3.3. Távoli repository kezelése

Távoli repository hozzáadása

```
git remote add origin https://github.com/username/my-project.git
```

Az `origin` a távoli repository alapértelmezett neve.

Távoli repositoryk listázása

```
git remote -v
```

Első feltöltés

```
git push -u origin main
```

Az `-u` kapcsoló beállítja az upstream kapcsolatot, így később elegendő a `git push` parancs használata.

10.3.4. Clone, push, pull műveletek

Clone – repository letöltése

Egy meglévő GitHub-repository teljes letöltése:

```
git clone https://github.com/username/my-project.git
cd my-project
```

Ez a parancs létrehozza a repository helyi másolatát a teljes verziótörténettel együtt.

Push – változások feltöltése

A helyi commitok feltöltése a távoli repositoryba:

```
git add .
git commit -m "Add contact form"
git push origin main
```

A `git push` parancs a helyi commitokat szinkronizálja a távoli repositoryval.

Pull – változások letöltése

Más fejlesztők által feltöltött módosítások letöltése és beolvasztása:

```
git pull origin main
```

A `git pull` parancs két lépést hajt végre: letölti a változásokat (`git fetch`), majd beolvasztja azokat az aktuális branchbe (`git merge`).

10.3.5. Pull request-alapú munkafolyamat

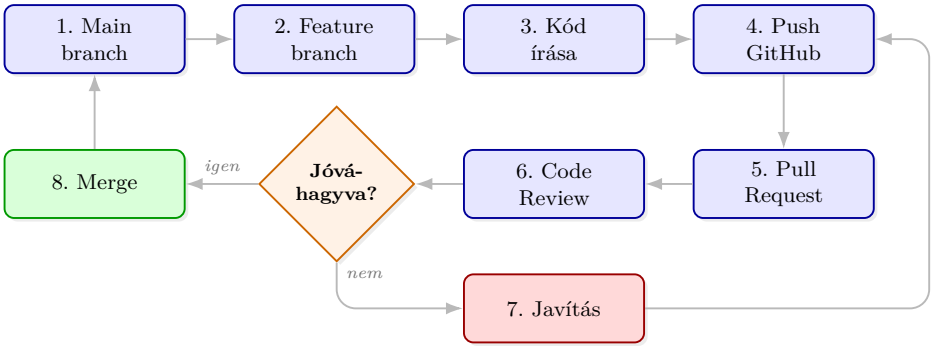
A pull request (PR) egy GitHub-funkció, amely lehetővé teszi a kódváltozások áttekintését, megbeszélését és ellenőrzését, mielőtt azok bekerülnek a fő branchbe (általában a `main` branch). A 10.3. ábra a folyamat tipikus lépéseit mutatja be.

1. **Branch létrehozása:**

   ```
   git switch -c feature-api-endpoint
   ```

2. **Fejlesztés és commitok készítése:**

   ```
   git add api.php
   git commit -m "Add user API endpoint"
   git push origin feature-api-endpoint
   ```



10.3. ábra. *Pull request-alapú fejlesztési munkafolyamat*

3. **Pull request létrehozása GitHubon:**
   - A GitHub automatikusan felajánlja a pull request létrehozását;
   - Meg kell adni a PR címét és leírását (mit és miért változott);
   - Reviewerek kijelölése.
4. **Code review:**
   - A csapattagok átnézik a kódot;
   - Kommenteket írnak és kérdéseket tesznek fel;
   - Szükség esetén módosításokat kérnek.
5. **Módosítások alapján új commitok:**

```
git add api.php
git commit -m "Fix validation logic based on review"
git push origin feature-api-endpoint
```

6. **Merge:**
   - Ha minden ellenőrzés sikeres, a pull request beolvasztható;
   - Gyakori merge stratégiák: merge commit, squash merge, rebase merge.
7. **Branch törlése:**

```
git branch -d feature-api-endpoint
git push origin --delete feature-api-endpoint
```

10.3.6. Code review

A code review célja a kód minőségének javítása, valamint a tudás megosztása a fejlesztőcsapaton belül. A folyamat során a csapattagok áttekintik egymás módosításait, mielőtt azok bekerülnek a `main` branchbe.

A review során érdemes az alábbi szempontokat figyelembe venni:

- **Funkcionalitás**: a kód teljesíti a megadott követelményeket?
- **Kódminőség**: olvasható, érthető és karbantartható a megoldás?
- **Tesztek**: rendelkezik megfelelő automatizált tesztekkel?
- **Biztonság**: nem tartalmaz nyilvánvaló sérülékenységeket?
- **Teljesítmény**: nem okoz indokolatlan teljesítményromlást?
- **Kódstílus**: követi a projektben elfogadott konvenciókat?

10.3.7. GitHub Actions és CI-alapok

A Continuous Integration (CI) egy fejlesztési gyakorlat, amely során a kódváltoztatások automatikus ellenőrzésen mennek keresztül minden egyes push vagy pull request esetén. Célja, hogy a hibák minél korábban, automatizált módon kerüljenek felismerésre.

A GitHub Actions segítségével CI-workflow-k hozhatók létre, amelyek például automatikusan futtatják a teszteket vagy egyéb ellenőrzéseket.

Érdemes a `composer.json` fájlban egy `test` szkriptet definiálni, hogy a tesztfuttatás egységes és egyszerűen használható legyen:

```
{
    "scripts": {
        "test": "vendor/bin/phpunit"
    }
}
```

10.12. kódpélda. *composer.json*

Ezután a tesztek a `composer test` paranccsal futtathatók.

A CI-folyamat részeként egy egyszerű GitHub Actions-workflow a `.github/workflows` könyvtárban definiálható:

```
name: PHPUnit Tests

on:
  push:
    branches: [ main ]
```

```yaml
  pull_request:
    branches: [ main ]

jobs:
  test:
    runs-on: ubuntu-latest

    steps:
      - uses: actions/checkout@v4

      - name: Setup PHP
        uses: shivammathur/setup-php@v2
        with:
          php-version: '8.2'

      - name: Install dependencies
        run: composer install

      - name: Run tests
        run: composer test
```

10.13. kódpélda. *GitHub Actions konfiguráció (.github/workflows/tests.yml)*

Ez a konfiguráció minden `main` branchre történő push és pull request esetén automatikusan lefuttatja a PHPUnit-teszteket. Sikertelen futás esetén a fejlesztők azonnali visszajelzést kapnak a GitHub-felületén.

A CI előnyei:

- Gyors visszajelzés a kód aktuális állapotáról;
- Csökkenti a manuális tesztelés szükségességét;
- Megakadályozza a hibás kód bekerülését a `main` branchbe;
- Objektív visszajelzést ad a tesztek futási eredményeiről.

10.4. Gyakorlófeladatok

1. Készítsünk egy `PasswordValidator` osztályt, amely ellenőrzi a jelszó erősségét, és logikai értéket ad vissza. Az érvényes jelszónak legalább 8 karakter hosszúnak kell lennie, tartalmaznia kell legalább egy nagybetűt, egy számot és egy speciális karaktert.

   Telepítsük a PHPUnitot fejlesztői függőségként, hozzuk létre a `phpunit.xml` konfigurációs fájlt, majd írjunk egységteszteket az AAA (Arrange, Act, Assert) minta szerint.

A tesztek fedjék le:

- a helyes bemeneteket (várható visszatérési érték: `true`);
- a hibás bemeneteket (várható visszatérési érték: `false`);
- a határeseteket (pl. pontosan 8 karakter, hiányzó nagybetű stb.).

2. Készítsünk egy `Mailer` osztályt `send(string $to, string $message): void` metódussal. Hozzunk létre egy `NotificationService` osztályt is, amely a `Mailer` példányát konstruktoron keresztül kapja meg. Írjunk egységtesztet PHPUnit-mock (test double) használatával:

   - hozzuk létre a `Mailer` mock objektumát a `createMock()` metódussal;
   - ellenőrizzük, hogy a `send()` metódus pontosan egyszer hívódik meg (`expects(once())`);
   - ellenőrizzük, hogy a megfelelő paraméterekkel hívódik meg.

3. Hozzunk létre egy `git-practice` nevű Git-repositoryt egy egyszerű PHP-projekttel:

   - készítsünk logikus fejlesztési lépésekre bontott commitokat;
   - hozzunk létre egy `feature-validation` branchet, és fejlesszük ott az inputvalidációs logikát;
   - hozzunk létre szándékosan konfliktust úgy, hogy ugyanazt a sort módosítjuk a `main` és a `feature-validation` branchen is;
   - merge-eljük a feature branchet a `main` branchbe, majd oldjuk fel a keletkező konfliktust manuálisan, szerkesztőben.

   Ellenőrizzük az eredményt a committörténet vizuális megjelenítésével (`git log --oneline --graph`).

4. Töltsük fel az előző projektet GitHubra, és kapcsoljuk össze a helyi repositoryt a GitHub-repositoryval:

   - hozzunk létre egy feature branchet, pusholjuk GitHubra, majd nyissunk pull requestet;
   - készítsünk egy egyszerű GitHub Actions-workflow-t (`.github/workflows/tests.yml`), amely automatikusan lefuttatja a PHPUnit-teszteket minden `main` branchre érkező push és pull request esetén;
   - ellenőrizzük a workflow futási eredményét a GitHub Actions-felületén, és értelmezzük a sikeres, illetve sikertelen futások visszajelzéseit.

SZAKIRODALOM

BEAULIEU, A.
2020 *Learning SQL*. Sebastopol, O'Reilly Media.

BECK, K.
2002 *Test-driven development: By example*. Boston, Addison-Wesley.

BERGMANN, S.
2026 *PHPUnit: The PHP testing framework*.
https://phpunit.de/documentation.html

CHACON, S.–STRAUB, B.
2014 *Pro Git*. New York, Apress.
https://git-scm.com/book

COMPOSER CONTRIBUTORS
2026 *Composer: Dependency manager for PHP*.
https://getcomposer.org/doc/

ELMASRI, R.–NAVATHE, S. B.
2016 *Fundamentals of database systems*. Hoboken, Pearson.

FIELDING, R. T.
2000 *Architectural styles and the design of network-based software architectures*. Doktori értekezés. Irvine, University of California.

FOWLER, M.
2002 *Patterns of enterprise application architecture*. Boston, Addison-Wesley.

FOWLER, M.
2003 *UML distilled: A brief guide to the standard object modeling language*. Boston, Addison-Wesley.

FOWLER, M.
2004 *Inversion of control containers and the dependency injection pattern*.
https://martinfowler.com/articles/injection.html

FREEMAN, E.–ROBSON, E.
2021 *Head first design patterns*. Sebastopol, O'Reilly Media.

MARTIN, R. C.
2002 *Agile software development: Principles, patterns, and practices*. Upper Saddle River, Prentice Hall.

MARTIN, R. C.
2008 *Clean code: A handbook of agile software craftsmanship*. Upper Saddle River, Prentice Hall.

OWASP FOUNDATION
2021 *OWASP top 10: Web application security risks.*
https://owasp.org/www-project-top-ten/

PHP-FIG
2013 *PSR-4: Autoloader.*
https://www.php-fig.org/psr/psr-4/

PHP GROUP
2026 *PHP manual.*
https://www.php.net/manual/

STAUFFER, M.
2023 *Laravel: Up & running.* Sebastopol, O'Reilly Media.

ZANDSTRA, M.
2021 *PHP 8 objects, patterns, and practice.* New York, Apress.

REZUMAT

Programare web în PHP

Volumul oferă o prezentare cuprinzătoare a dezvoltării web server-side în limbajul PHP. Lucrarea pornește de la bazele comunicării web și ale limbajului PHP și introduce treptat cititorul în principalele domenii ale dezvoltării aplicațiilor web moderne.

Primele capitole explică modul de funcționare a Web-ului, prezentând protocoalele HTTP și HTTPS, sintaxa limbajului PHP, sistemul de tipuri și structurile de control fundamentale. Capitolul dedicat programării orientate pe obiecte tratează noțiunile de clasă și obiect, moștenirea și compoziția, clasele abstracte și interfețele, polimorfismul, trait-urile și enumerările, precum și namespace-urile și gestionarea excepțiilor.

Capitolele următoare prezintă gestionarea formularelor web, validarea datelor, modelul PRG și încărcarea fișierelor, urmate de gestionarea sesiunilor și a cookie-urilor, cu accent pe aspectele de securitate, precum protecția CSRF și prevenirea deturnării sesiunilor. Capitolul dedicat bazelor de date acoperă accesul la date prin PDO și utilizarea instrucțiunilor pregătite. Lucrarea continuă cu proiectarea și implementarea API-urilor REST, gestionarea dependențelor cu Composer, arhitectura MVC, testarea automată și controlul versiunilor cu Git.

Capitolele conțin fundamente teoretice, exemple de cod și exerciții de verificare. Obiectivul manualului este de a oferi studenților cunoștințele necesare pentru dezvoltarea unor aplicații web moderne, sigure și ușor de întreținut.

ABSTRACT

Web Programming in PHP

This textbook provides a comprehensive guide to server-side web development in PHP. Starting from the fundamentals of web communication and the PHP language, it gradually introduces the reader to the main areas of modern web application development.

The opening chapters explain how the web works, covering the HTTP and HTTPS protocols, the PHP syntax, the type system, and fundamental control structures. The chapter on object-oriented programming covers classes and objects, inheritance and composition, abstract classes and interfaces, polymorphism, traits and enumerations, as well as namespaces and exception handling.

The following chapters cover web form processing, data validation, the PRG pattern, and file uploads, followed by session and cookie management with an emphasis on security aspects such as CSRF protection and session hijacking prevention. The chapter on database covers database access with PDO and the use of prepared statements. The textbook continues with the design and implementation of REST APIs, dependency management with Composer, MVC architecture, automated testing, and version control with Git.

The chapters include theoretical foundations, code examples, and review exercises. The goal of the textbook is to equip students with the knowledge necessary to develop modern, secure, and maintainable web applications.

A SZERZŐRŐL

Pál László 1975-ben született Csíkszeredában. Középiskolai tanulmányait a csíkszeredai Márton Áron Főgimnáziumban végezte, majd 1998-ban matematikából diplomázott a kolozsvári Babeș–Bolyai Tudományegyetemen. 2002 óta a Sapientia Erdélyi Magyar Tudományegyetem oktatója, 2009-től adjunktus, jelenleg egyetemi docens a Csíkszeredai Kar Gazdaságtudományi Tanszékén. 2011-ben szerzett informatikából doktori fokozatot a Szegedi Tudományegyetemen.

Kutatási területe az optimalizálás és annak gyakorlati alkalmazásai, különösen a globális optimalizálási módszerek és azok alkalmazása valós problémákra. Az utóbbi években kutatásai kiterjedtek ökonometriai modellekre és pénzügyi alkalmazásokra, ezen belül a portfólióoptimalizálásra és a mesterséges intelligencia pénzügyi döntéstámogatási alkalmazásaira is.

Az egyetemen Java-programozást, webprogramozást és mobilalkalmazásfejlesztést oktat.

ORCID: 0000-0002-7321-4558
E-mail-cím: pallaszlo@uni.sapientia.ro

Scientia Kiadó
400112 Kolozsvár (Cluj-Napoca)
Mátyás király (Matei Corvin) u. 4. sz.
Tel./fax: +40-364-401454
E-mail: scientia@kpi.sapientia.ro
www.scientiakiado.ro

Műszaki szerkesztés:
Pál László

Borítóterv:
Tipotéka Kft.

Korrektúra:
Szenkovics Enikő

Tipográfia:
Könczey Elemér

A webfejlesztés napjaink egyik legfontosabb informatikai területe, a modern digitális szolgáltatások jelentős része ugyanis szerveroldali technológiákra épül.

A kötet átfogóan mutatja be a PHP-alapú szerveroldali webfejlesztést, és az alapoktól vezeti az olvasót a modern webalkalmazások felépítéséig. A PHP szintaxisától és az objektumorientált szemlélettől az adatbázis-kezelésen, a REST API-k tervezésén, a csomagkezelésen és az MVC-architektúrán át egészen a tesztelésig és a verziókövetésig minden lényeges témakör helyet kap. Minden fejezet ötvözi az elméleti alapokat a szemléletes, futtatható kódpéldákkal, így a kötet egyaránt hasznos a témával most ismerkedő hallgatók és azok számára is, akik meglévő ismereteiket szeretnék rendszerezni és elmélyíteni.

ISBN 978-606-975-123-7