

# A rendszertervezés korszerű módszerei

---

Használati esetek (Use cases)

# Mi a használati eset? I.

- A felhasználóknak céljai, igényei vannak.
- Rövid történetek arról, hogy a rendszer hogyan használható (a felhasználó céljainak elérése érdekében).
- Egyszerű, mindenki számára érthető forma.
- Elsősorban szöveges dokumentum, de grafikus UML megjelenítése is használatos.
  - Szülője Ivar Jacobson (*three amigos*), 1986
  - Mai use-case guru: Alistair Cockburn ([www.usecases.org](http://www.usecases.org))

## Példa: Process Sale

**Process Sale:** A customer arrives at a checkout with items to purchase. The cashier uses the POS system to record each purchased item. The system presents a running total and line-item details. The customer enters payment information, which the system validates and records. The system updates inventory. The customer receives a receipt from the system and then leaves with the items.

- Ez egy ún. rövid használati eset forma. Ennél részletesebb leírások is használatosak.

## Mi a használati eset? II.

- An **actor** is something with behavior, such as a person (identified by role), computer system, or organization; for example, a cashier.
- A **scenario** is a specific sequence of actions and interactions between actors and the system under discussion; it is also called a **use case instance**.
  - Pl.: sikeres készpénzes vásárlás, sikertelen vásárlás bankkártya visszautasítás miatt.
- A **use case** is a collection of related success and failure scenarios that describe actors using a system to support a goal.

## Mi a használati eset? III.

- Fő kérdés: Hogyan segíti a rendszer a felhasználót célja elérésében (jól azonosítható eredmény)?
- **Nem** a követelmények száraz felsorolása.
- A használati eset a követelmények (de nem az összes követelmény) megfogalmazásának egy módja.
  - Funkcionális követelmények!
  - **F** a FURPS+-ból

# A használati esetek fajtái I.

Láthatóság szempontjából: **Black box** – white box

- ❑ Black box: a rendszer feladatait írja le, de nem azt, hogy ezt a rendszer belül hogyan oldja meg.
- ❑ A „hogyan” kérdésre nem a követelménytervezés során kell válaszolni!

Black-box style	Not
The system records the sale.	The system writes the sale to a database. ...or (even worse): The system generates a SQL INSERT statement for the sale...

# A használati esetek fajtái II.

A forma szempontjából:

- **brief**—terse one-paragraph summary, usually of the main success scenario.
- **casual**—informal paragraph format. Multiple paragraphs that cover various scenarios.
- **fully dressed**—the most elaborate. All steps and variations are written in detail, and there are supporting sections, such as preconditions and success guarantees.

## Példa: rövid forma

**Process Sale:** A customer arrives at a checkout with items to purchase. The cashier uses the POS system to record each purchased item. The system presents a running total and line-item details. The customer enters payment information, which the system validates and records. The system updates inventory. The customer receives a receipt from the system and then leaves with the items.

# Példa: “kényelmes” (casual) forma

## Process Sale

- *Main Success Scenario:* A customer arrives at a checkout with items to purchase. The cashier uses the POS system to record each item ...
- *Alternate Scenarios:*
  - If the credit authorization is reject, inform the customer and ask for an alternate payment method.
  - If the item identifier is not found in the system, notify the Cashier and suggest manual entry of the identifier code (perhaps it is corrupted).
  - If the system detects failure to communicate with the external tax calculator system, ...

# Részletes forma (template)

**Use Case :**

**Primary Actor:**

**Stakeholders and Interests:**

**Preconditions:**

**Success Guarantee (Postconditions):**

**Main Success Scenario (or Basic Flow):**

**Extensions (or Alternative Flows):**

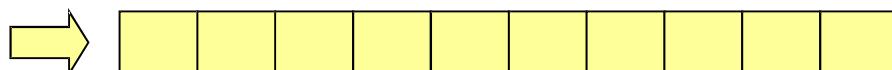
**Special Requirements:**

**Technology and Data Variations List:**

**Frequency of Occurrence:**

**Open Issues:**

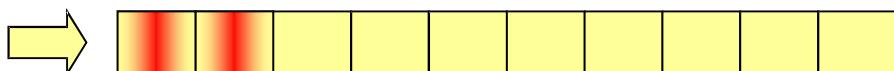
- [www.usecases.org](http://www.usecases.org)



# Példa: részletes forma (1)

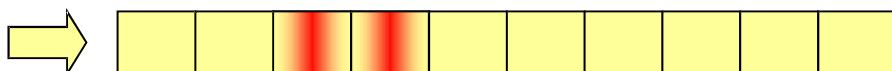
## Use Case UC1: Process Sale

- ❑ **Primary Actor:** Cashier
- ❑ **Stakeholders and Interests:**
  - **Cashier:** Wants accurate, fast entry, and no payment errors, as cash drawer shortages are deducted from his/her salary.
  - **Salesperson:** Wants sales commissions updated.
  - **Customer:** Wants purchase and fast service with minimal effort. Wants proof of purchase to support returns.
  - **Company:** Wants to accurately record transactions [...]
  - **Government Tax Agencies:** Want to collect tax [...]
  - **Payment Authorization Service:** Wants to receive digital authorization requests in the correct format and protocol. Wants to accurately account for their payables to the store.



## Példa: részletes forma (2)

- **Preconditions:** Cashier is identified and authenticated.
- **Success Guarantee (Postconditions):** Sale is saved. Tax is correctly calculated. Accounting and Inventory are updated. Commissions recorded. Receipt is generated. Payment authorization approvals are recorded.



# Példa: részletes forma (3)

- **Main Success Scenario (or Basic Flow):**

1. Customer arrives at POS checkout with goods and/or services to purchase.
2. Cashier starts a new sale.
3. Cashier enters item identifier.
4. System records sale line item and presents item description, price, and running total. Price calculated from a set of price rules.

*Cashier repeats steps 3-4 until indicates done.*

5. System presents total with taxes calculated.

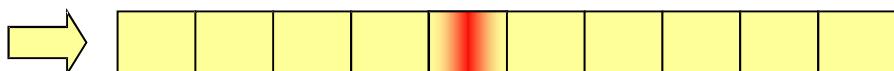
...



# Példa: részletes forma (4)

## □ Main Success Scenario (cont'd):

6. Cashier tells Customer the total, and asks for payment.
7. Customer pays and System handles payment.
8. System logs completed sale and sends sale and payment information to the external Accounting system (for accounting and commissions) and Inventory system (to update inventory).
9. System presents receipt.
10. Customer leaves with receipt and goods (if any).



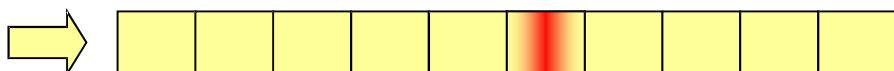
# Példa: részletes forma (5)

## □ Extensions (or Alternative Flows):

\*a. At any time, System fails:

To support recovery and correct accounting, ensure all transaction sensitive state and events can be recovered from any step of the scenario.

1. Cashier restarts System, logs in, and requests recovery of prior state.
2. System reconstructs prior state.
  - 2a. System detects anomalies preventing recovery:
    1. System signals error to the Cashier, records the error, and enters a clean state.
    2. Cashier starts a new sale.



# Példa: részletes forma (6)

## ❑ Extensions (cont'd):

3a. Invalid identifier:

1. System signals error and rejects entry.

3b. There are multiple of same item category and tracking unique item identity not important (e.g., 5 packages of veggie-burgers):

1. Cashier can enter item category identifier and the quantity.

3-6a: Customer asks Cashier to remove an item from the purchase:

1. Cashier enters item identifier for removal from sale.
2. System displays updated running total.

3-6b. Customer tells Cashier to cancel sale:

1. Cashier cancels sale on System.



# Példa: részletes forma (7)

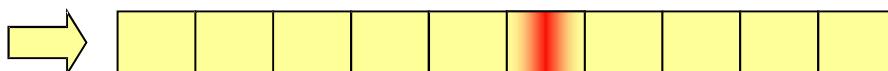
- **Extensions (cont'd):**

- 7a. Paying by cash:

1. Cashier enters the cash amount tendered.
2. System presents the balance due, and releases the cash drawer.
3. Cashier deposits cash tendered and returns balance in cash to Customer.
4. System records the cash payment.

- 7b. Paying by check...

- ...

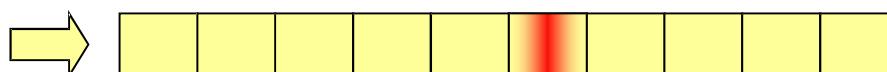


# Példa: részletes forma (8)

- **Extensions (cont'd):**

- 7c. Paying by credit:

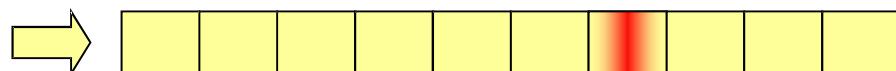
- 1. Customer enters their credit account information.
    - 2. System sends payment authorization request to an external Payment Authorization Service System, and requests payment approval.
    - 2a. System detects failure to collaborate with external system:
      - 1. System signals error to Cashier.
      - 2. Cashier asks Customer for alternate payment.
    - 3. System receives payment approval and signals approval to Cashier.
    - 3a. System receives payment denial:
      - 1. System signals denial to Cashier.
      - 2. Cashier asks Customer for alternate payment.
    - 4. System records the credit payment, which includes the payment approval.
    - 5. System presents credit payment signature input mechanism.
    - 6. Cashier asks Customer for a credit payment signature. Customer enters signature.



# Példa: részletes forma (9)

## □ Special Requirements:

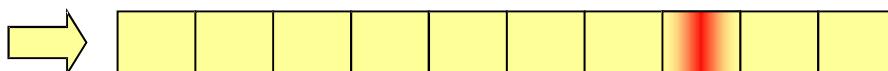
- Touch screen UI on a large flat panel monitor. Text must be visible from 1 meter.
- Credit authorization response within 30 seconds 90% of the time.
- Somehow, we want robust recovery when access to remote services such the inventory system is failing.
- Language internationalization on the text displayed.
- Pluggable business rules to be insertable at steps 3 (enter item ID) and 7 (payment).



# Példa: részletes forma (10)

## □ Technology and Data Variations List:

- 3a. Item identifier entered by bar code laser scanner (if bar code is present) or keyboard.
- 3b. Item identifier may be any UPC, EAN, JAN, or SKU coding scheme.
- 7a. Credit account information entered by card reader or keyboard.
- 7b. Credit payment signature captured on paper receipt. But within two years, we predict many customers will want digital signature capture.



# Példa: részletes forma (11)

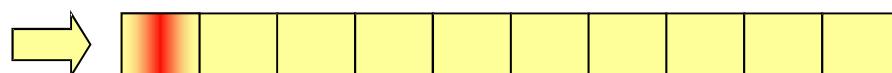
- **Frequency of Occurrence:** Could be nearly continuous.
- **Open Issues:**
  - What are the tax law variations?
  - Explore the remote service recovery issue.
  - What customization is needed for different businesses?
  - Must a cashier take their cash drawer when they log out?
  - Can the customer directly use the card reader, or does the cashier have to do it?



# Primary Actor

**Primary Actor:** Cashier

- Ki a használati eset legfontosabb szereplője?



# Stakeholders and Interests

## Stakeholders and Interests:

**Salesperson:** Wants sales commissions updated.

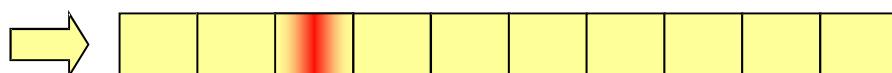
- Nagyon fontos elem!
- „The [system] operates a contract between stakeholders, with the use cases detailing the behavioral parts of that contract...The use case, as the contract for behavior, captures *all and only* the behaviors related to satisfying the stakeholders' interests” (A. Cockburn).
- Segít megválaszolni a kérdést: mi legyen a használati esetben?
- Válasz: Az, ami segít kielégíteni a résztvevők igényeit.
- **Salesperson:** Wants sales commissions updated.
  - Gondolnánk erre később?



# Előfeltételek

**Preconditions:** Cashier is identified and authenticated.

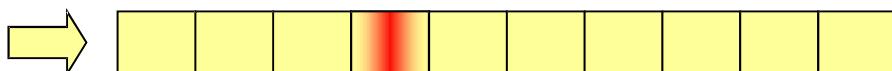
- A szcenárió kezdete előtt igaznak kell lennie
- Az előfeltételeket a használati eset során nem teszteljük
- Pl. egy előző szcenárió sikeres végrehajtását feltételezi
  - pl.: a pénztáros azonosította magát és bejelentkezett a rendszerbe
- Csak a fontos előfeltételeket írjuk fel
  - **nem kell** pl.: „a rendszer be van kapcsolva”



# Sikeres végrehajtás

**Success Guarantee (Postconditions):** Sale is saved.  
Tax is correctly calculated. Accounting and Inventory  
are updated. Commissions recorded. Receipt is  
generated.

- Mi az, aminek igaznak kell lennie a használati eset sikeres végrehajtása után?
- A részvényesek igényeit kell kielégítenie.



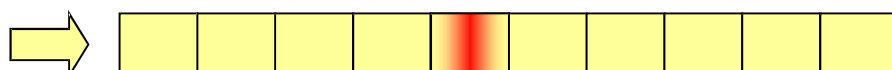
# A fő szcenárió

## Main Success Scenario

2. Cashier starts a new sale.
3. Cashier enters item identifier.

A részvényesek igényeit kielégítő, tipikus, sikeresen végrehajtott eset leírása

- Általában nem tartalmaz elágazást, vagy ugrásokat
- Ezeket az Extensions szekció tartalmazza
- Háromféle lépést tartalmaz:
  - Aktorok közötti interakció
  - Ellenőrzés (a rendszer által)
  - A rendszer állapotváltozása (pl. egy adat rögzítése vagy változtatása)



# Kiterjesztések

## Extensions:

3a. Invalid identifier:

1. System signals error and rejects entry.

3b. There are multiple of same item category and tracking unique item identity not important (e.g., 5 packages of veggie-burgers):

1. Cashier can enter item category identifier and the quantity.

- Az összes többi szcenárió leírása (sikeres vagy sikertelen)
- Általában sokkal hosszabb, mint a fő szcenárió
- A fő szcenárió lépéseihez kapcsolódnak
  - pl. elágazások, lsd. 3 → 3a. De pl. 3-6a, \*a.
- minden kiterjesztés 2 részből áll:
  - **Feltétel** (olyasmi, amit a rendszer vagy aktor észlelni tud)
    - Jó: System detects failure to communicate with external tax calculation system service:
    - Rossz: External tax calculation system not working:
  - **Kezelés**



# Speciális követelmények

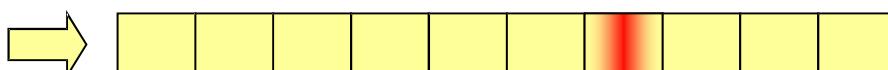
## Special Requirements:

Touch screen UI on a large flat panel monitor.

Text must be visible from 1 meter.

Credit authorization response within 30 seconds  
90% of the time.

- A használati esethez köthető *nem funkcionális* követelmények, minőségi jellemzők, kényszerek
  - Teljesítmény
  - Megbízhatóság
  - Használhatóság
  - Tervezési kényszerek
  - ...
- Alternatíva: ezen követelmények a *Supplementary Specification* dokumentumban is rögzíthetők.

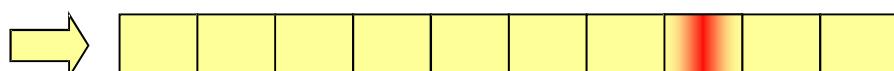


# Technológiai- és adatvariációk

## **Technology and Data Variations List:**

- 3a. Item identifier entered by bar code laser scanner (if bar code is present) or keyboard.
- 7a. Credit account information entered by card reader or keyboard.

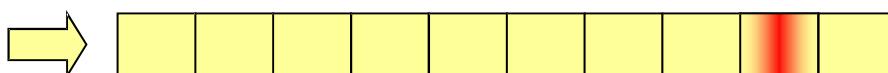
- Ezek a „hogyan” kérdésre adandó tervezési megfontolások!
- Általában kerülendők,
  - de időnként elkerülhetetlenek (külső kényszer)
  - vagy nyilvánvaló tények (pl. I/O)



# Előfordulási gyakoriság

**Frequency of Occurrence:** Could be nearly continuous.

- Milyen sűrűn következik be a használati eset?
- Mennyire fontos?



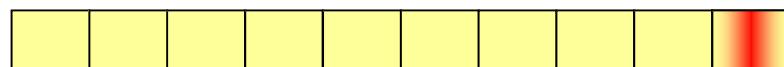
# Nyitott kérdések

## **Open Issues:**

What are the tax law variations?

...

- Nyitott, később tisztázandó kérdések



# Az Elemi Üzleti Folyamat

- Miről szóljon a használati eset? Ilyen mélységű esteket írunk le?
- Válasz: A legmegfelelőbb szint az elemi üzleti folyamatok szintje
- Elementary Business Process (EBP)
  - „A task performed by one person in one place at one time, in response to a business event, which adds measurable business value and leaves the data in a consistent state. e.g., Approve Credit or Price Order”

# Az Elemi Üzleti Folyamat

## ■ Melyik lehet EBP?

- Negotiate a Supplier Contract



- Handle Returns



- Log In



Általános hiba: túl sok használati eset, amelyek túl alacsony szintűek; ezek részfunkciók, egy EBP részei csak.

# Használati esetek és felhasználói célok

- A felhasználónak valamilyen **célja** van, ezért akarja a rendszert használni.
- Az EBP-szintű használati eseteket **felhasználói cél**-szintű használati esetnek is hívják (user goal-level use case).
- Ajánlott eljárás
  - A felhasználói célok azonosítása
  - Mindegyikhez használati eset definiálása
- Helytelen kérdés: mi a használati eset?
- Helyes kérdés: mi a felhasználó (v. aktor) célja?

# Felhasználói célok

## Miért előnyös stratégia a célok kutatása?

- „Mit csinál?”
  - A válasz nagy valószínűséggel a jelen rendszer megoldásait tükrözi.
- „Mi a célja?”
  - Erre a kérdésre adott válaszok új, jobb megoldások felé nyithatnak utat, a valódi üzleti értékre fókuszál, kideríti, hogy a részvényszek *valójában* mit akarnak.
  - Hasznos módosítás: mi ennek a célnak célja? (felsőbb szintű célok kiderítése, EBP szintjének meghatározása)

# Példa: Mik a felhasználói célok

**System analyst:** "What are some of your goals in the context of using a POS system?"

**Cashier:** "One, to quickly log in..."

Alacsony  
szint, nem  
EBP

**System analyst:** "What do you think is the higher level goal motivating logging in?"

**Cashier:** "I'm trying to identify myself to the system, so it can validate that I'm allowed to use the system for sales capture and other tasks."

Ez jónak  
tűnik

**System analyst:** "Higher than that?"

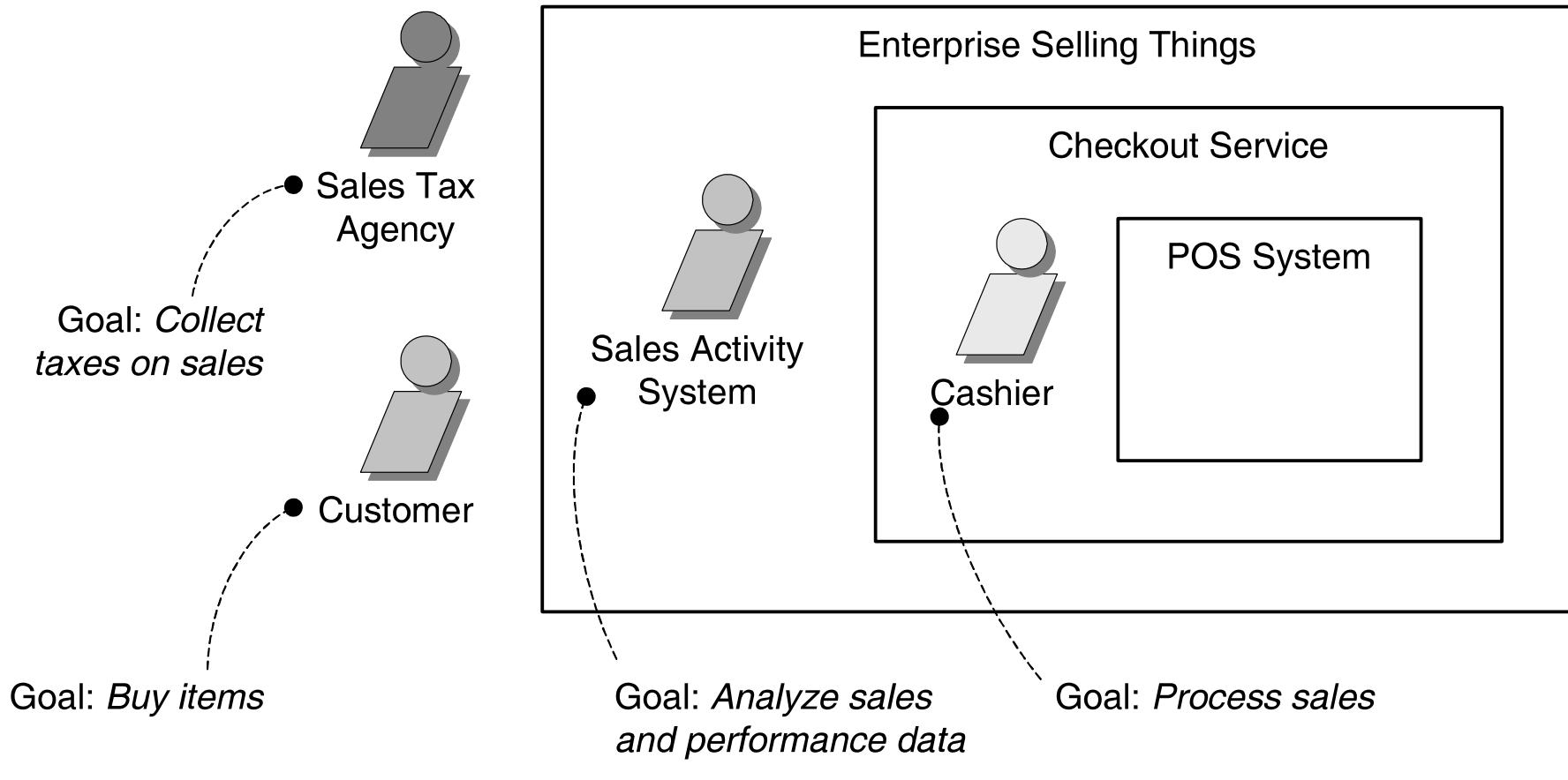
**Cashier:** "To prevent theft, data corruption, and display of private company information."

Ez már vállalat-  
szintű cél, nem  
felhasználói

# Javasolt eljárás

1. **Choose the system boundary.** Is it just a software application, the hardware and application as a unit, that plus a person using it, or an entire organization?
2. **Identify the primary actors** – those that have user goals fulfilled through using services of the system.
3. For each, **identify their user goals.** Raise them to the highest user goal level that satisfies the EBP guideline.
4. **Define use cases** that satisfy user goals; name them according to their goal. Usually, user goal-level use cases will be one-to-one with user goals.

# Kapcsolat a rendszer határa, az elsődleges aktorok és céljaik között



# Tanácsok használati esetek írásához

- Állandó kommunikáció a felhasználóval
  - XP: „*User full-time on the project, in the project room.*”
- A használati eseteket a felhasználó interfésekre való utalás nélkül írjuk
  - Fókusz a valódi célokra!



1. Administrator identifies self.
2. System authenticates identity.

1. Adminstrator enters ID and password in dialog box (see Picture 3).
2. System authenticates Adminstrator.
3. System displays the "edit users" window (see Picture 4).



# Aktorok

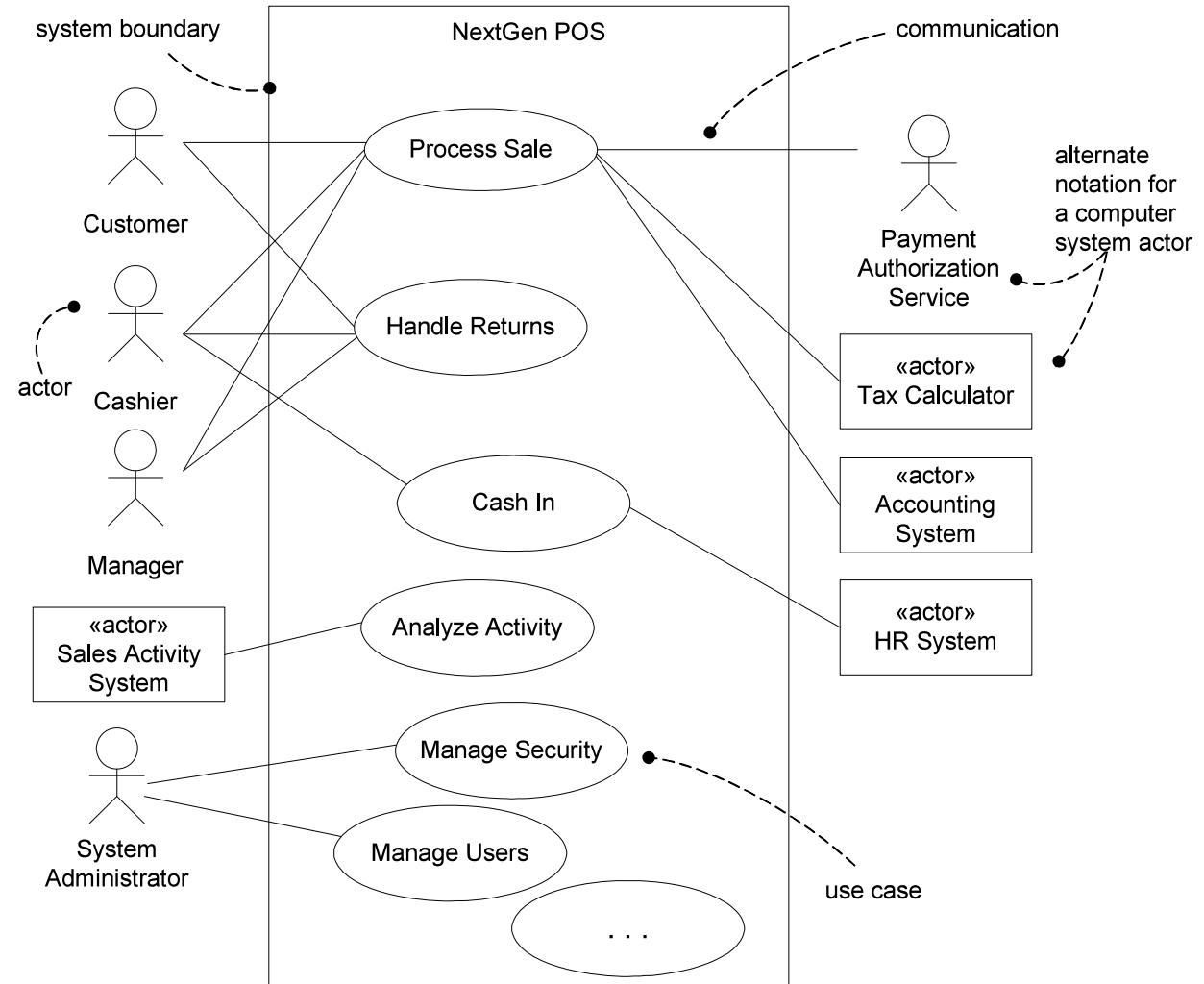
**Aktor:** bárki vagy bármi, ami viselkedéssel bír

- Emberek, szervezetek, szoftver, gépek

Három alapvető aktor:

- **Primary actor** – has user goals fulfilled through using services of the SuD. For example, the cashier.
  - Why identify? To find user goals, which drive the use cases.
- **Supporting actor** – provides a service (for example, information) to the SuD. The automated payment authorization service is an example. Often a computer system, but could be an organization or person.
  - Why identify? To clarify external interfaces and protocols.
- **Offstage actor** – has an interest in the behavior of the use case, but is not primary or supporting; for example, a government tax agency.
  - Why identify? To ensure that *all* necessary interests are identified and satisfied. Offstage actor interests are sometimes subtle or easy to miss unless these actors are explicitly named.

# Használati eset diagramok

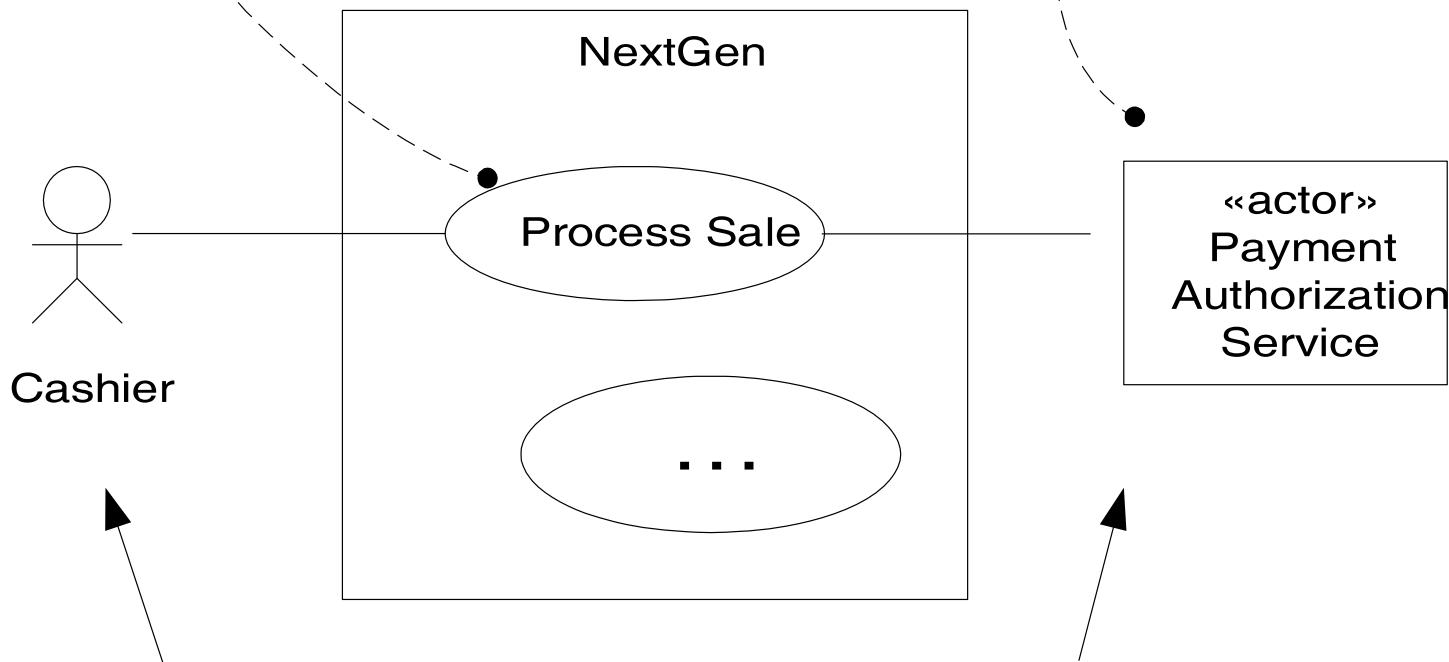


Csak illusztráció, a lényeg a szöveges leírás!

# Használati eset diagramok - jelölések

For a use case context diagram, limit the use cases to user-goal level use cases.

Show computer system acto with an alternate notation to human actors.

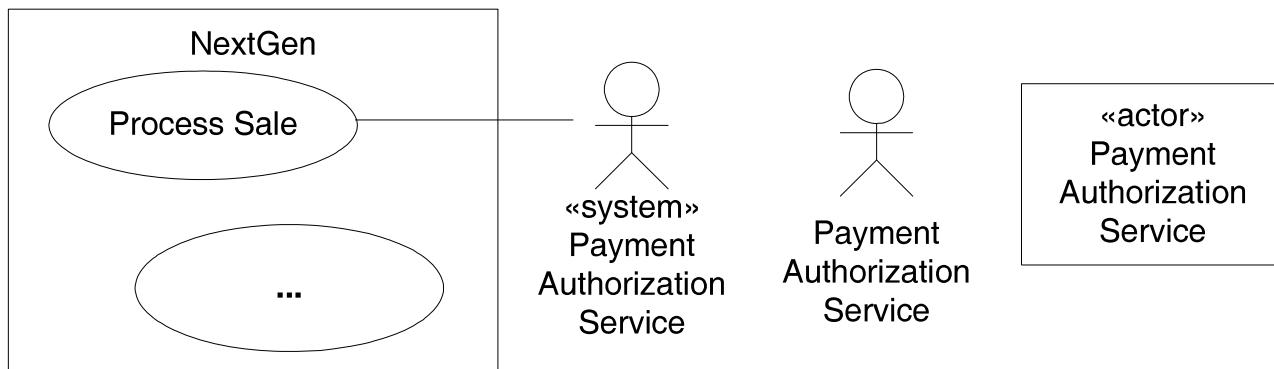


primary actors on  
the left

supporting actors  
on the right

# Használati eset diagramok - jelölések

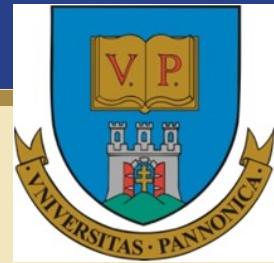
## Alternatívák az aktorok jelölésére



Some UML alternatives to illustrate external actors that are other computer systems.

The class box style can be used for any actor, computer or human. Using it for computer actors provides visual distinction.

# A rendszerfejlesztés korszerű módszerei

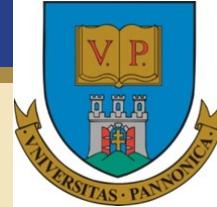


## Kommunikáció a szerver és a kliensek között

Pannon Egyetem, Műszaki Informatikai Kar

2022

# Bevezetés



- A kliens-szerver szoftverarchitektúra egy sokoldalú, üzenetalapú és moduláris infrastruktúra amely azért alakult ki, hogy a használhatóságot, rugalmasságot, együttműködési lehetőségeket és bővíthetőséget megnövelje a centralizált, nagyszámítógépes, időosztásos rendszerekhez képest.
- A kliens és a szerver között fontos a kommunikáció megléte.
- Példák: <https://github.com/pekmil/>  
<https://github.com/horvath-adam>

# REST



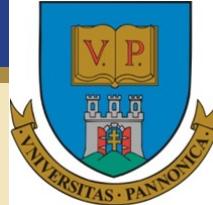
- HTTP protokollra fejlesztett kommunikációs architektúra típus
- Kliens és szerver közti kommunikáció megvalósítására használható
- Kihasználja a HTTP állapotkódokat (pl. 404 Not Found, 200 Ok stb.) és metódusokat (pl. GET, POST, PUT, DELETE)
- A kérések URI-k használatával történnek
  - Az URI-k egységes interfész biztosítanak a kliens számára
- minden kérésre azonos formátumban reagál a szerver
  - JSON
  - XML
  - HTML
- A kéréseknek állapotmentesnek kell lenniük
  - A szerver nem ismeri a kliens állapotát két kérés között
  - Az autorizáció folyamatos figyelmet igényel!

# REST megszorítások



- Kliens-szerver architektúra
- Állapotmentesség
- Gyorsítótárazhatóság
- Réteges felépítés
- Egységes interfész
- Igényelt kód (opcionális)

# REST megszorítások



## ■ Kliens-szerver architektúra

- A kliensek el vannak különítve a szerverektől egy egységes interfész által.
- A kliensek nem foglalkoznak adattárolással, ami a szerver belső ügye marad -> a kliens kód hordozhatósága megnő.
- A szerverek nem foglalkoznak a felhasználói felülettel vagy a kliens állapotával -> a szerverek egyszerűbbek és még skálázhatóbbak lehetnek.
- A szerverek és kliensek áthelyezhetők és fejleszthetők külön-külön is, egészen addig amíg az interfész nem változik meg.

# REST megszorítások



## ■ Állapotmentesség

- A szerveren nem tárolják a kliens állapotát a kérések között.
- minden egyes kérés bármelyik klienstől tartalmazza az összes szükséges információt a kérés kiszolgálásához, és minden állapotot a kliens tárol.
- A szerver oldali erőforrás-állapotok URL által címezhetőek legyenek.
- (Ez nem csak a szerver felügyeletét teszi lehetővé, de megbízhatóbbá teszi őket a hálózati meghibásodásokkal szemben, valamint tovább fokozza a skálázhatóságot.)

# REST megszorítások



## ■ Gyorsítótárazhatóság

- A kliensek itt is képesek gyorsítótárazni a válaszokat.
- A válaszoknak impliciten vagy expliciten tartalmazniuk kell, hogy gyorsítótárazhatóak-e vagy sem.
- Elkerülhető, hogy a kliens téves vagy elavult adatokat használjon fel újra.
- Egy jól menedzselt gyorsítótár lehetővé teszi, hogy teljesen megkerüljünk egyes kliens-szerver interakciókat, továbbá megnöveli a rendszer skálázhatóságát és a teljesítményét.

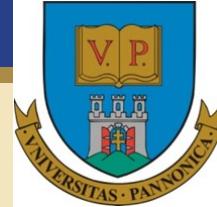
# REST megszorítások



## ■ Réteges felépítés

- Egy kliens általában nem tudja megmondani, hogy direkt csatlakozott-e a végpont szerverhez, vagy közvetítő segítségével.
- A közvetítő szerverek megnövelhetik a rendszer skálázhatóságát terheléseloszlás kiegyenlítéssel és megosztott gyorsítótárak használatával.

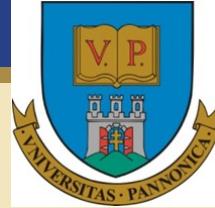
# REST megszorítások



## ■ Egységes interfész

- A kliens és a szerver között egyszerűsíti és kettéválasztja az architektúrát.
- Lehetővé teszi, hogy egymástól függetlenül fejlődjenek az egyes részek.
- Az interfész négy irányadó elve:
  - Erőforrások azonosítása
  - Erőforrások manipulációja ezeken a reprezentációkon keresztül
  - Önleíró üzenetek
  - Hipermédia, mint az alkalmazásállapot motorja

# REST megszorítások

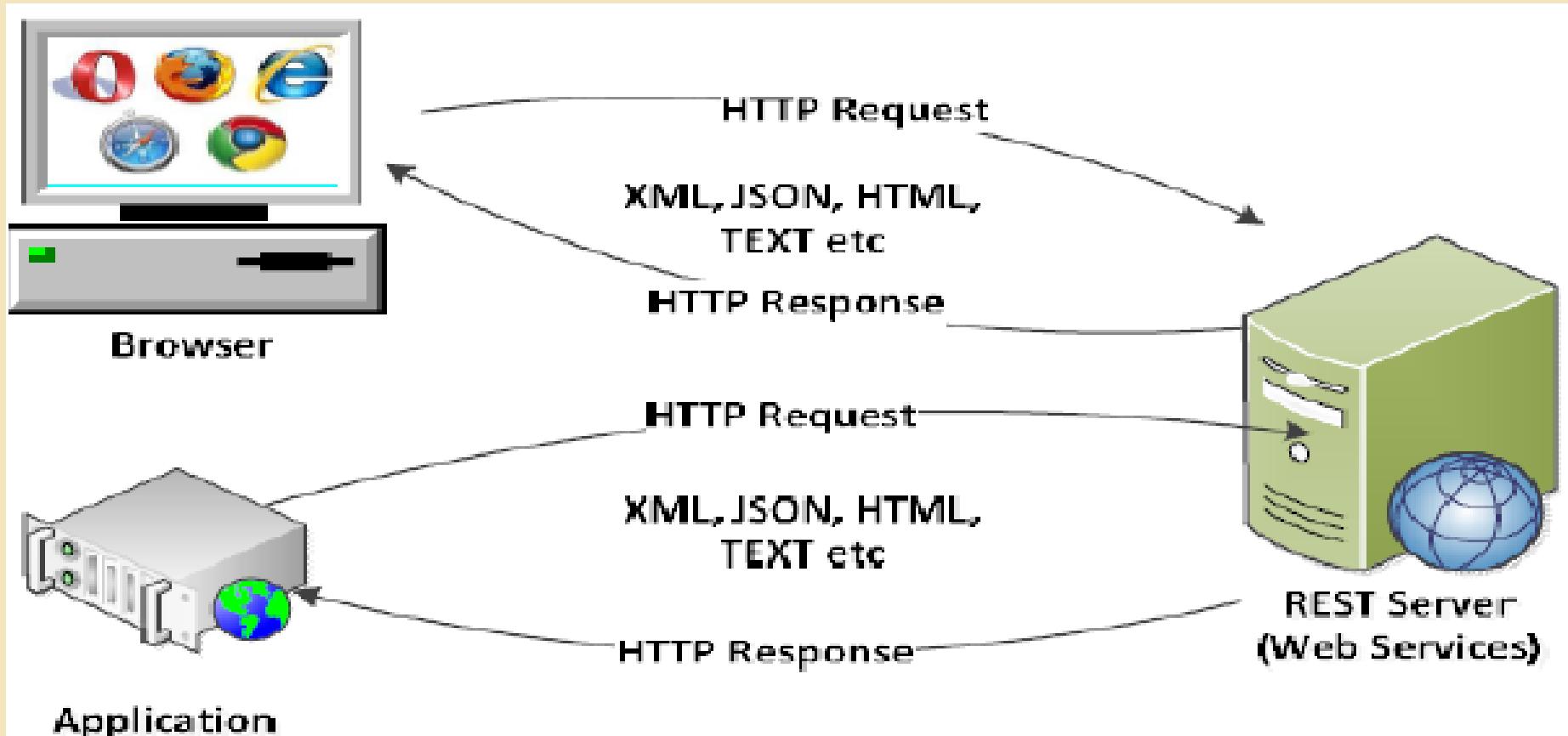
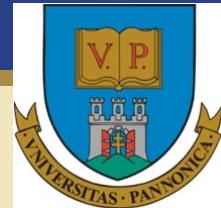


## ■ Igényelt kód (opcionális)

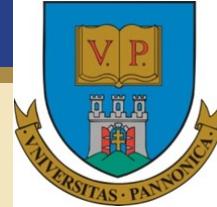
- A szerverek képesek időlegesen kiterjeszteni vagy testre szabni egy kliens funkcionálitását, programrészek átadásával, amelyeket a kliens futtatni képes.
- Ide tartoznak az előre fordított komponensek (pl. Java appletek) és a kliensoldali szkriptek (pl. JavaScript).
- A REST architektúra egyetlen opcionális megszorítása az igényelt kód.

## ■ Ha egy szolgáltatás sért bármely más megszorítást, azt nem lehet feltétlenül "RESTful"-nak nevezni.

# RESTful architektúra

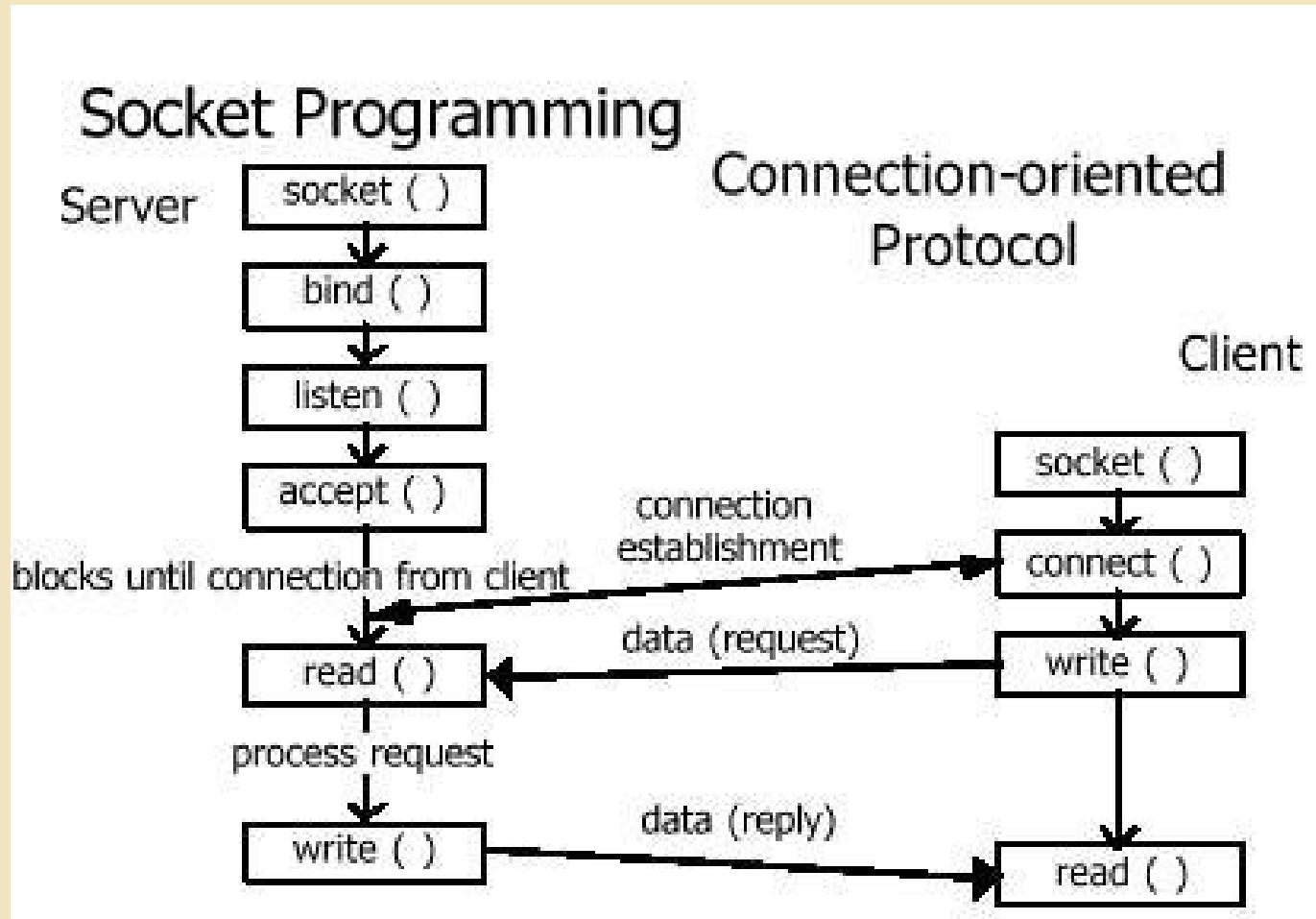


# Kommunikáció socketen



- A socket egy Internet Protocol-alapú hálózatban, valamely kétirányú folyamatközi kommunikációs (IPC) hálózati folyam végpontja.
- A szerverek induláskor listening állapotban lévő socketeket hoznak létre. Ezek a socketek arra várakoznak, hogy egy kliens megszólítsa őket.
- Egy TCP szerver minden klienshez létrehoz egy gyerekprocesszt, és a TCP-kapcsolatok a gyerekprocessz és a kliens között épülnek ki.
  - Ezek akkor tekinthetők „élő” (established) állapotúnak, amikor a távoli sockettel kiépül a socket-socket közötti munkamenet, ami egy duplex bytestream átvitelét biztosítja.
- A socket címzése rendezett 4-essel és egy portszámmal történik

# Kommunikáció socketen

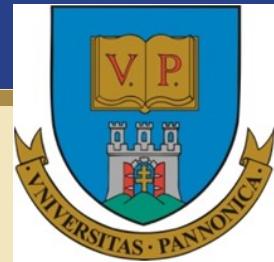


# WebSocket



- Kétirányú, duplex kommunikációs csatornák kiépítését teszi lehetővé egyetlen TCP protokollon keresztül
- Fő motivációja: a böngészőben futó alkalmazás képes legyen a szerverrel való kétirányú kommunikációra.
- Tipikus felhasználási példája egy chat alkalmazás.
- A WebSocket kapcsolat kiépítéséhez a kliens egy kézfogási kérelmet küld (handshake request), amire a szerver kézfogási válasszal (handshake response) felel.

# A rendszerfejlesztés korszerű módszerei



Multitier architecture,  
application layers

Pannon Egyetem, Műszaki Informatikai Kar

2021

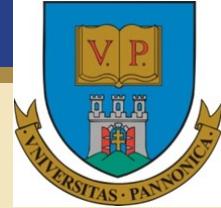
[https://github.com/horvath-adam/serversidedotnet/tree/lesson5\\_services/EventApp](https://github.com/horvath-adam/serversidedotnet/tree/lesson5_services/EventApp)

# Common web application architectures



- Most traditional applications are deployed as single units corresponding to an executable or a single web application running within a single appdomain.
- This approach is the simplest deployment model and serves many internal and smaller public applications very well.
- However, even given this single unit of deployment, most non-trivial business applications benefit from some logical separation into several layers.

# What are layers?



- As applications grow in complexity, one way to manage that complexity is to break up the application according to its responsibilities or concerns.
- This approach follows the separation of concerns principle and can help keep a growing codebase organized so that developers can easily find where certain functionality is implemented.
- Layered architecture offers a number of advantages beyond just code organization, though.

# Traditional "N-Layer" architecture applications



## Application Layers

User Interface

Business Logic

Data Access

# UI Layer



- The user interface layer in an application is the entry point for the application.
- UI Layer types
  - Controllers
  - Filters
  - Views
  - ViewModels

# Controllers



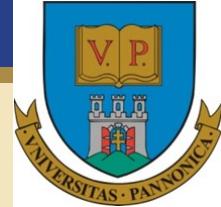
- A controller is used to define and group a set of actions. An action (or action method) is a method on a controller which handles requests.
- Controllers logically group similar actions together.
- This aggregation of actions allows common sets of rules, such as routing, caching, and authorization, to be applied collectively.
- Requests are mapped to actions through routing.

# Controllers



- Controllers should follow the Explicit Dependencies Principle.
  - Methods and classes should explicitly require any collaborating objects they need in order to function correctly.
- There are a couple of approaches to implementing this principle.
  - If multiple controller actions require the same service, consider using constructor injection to request those dependencies.
  - If the service is needed by only a single action method, consider using Action Injection to request the dependency.

# Controllers



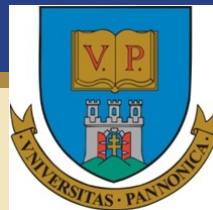
- Within the Model-View-Controller pattern, a controller is responsible for the initial processing of the request and instantiation of the model.
- Generally, business decisions should be performed within the model.
- The controller takes the result of the model's processing (if any) and returns either the proper view and its associated view data or the result of the API call.

# Controllers



- The controller is a UI-level abstraction.
  - Its responsibilities are to ensure request data is valid and to choose which view (or result for an API) should be returned.
- In well-factored apps, it doesn't directly include data access or business logic. Instead, the controller delegates to services handling these responsibilities.

# Simple Controller Example



```
namespace Example
{
    [Route("api/[controller]")]
    [ApiController]
    0 references | 0 changes | 0 authors, 0 changes
    public class SimpleExampleController : ControllerBase
    {
        // GET: api/SimpleExample
        [HttpGet]
        0 references | 0 changes | 0 authors, 0 changes
        public IEnumerable<string> Get()
        {
            return new string[] { "value1", "value2" };
        }

        // GET: api/SimpleExample/5
        [HttpGet("{id}", Name = "Get")]
        0 references | 0 changes | 0 authors, 0 changes
        public string Get(int id)
        {
            return "value";
        }

        // POST: api/SimpleExample
        [HttpPost]
        0 references | 0 changes | 0 authors, 0 changes
        public void Post([FromBody] string value)
        {
        }
    }
}
```

```
        // PUT: api/SimpleExample/5
        [HttpPut("{id}")]
        0 references | 0 changes | 0 authors, 0 changes
        public void Put(int id, [FromBody] string value)
        {
        }

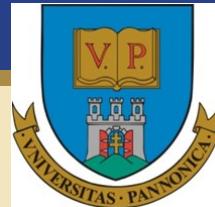
        // DELETE: api/ApiWithActions/5
        [HttpDelete("{id}")]
        0 references | 0 changes | 0 authors, 0 changes
        public void Delete(int id)
        {
        }
    }
}
```

# Business logic layer



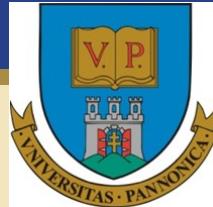
- The Business logic holds the business model, which includes entities, services, and interfaces.
- These interfaces include abstractions for operations that will be performed using Infrastructure, such as data access, file system access, network calls, etc.
- Business logic layer types:
  - Entities (business model classes that are persisted)
  - Interfaces
  - Services
  - DTOs

# Separating Concerns



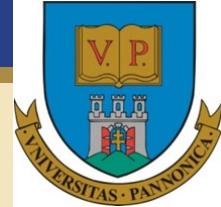
- When you build an application, you should not place your database logic inside your controller actions.
- Mixing your database and controller logic makes your application more difficult to maintain over time.
- The recommendation is that you place all of your database logic in a separate data access layer.
- Creating a service layer enables you to maintain a clean separation of concerns.
- Controllers are responsible for the initial processing of the request and instantiation of the model and the service is responsible for your business logic.

# Service Layer



- A service layer is an additional layer in an application that mediates communication between a controller and repository layer.
- The service layer contains business logic. In particular, it contains validation logic.
- The controller use the service layer instead of the data access layer.
- The controller layer talks to the service layer.
- The service layer talks to the data access (unit of work or repository) layer.
- Each layer has a separate responsibility.

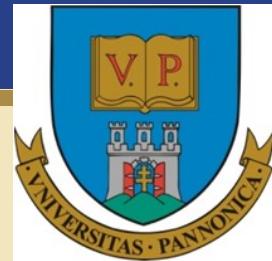
# Data access layer



- The Data access layer typically includes data access implementations.
- For example in a typical ASP.NET Core web application, these implementations include the Entity Framework (EF) DbContext, any EF Core Migration objects that have been defined, and data access implementation classes.
- The most common way to abstract data access implementation code is through the use of the Repository design pattern.
- Data access layer types:
  - ORM (EF Core) types (DbContext, Migration)
  - Data access implementation types (Repositories)
  - Infrastructure- or environment-specific services (for example, FileLogger or SmtpNotifier)

# A rendszerfejlesztés korszerű módszerei

Authentikáció &  
authorizáció



Pannon Egyetem, Műszaki Informatikai Kar

2023

# Bevezetés



- Az authentikáció és az authorizáció szükségesek a rendszer, az alkalmazás, vagy az adatok védelmére.
- Az authentikáció a felhasználó azonosítását jelenti, azaz meghatározza, hogy a felhasználó valóban az, akinek mondja magát.
- Az authorizáció pedig a hozzáférési jogok ellenőrzését jelenti, azaz meghatározza, hogy a felhasználónak mely adatokhoz, funkcióhoz, vagy információhoz van joga.

# Bevezetés



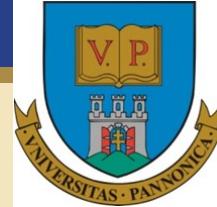
- Általánosságban egy rendszer például jelszavakat használ az authentikációhoz, hogy megállapítsa a felhasználó valódiságát.
- Az autorizáció a jogokat adja meg, például, hogy a felhasználó módosíthatja-e a rendszerben lévő adatokat, vagy csak megtekintheti azokat.
- Az authentikáció és autorizáció nélküli rendszerek sérülékenyek a véletlenszerű hozzáférésre, az adatokkal való visszaélésre, és a jogosulatlan hozzáférésre, így fontos, hogy minden adatbiztonsági rendszernek legyen valamilyen védelme.

# Authentikáció



- Az autentikáció az a folyamat, amelynek során ellenőrizni lehet, hogy valaki vagy valami valóban az-e, aminek/akinek mondja magát.
- A leggyakrabban ismert authentikációs eljárások:
  - Felhasználónév és jelszó
  - Kétfaktoros hitelesítés (például SMS- vagy akkumulátoros hitelesítő eszközök)
  - Tanúsítványalapú hitelesítés

# Felhasználónév és jelszó alapú authentikáció



- Leggyakrabban használt azonosítási módszer.
- A felhasználónak meg kell adnia felhasználónevét és jelszavát, amelyet a rendszer ellenőriz.
  - Ha a megadott adatok egyeznek a rendszer által tárolt adatokkal, akkor a felhasználó azonosítása megerősítésre kerül, és hozzáférhet a megfelelő jogosultságokkal rendelkező szolgáltatásokhoz.
- Ez a módszer könnyen hozzáférhető, egyszerűen kezelhető és elterjedt, de biztonsági szempontból korlátozott.
  - Ha a felhasználó jelszava nem biztonságos, vagy ha a jelszavakat tároló adatbázis sérült, akkor az adatok veszélynek vannak kitéve.
  - A felhasználók hajlamosak a gyenge jelszavak használatára, amelyek könnyen törhetők.
  - Ajánlott a felhasználók számára a jelszavak változtatásának rendszeres megkövetelése és a biztonság növelése érdekében a bonyolult jelszó követelményeket bevezetni.

# Egyszerű példa



```
using System;

namespace LoginExample
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Login form");
            Console.WriteLine("Enter username:");
            string username = Console.ReadLine();
            Console.WriteLine("Enter password:");
            string password = Console.ReadLine();

            if (username == "user" && password == "pass")
            {
                Console.WriteLine("Login successful!");
            }
            else
            {
                Console.WriteLine("Login failed. Incorrect username or password.");
            }

            Console.ReadLine();
        }
    }
}
```

# Egyszerű példa



- Ez a példa csak illusztrációs célokat szolgál, és nem alkalmas biztonságos jelszó tárolására, mivel a jelszavakat nyers formában tárolja.
- A valós alkalmazásokban a jelszavakat biztonságosan, jelszó kódolással kell tárolni.
- A példafájlok között a *simple login.cs* fájlban található egy példa a jelszó kódolásra.

# Kétfaktoros hitelesítés



- A kétfaktoros hitelesítés (two-factor authentication, 2FA) egy biztonságosabb authentikációs eljárás, amely a felhasználó azonosításához két különálló bizonyítékot igényel.
  - Az első bizonyíték a hagyományos jelszó,
  - A második bizonyíték lehet egy mobiltelefon, egy token vagy egy biometrikus adat, például ujjlenyomat, arcfelismerés vagy személyi azonosító szám (PIN).
- A kétfaktoros hitelesítés lehetővé teszi, hogy a rendszer ellenőrizze a felhasználó identitását, anélkül, hogy csak egy egyszerű jelszóra volna szükség.
  - Ha a felhasználó csak jelszót ad meg, akkor a hozzáférés megtagadásra kerül.
  - Ha a felhasználó helyesen adja meg a jelszót és a második bizonyítékot, akkor a hozzáférés engedélyezésre kerül.

# Kétfaktoros hitelesítés



- A kétfaktoros hitelesítés növeli a rendszer biztonságát, mivel egy rosszindulatú támadónak minden két bizonyítékot meg kell szereznie a hozzáféréshez.
- Az online szolgáltatásokban és adatbiztonsági rendszerekben egyre népszerűbb a kétfaktoros hitelesítés alkalmazása, ezzel növelik a felhasználók adatainak és információinak a biztonságát.
- A példafájlok között a *two factor authentication (2FA).cs* fájlban található egy példa a kétfaktoros hitelesítésre.

# Tanúsítványalapú hitelesítés



- A tanúsítványalapú hitelesítés a digitális tanúsítványok használatával történő hitelesítési folyamat.
- Ebben a folyamatban a kliensnek kell birtokolnia egy digitális tanúsítványt, amelynek aláírásával igazolja magát.
- A szerver ellenőrzi a tanúsítvány érvényességét és az aláírást, majd az aláírt adatok alapján hitelesíti a klienst.
- Ez a módszer biztonságosabb, mint a jelszó alapú hitelesítés, mivel a tanúsítványt csak a hitelesített személy birtokolhatja, és általában szervezetek által kiállított, hiteles tanúsítványokat használnak.
- A tanúsítványalapú hitelesítés lehetővé teszi a hitelesített személy tulajdonságainak ellenőrzését is, például a nevét, a dátumát, a tanúsítványt kibocsátó szervezetet és a érvényességi időtartamát.

# Tanúsítványalapú hitelesítés



- A példafájlok között a *certificate-based login.cs* fájlban található egy példa a tanúsítványalapú hitelesítés megoldására.
- A példa a X509Certificate2 osztályt használja a tanúsítványok betöltésére és a hitelesítés végrehajtására.
  - A példa először ellenőrzi a szerver tanúsítványának érvényességét, majd ha az érvényes, akkor hitelesíti a kliens tanúsítványát.
  - Ha mindkét tanúsítvány hitelesítése sikeres, akkor a kliens hitelesítése sikeresnek minősül.

# Tanúsítványalapú hitelesítés



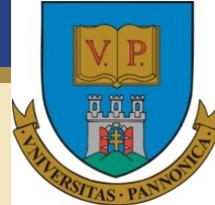
- A példafájlok között a *certificate-based login server.cs* fájlban található egy példa a tanúsítványalapú hitelesítés szerver oldali megoldására.
- A példa egy HTTP szervert hoz létre, amely tanúsítványalapú hitelesítést használ a kliens bejelentkezésének ellenőrzésére.
- A GetClientCertificate metódussal ellenőrizzük a kliens által küldött tanúsítvány hitelességét, és ha a tanúsítvány érvényes, akkor engedélyezzük a kliensnek a hozzáférést a szerverhez.

# Tanúsítványalapú hitelesítés



- A példafájlok között a *certificate-based login client.cs* fájlban található egy példa a tanúsítványalapú hitelesítés kliens oldali megoldására.
- A példában a *client.pfx* fájl tartalmazza a tanúsítványunkat és a hozzá tartozó kulcsot.
- A *clientHandler* objektum a tanúsítvány beállításait tartalmazza, beleértve a titkosítási protokollt és a tanúsítvány hozzáadását a kliensnek.
- A *HttpClient* objektumot ezután a szerverrel való kommunikációhoz használhatjuk.
- A *HttpResponseMessage* objektum tartalmazza a szerver által visszaadott választ.

# Tanúsítványalapú hitelesítés



- A példafájlok között a *certificate-based login netcore.cs* fájlban található egy példa a tanúsítványalapú hitelesítés AspNetCore környezetben történő beállítását mutatja meg.
- Ebben a példában látható, hogy a ConfigureServices metódusban a AddAuthentication és AddCertificate függvényeket hívjuk meg a tanúsítványalapú hitelesítés beállításához.
- A Configure metódusban pedig a UseAuthentication és UseAuthorization middleware-eket hívjuk meg a bejelentkezási információk ellenőrzésére.

# Authorizáció



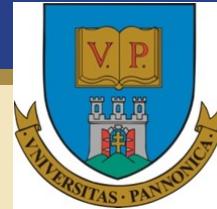
- Az authorizáció meghatározza, hogy a felhasználónak melyik műveletekre van jogosultsága.
- Az authorizációs eljárások között szerepelnek:
  - Roles-based access control (RBAC)
  - Attribute-based access control (ABAC)
  - Access control lists (ACLs)
  - Policy-based access control.

# Roles-based access control (RBAC)



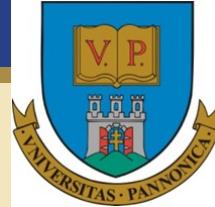
- A Roles-based access control (RBAC) egy olyan autorizációs eljárás, amely a felhasználók jogosultságait a hozzájuk rendelt szerepek alapján határozza meg.
- Az RBAC használata esetén a rendszerért felelős felhasználók különböző szerepeket hoznak létre, és ezekhez rendelnek hozzá jogosultságokat, pl. elérhető adatokat, műveleteket stb.
- A felhasználók ezek után a rendszerbe való bejelentkezéskor hozzárendelhetőek egy vagy több szerephez, és a rendszer ezen szerepek alapján ad hozzáférést a rendszer adataihoz és funkcióihoz.
- Ezzel biztosítható, hogy csak azok a felhasználók férjenek hozzá az adatokhoz és műveletekhez, akiknek ehhez jog van.

# Roles-based access control (RBAC)



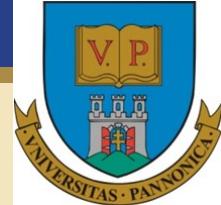
- A példafájlok között a *role-based auth.cs* fájlban található egyszerű példa az RBAC alkalmazását mutatja meg.
- Ez az alkalmazás egyszerű példa a bejelentkezést követő RBAC alapú autorizációra, amely az adatbázisban tárolt felhasználónév és jelszó alapján meghatározza a szerepköröket és megadja a hozzáférési jogokat.

# Roles-based access control (RBAC)



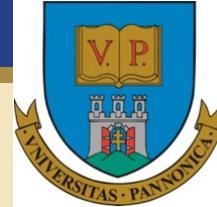
- A példafájlok között a *role-based auth netcore.cs* fájlban található példa az RBAC AspNetCore környezetben történő alkalmazását mutatja meg.
- Ebben a példában a Authorize attribútumot használjuk a kérés authentikációjának és autorizációjának ellenőrzésére.
- Az attribútum Roles tulajdonsága meghatározza, hogy a kérés mely szerepek számára lesz elérhető.
- AzHttpGet attribútum meghatározza a REST API hívás URL-jét és metódusát.

# Attribute-based access control (ABAC)



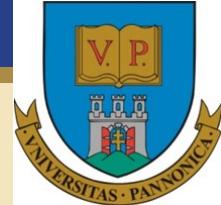
- Az Attribute-based access control (ABAC) egy autorizációs megközelítés, amely a felhasználók, az objektumok és a környezeti attribútumok (például idő, hely, stb.) jellemzőit használja az engedélyek meghatározásához.
- Az ABAC használatával a szabályok flexibilisebbek és rugalmasabbak lehetnek, mivel a rendszer minden fontos információt tartalmaz a döntéshozatalhoz.
- Az ABAC alapján történő autorizáció lehetővé teszi a szabályok dinamikus megváltoztatását, anélkül, hogy a szoftver kódját módosítanák.
- Az ABAC nagyon hasznos lehet a komplex jellegű, sok információt tartalmazó rendszerekben, ahol az összetett döntési folyamatok szükségesek a hozzáférés ellenőrzéséhez.

# Attribute-based access control (ABAC)



- A példafájlok között a *attribute-based auth.cs* fájlban található egyszerű példa az ABAC alkalmazását mutatja meg.
- Ebben a példában, a CustomAuthorizeAttribute attribútumot használjuk a AccessFinanceData művelethez való hozzáférés ellenőrzésére.
- Az AuthorizeCore metódusban ellenőrizzük, hogy a felhasználó rendelkezik-e a szükséges admin szereppel és a finance osztályal.
- Ha minden feltételnek megfelel, akkor engedélyezett a hozzáférés.

# Access control lists (ACLs)



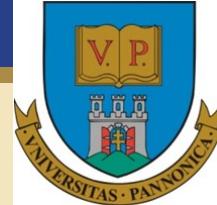
- Az Access control lists (ACLs) egy másik tipikus autorizációs megközelítés, amelynek lényege, hogy a hozzáférést a felhasználók és csoportok jogosultságaihoz rendeljük.
- Az ACL-ek általában egy táblázat formájában vannak tárolva, amely meghatározza a felhasználók vagy csoportok jogosultságait a rendszerben található objektumokra.
- minden objektumhoz tartozik egy ACL, amely meghatározza, hogy mely felhasználók vagy csoportok jogosultak olvasni, írni, módosítani vagy törölni az objektumot.
- Az ACL-ek használatával a rendszerek rugalmassabban és pontosabban tudják szabályozni a hozzáférést.

# Access control lists (ACLs)



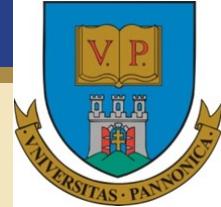
- A példafájlok között a *acl auth.cs* fájlban található egyszerű példa az ACLs alkalmazását mutatja meg.
- Ebben a példában létrehoztunk egy ACL osztályt, amely lehetővé teszi a felhasználók és jogosultságuk hozzáadását, valamint megnézhetjük, hogy egy adott felhasználónak van-e jogosultsága egy adott műveletre.
- Ezeket a függvényeket alkalmazzuk a műveletek végrehajtása előtt a későbbiekben.

# Policy-based access control



- A Policy-based access control (PBAC) egy olyan autorizációs eljárás, amely esetén a hozzáférés engedélyezését egy szabályrendszer alapján történő döntéshozatal jelenti.
- Az alapja egy szabályrendszer, amelynek az a feladata, hogy eldöntse, egy adott felhasználó vagy eszköz melyik adatokhoz vagy műveletekhez fér hozzá.
- A szabályrendszer lehetővé teszi a hozzáférés engedélyezésének konfigurálását a szerepkörök, jogosultsági szintek, vagy egyéb attribútumok figyelembevételével.

# Policy-based access control



- Például, a PBAC lehetővé teszi, hogy a rendszergazdák konfigurálják, hogy egy adott szerepkörnek milyen hozzáférési jogai legyenek adott adatbázishoz vagy más adattárházhhoz, vagy hogy milyen műveletekre legyen lehetőségük.
- A PBAC esetén a hozzáférési jogokat a felhasználói profilok, a felhasználói szerepek, vagy a műveletek alapján lehet konfigurálni, hogy a felhasználók csak a saját jogai alapján férhessenek hozzá az adatokhoz vagy a műveletekhez.

# Policy-based access control



- A példafájlok között a *policy-based auth.cs* fájlban található egyszerű példa az ACLs alkalmazását mutatja meg.
- A példában egy AuthorizationPolicy nevű osztályt használunk a szabályok beállítására és a Authorize attribútumot a hozzáférés ellenőrzésére.
- Az Authorize attribútumot az AdminController osztályra helyezzük el, hogy biztosítsuk, hogy csak az "Admin" szerepkörrel rendelkező felhasználók férhessenek hozzá az adatokhoz.
- A policy-k konfigurálását a Startup osztályban végezzük el, ahol létrehozunk egy Admin nevű policy-t, amelynek feltétele az AuthorizationPolicy osztály használata.
- Az egyes policy-k ellenőrzését a Authorize attribútum HandleRequirementAsync metódusa végzi el.

# Git verziókezelő

*Készítette: Hugyák Tamás*

*Pannon Egyetem  
Műszaki Informatikai Kar*

*2017.02.17. – v1.1*

# Tartalom

1. Git használata .....	4
1.1. Bevezetés .....	4
1.2. Fogalmak .....	4
1.3. Fájlok állapotai .....	6
1.4. A repository-k fajtái.....	7
1.5. A Git felépítése és egy rövid ismertető.....	7
1.6. Repository-k közötti kommunikáció .....	8
2. A Git-ről részletesebben.....	11
2.1. Branch-ek .....	11
2.2. Konfliktus.....	12
2.3. Remote és local branch-ek.....	13
2.4. Saját adatok beállítása .....	13
2.5. .gitignore fájl .....	13
3. Git parancsok ismertetése .....	15
3.1.1. clone .....	15
3.1.2. status.....	15
3.1.3. add.....	15
3.1.4. commit .....	15
3.1.5. checkout.....	16
3.1.6. fetch .....	16
3.1.7. push.....	16
3.1.8. pull.....	17
3.1.9. revert.....	18
3.1.10. merge .....	18
3.1.11. branch .....	18

3.1.12. diff .....	19
3.1.13. reset .....	19
3.1.14. tag.....	19
3.1.15. stash .....	20
3.1.16. log.....	20
3.1.17. rm .....	20
3.1.18. mv.....	21
3.2. Néhány ábra a Git parancsokról.....	21
3.3. checkout vs reset .....	22

# 1. Git használata

## 1.1. Bevezetés

A Git egy nyílt forráskódú, elosztott verziókezelő szoftver, mely a sebességre helyezi a hangsúlyt. A fejlesztők a saját gépükön nem csak a repository-ban (tárolóban) lévő legfrissebb állapotát tárolják, hanem az egész repot.

A verziókezelői tevékenységek végrehajtása nagyon gyorsan történik, mely a Git erősségét is adja. A központi szerverrel történő hálózati kommunikáció helyett a lokális, saját számítógépen hajtódnak végre a parancsok, így a fejlesztés offline megy végbe a workflow megváltoztatása nélkül. A központi repository-val csak akkor történik kommunikáció, hogyha arra a felhasználó parancsot ad.

Mivel minden egyes fejlesztő lényegében teljes másolattal rendelkezik az egész projektről, ezért a szerver meghibásodásának, a tároló megsérülésének vagy bármilyen bekövetkező adatvesztésnek a kockázata sokkal kisebb, mint a központosított rendszerek által támasztott pont-hozzáférés esetében, hiszen bármely lokális repository-ból visszaállítható a központi szerverre az eredeti tároló állapota.

A Git repository minden egyes példánya – akár local, akár remote – rendelkezik a projekt teljes verziótörténetével, így egy elszigetelt fejlesztői környezetet biztosít minden fejlesztő számára, hogy szabadon kísérletezzenek új funkciók fejlesztésével mindaddig, amíg egy tiszta, publikálható verziót nem képesek előállítani.

## 1.2. Fogalmak

A Git hasonló egy hash-fához, azonban az egyes csomópontokon és leveleken hozzáadott adatokkal rendelkezik.

A Git célja az adott projekt menedzselése, ill. az adatok változtásának nyomon követése. Ezen információk adatstruktúrákban történő tárolását *repository*-nak, röviden *repo*-nak, avagy lokális adatbázisnak nevezik.

A *working directory*, *working copy* vagy *history* az adott projektről, a gyökérkönyvtárról – amelyben a fájlok, forráskódok és mappák találhatóak – egy változatot, verziót, állapotot tartalmaz.

*Snapshot* egy adott pillanatban, időpontban a könyvtárak, fájlok aktuális állapotát, tartalmát, verzióját jelenti. A pillanatkép tulajdonképpen nem más a fájlok esetében, mint egy teljes másolat azok tartalmáról.

*Commit*-olásnak nevezik azt a folyamatot, amely során a Git a *megjelölt* és a *staging area*-ben lévő fájlok ról készült snapshot-okat a lokális adatbázisában (a *.git* könyvtárában) eltárolja, és a tartalmuk alapján egy *SHA-1 hash* kódot (*commit id*) generálva hivatkozik rájuk.

A Git lehetővé teszi, hogy a módosult fájlok közül egy csokorban csak azok kerüljenek eltárolásra az adatbázisában, amelyeket a fejlesztők kiválasztottak. Ezért a *working directory* és a lokális adatbázis közé egy harmadik, *index*, *cache*, *staging area* szinonima nevekkel illetett átmeneti területet alakítottak ki, amelyben információk szerepelnek arról, hogy a következő commit-ban *mely* snapshot-ok legyenek eltárolva. Röviden a *staging area* a fájlrendszernek a következő commit-ra jelölt elemei snapshot-jait tartalmazza.

*Staged*, *cached* jelzővel illetik azokat a fájlokat, amelyek verziókezelve vannak a Git által és a legutolsó commit óta módosítva lettek, illetve az állapotukról, verziójukról, tartalmukról már készült snapshot.

A *working directory*-t *tisztának* (clear) nevezik, ha a fájlokon végzett összes módosítás el van mentve a lokális Git adatbázisba, a repo-ba. Ilyenkor az utolsó commit óta nem történt változtatás az adatokon. A *working directory*-t *piszkosnak* (dirty) nevezik, ha a legutolsó commit óta a verziókezelt fájlokon történtek olyan változtatások, módosítások, amelyek még nem lettek stage-elve, azaz a *staging area*-ba helyezve.

*Branch*-nek nevezik azon commit-ok összességét, melyek egy közös ágra lettek rendezve, és egy közös ős commit-ból erednek lineárisan egymás után fűzve és hivatkozva.

*HEAD* az aktuális lokális branch-ben a legutolsó commit-ra való hivatkozás. Ha a HEAD általt mutatott commit-tól kezdve az egyes commit-ok szülöjén felfelé lépkedünk,

akkor egy elágazásmentes úton haladva a repository legelső commit-jához, a gyökérhez jutunk. Ezt az utat nevezzük branch-nek.

Egy HEAD-et *detached* tulajdonsággal illetik, ha az aktuális commit, amelyre mutat, egyik lokális branch-nek sem a *legutolsó* commit-ja.

A kiadott Git parancsok minden esetben az aktuális branch-re vonatkoznak. Mindig létezik egy aktív, kiválasztott aktuális lokális branch.

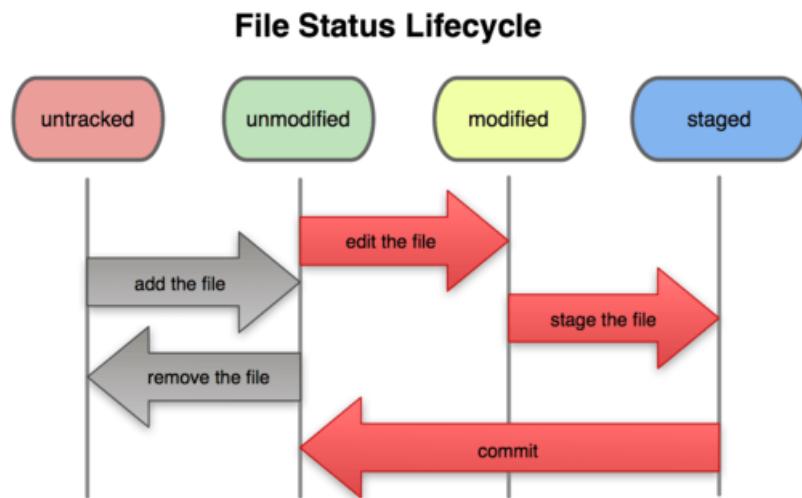
### 1.3. Fájlok állapotai

A Git *untracked* jelzővel illeti azokat a fájlokat és mappákat, amelyekről még egyetlen snapshot sem készült (nem szerepelnek a staging area-ban, ill. nem része a lokális adatbázisnak), tehát nem követi nyomon a rajtuk végzett módosításokat, azaz nincsenek verziókezelve.

*Tracked* megjelöléssel szerepelnek azok az állományok és könyvtárak, amelyek tartalmának változását a Git nyomon követi, verziókezeli.

A *tracked* jelöléssel rendelkező fájloknak 3 további állapotuk lehetséges:

- *modified*: a fájl (tartalma vagy neve, kiterjesztése) módosult a legutóbbi commit óta, de az aktuális állapota még nincs a staging area-ban (nem készült róla snapshot, nincs stage-elve), ezáltal a következő commit-nak nem lesz része.
- *unmodified*: a fájl nem módosult a legutolsó commit óta.
- *staged, cached*: a fájl módosult a legutóbbi commit óta és már snapshot is készült róla (a staging area része, stage-elve lett), a következő commit során el lesz tárolva az adatbázisban.



**1. ábra:** A fájlok lehetséges állapotai.

## 1.4. A repository-k fajtái

Egy adott repo-nak két fajtáját különböztetik meg elhelyezkedés szerint:

- *remote*: a távoli, szerveren lévő központi repo-t jelenti, mely segítségével tartják a kapcsolatot egymással a fejlesztők repo-jai; alapértelmezetten az *origin* névvel hivatkoznak rá
- *local*: a helyi számítógépen lévő, klónozott repo; a tényleges fejlesztés ezen történik

Egy adott repo-nak két fajtáját különböztetik meg hozzáférhetőség szerint:

- *private*: a repository-hoz csak regisztrált és engedélyezett felhasználók férhetnek hozzá
- *public*: a repository-hoz bárki hozzáférhet és használhatja

## 1.5. A Git felépítése és egy rövid ismertető

A Git 3 alapkötöt és részét a working directory, a staging area és a local database (.git könyvtár) jelenti.



### Tracked (and modified)

 If a file was modified since it was last committed, you can stage & commit these changes  
..... **stage** .....

 Changes that were added to the Staging Area will be included in the next commit  
..... **commit** .....

 All changes contained in a commit are saved in the local repository as a new revision

 Changes that are **not staged** will not be committed & remain as local changes until you stage & commit or discard them

### Untracked

 If a file was modified since it was last committed, you can stage & commit these changes

**2. ábra: A Git felépítése.**

Egy új, üres repo létrehozása után a gyökérkönyvtár `.git` nevezetű, rejtett mappájában helyezi el a Git verziókezeléshez szükséges fájljait. Ezután a gyökérkönyvtárban bármilyen adatot elhelyezve a Git érzékelni fogja a változást. Például egy új, üres fájl létrehozásakor a verziókezelő jelzi, hogy egy olyan fájlt talált, amely még nem része az index-nek (Pontosabban nincs verziókezelve), ezáltal úgynevezett *untracked* minősítéssel illeti. Ebben az esetben a rendszer nem követi nyomon a fájlon végrehajtott módosításokat, csak jelzi, hogy nincs indexelve. Ahhoz, hogy a rendszer figyelje a fájlt, s a rajta végzett változtatásokat, hozzá kell adni az index-hez (stage vagy add parancs). Amikor ez megtörténik, *tracked* minősítést kap, és egy pillanatkép (másolat) készül róla, mely az aktuális tartalmát jelöli. Ezután egy üres könyvtár létrehozását is *untracked*-ként fogja megjeleníteni, melyet a cache-hez történő csatolás után *tracked*-re módosít a Git. Az első commit-olás után a fájl és könyvtár aktuális állapota (tartalma) mentésre kerül az adatbázisba és *unmodified* minősítést kapnak.

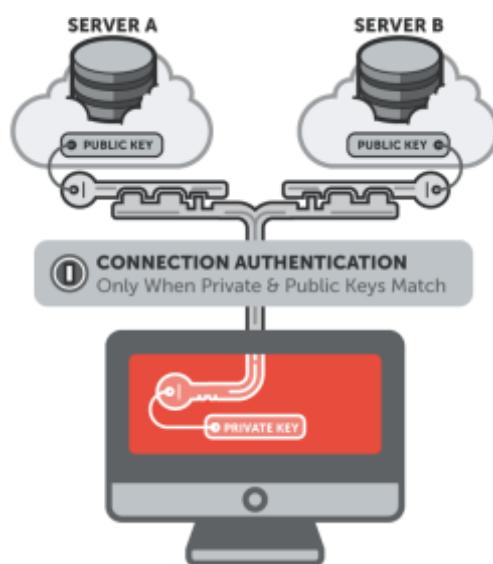
## 1.6. Repository-k közötti kommunikáció

A Git-es parancsok használata során – néhányat kivéve – a verziókezelő rendszer nem hoz létre (interneten keresztül kommunikációs) kapcsolatot a *local* (saját gép) és *remote*

(központi szerver) repo között. Kommunikációra csak akkor van szükség, amikor az adatbázisok szinkronizációja történik. Ez történhet a HTTPS protokollal vagy az SSH nyilvános kulcsai segítségével. Mindkét esetben titkosított csatornán folyik a kommunikáció.

A HTTPS protokollt használva a szolgáltató weboldalán (például Github, Bitbucket, Gitlab) regisztrálni kell egy felhasználónév - jelszó párossal, amelyre a csatlakozás során lesz szükség. (Természetesen ezután egy repository-t szükséges létrehozni a szolgáltató weboldalán.) Ezt követően a repo klónozása alkalmával a szerverhez történő kapcsolódás során a Git kérni fogja a felhasználónevet és a hozzá tartozó jelszót (amelyekkel a szolgáltatónál a regisztráció történt meg), s a sikeres authentikáció után megtörténik a tároló letöltése. Repo klónozása HTTPS protokoll segítségével az alábbi parancs kiadásával valósítható meg a terminálban:

```
git clone https://szolgaltato/felhasznalonev/repo-neve.git
```



3. ábra: Kapcsolódás Git szolgáltatóhoz az SSH protokoll segítségével és a privát-publikus kulcspáros használatával.

SSH protokollt használva először a sajátgépen egy privát-publikus RSA kulcspárost szükséges generálni, mely a kommunikáció alapját szolgálja. Ez a művelet az alábbi parancs kiadásával hajtható végre, amely 4096 bites RSA kulcspárost fog létrehozni:

```
ssh-keygen -t rsa -b 4096
```

A generálás során egy *passphrase*-t kell megadni, amely jelszóként szolgál majd a Git kommunikáció során. A generált kulcsok az operációs rendszer aktuális felhasználója főkönyvtárának a *.ssh* nevezetű rejtett mappájában

találhatóak. A publikus kulcs *.pub* végződésű, míg a privát kulcs nem rendelkezik kiterjesztéssel, s alapértelmezetten minden fájlnak *id\_rsa* a neve. A privát kulcs nem kerülhet másik fél kezébe, ügyelni kell annak biztonságára. Az *id\_rsa.pub* publikus kulcs tartalmát az adott Git szolgáltató oldalán történő bejelentkezés után a *Beállítások*

menüpont alatt az *SSH kulcsok*-at kiválasztva fel kell venni saját kulcsként. (Mindez azért szükséges, hogy a Git kommunikáció során – amely a saját gépemen a privát kulccsal történik meg – a szolgáltató a feltöltött publikus kulcs alapján tudja, hogy ahhoz én rendelkezem hozzáféréssel.) Ezután már az SSH protokoll használatával klónozható is a saját repository-m a szolgáltatótól a következő parancs kiadásával:

```
git clone git@szolgaltato:felhasznalonev/repo-neve.git
```

A csatlakozás során a rendszer kérdi fogja a passphrase-t, melyet a kulcsok generálása során kellett megadni. A helyes jelszó begépelése után a Git letölti a tárolót a remote szerverről az aktuális mappába úgy, hogy az a *repo-neve* nevet kapja. A szolgáltatónál létrehozott összes repository klónozható a feltöltött publikus kulcs segítségével, tehát nincs szükség minden egyes repo-hoz külön kulcspárost létrehozni.

## 2. A Git-ről részletesebben

Jelen fejezetek a Git részletesebb információt közölnek, amelyek nélkülözhetetlenek a használatának elsajátításához.

### 2.1. Branch-ek

A *branch* a fejlesztés egy ágát jelenti, melyet névvel szoktak ellátni. A branch-ek lehetővé teszik, hogy a fejlesztés ne csak egy szálban történjen. Az egyidejű, párhuzamos fejlesztések egymástól elszeparálva működnek. Egy új repo létrehozásakor automatikusan létrejön egy alapértelmezett branch: a *master*.

A projekt history commit-okból épül fel, melyek egymásra hivatkoznak. minden egyes commit-nak legalább egy, legfeljebb kettő szülője van (merge-ölés után). Branch létrehozása során ki kell jelölni egy bázis commit-ot, melyből az új branch származtatása történik. Amint olvasható, a bázis commit-ig minden két branch (az új és a régi, amelyben a bázis commit található) rendelkezik (Pontosabban osztozik) ugyanazon commit-okkal. A branch-ek egymással párhuzamosan léteznek.

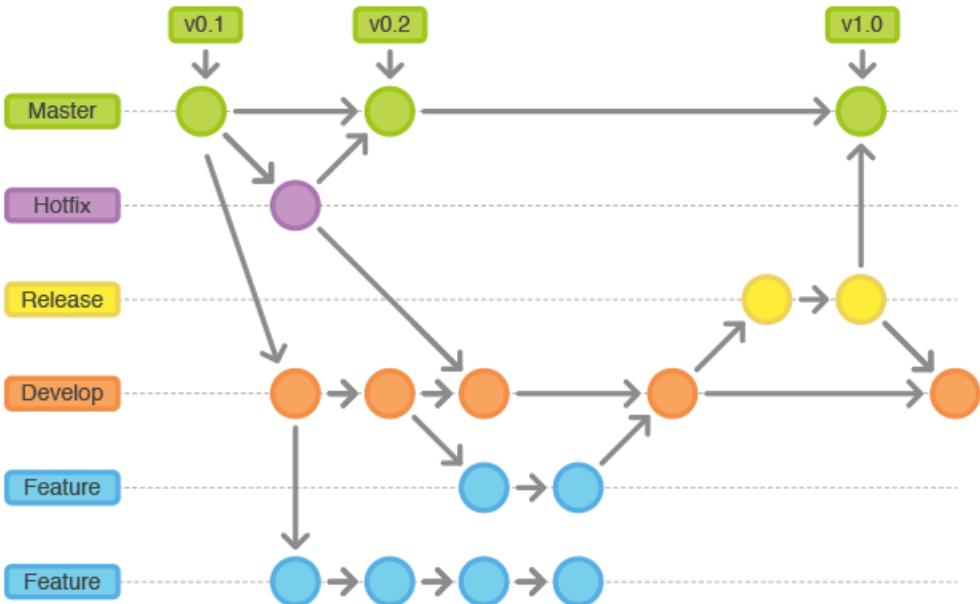
Megjegyzés: A branch-ek egyszerű módon úgy képzelhetők el, mintha a repo-t (tehát a projekt gyökérkönyvtárát) egymás mellé többszörösen lemosolnánk (CTRL + C, majd CTRL + V), s az egyes másolat könyvtárakat a branch nevekkel illetnénk.

A branch tulajdonképpen arra szolgál, hogy a fejlesztést különböző részekre (pl. fejlesztési fázisokra és modulokra) lehessen osztani. Ez azért fontos, mert a fejlesztők egymástól függetlenül dolgozhatnak az egyes ágakon ugyanazon repo-n. Lehetőség van branch-ek közötti váltásra és branch-ek egybefűzésére is. Az előbbi azért fontos, mert átjárhatóságot biztosít a fejlesztés különböző részei között, az utóbbi pedig a fejlesztés párhuzamosan folyó meneteit képes egybeolvasztani, ezáltal kettő vagy több eredményből egy közös keletkezik.

Az egész Git repository-nak a gyökerét az első commit jelenti. Az összes commit és branch ebből a commit-ból, vagy ennek leszármazottjaiból ered.

Minden tároló rendelkezik legalább 1 branch-csel. A fejlesztés az ágakban történik. Branch-ek közötti váltás során a Git visszakeresi a két branch legutolsó közös commit

hivatkozását (bázis vagy ōs branch), s attól a commit-tól kezdve a cél branch összes módosítását leírja a fájlrendszerre.



**4. ábra: Branch-ek kialakítása.**

Az ágak létrehozásakor a *master* branch legtöbbször a *production* fázist jelöli, s ebből származtatják a *release*, *develop* és *hotfixes* ágazatokat. A fő fejlesztés a *develop* ágon megy végbe, melyből modulonként 1-1 *feature* branch-et származtatnak. Az egyes modulok elkészülte után a *develop* ágba fűzik a kész *feature* branch-eket. A *release* branch-be a tesztelés alatt álló *develop* verziók kerülnek. Ha az adott *release* verzió stabil, akkor a végleges *master* ágba kerül befűzésre. A *master* branch-ben felmerült hibákat a *hotfix* nevezetű branch-ekkel javítják, melyeket közvetlenül a *master*-ből származtatnak.

## 2.2. Konfliktus

Konfliktus történik branch-ek merge-ölése vagy pull-olás során, ha két különböző commit egy fájl ugyanazon során történt változtatást tárolja. Ez esetben a Git nem tudja eldönteni, hogy mely módosításokat hagyja meg vagy törölje, ezért értesíti a fejlesztőt, hogy manuálisan javítsa ki a konfliktust.

Például konfliktus történik, ha 2 fejlesztő a master branch-en dolgozik, s egymás után commit-olnak úgy, hogy egy fájl egy adott sorát mind a ketten szerkesztették. Ez esetben az időben később push-oló fejlesztő kap egy figyelmeztetést (*rejected*), hogy a remote

szerveren találhatóak olyan commit-ok az aktuális branch-ben, amelyek még nem lettek letöltve a lokális repoba, ezért még nem push-olhatóak a lokális commit-ok. A fejlesztőnek ez esetben le kell töltenie a változtatásokat (*pull*), s ekkor értesül majd a konfliktusról, hogy a fájl egy sorát már más is módosította. Az adott fájlt szerkeszteni kell, össze kell vágni a helyes kódsort a két commit-ból. Ezután újra kell commit-olni a változtatásokat, s majd már a feltöltés (*push*) művelet sikeresen végrehajtódik.

### 2.3. Remote és local branch-ek

Egy klónozott, sajátgépen lévő repo-ban *remote* és *local* branch-ek találhatóak. A *remote* branch-ek referenciák a *remote* repo-ban lévő branch-ekre. Amolyan „könyvjelzőként” és emlékeztetőül szolgálnak, hogy a *remote* repo branch-eiben minden commit-ok szerepelnek.

A *remote* repository-kban csak a *remote* branch-ek léteznek, *local* branch-ek nem.

A *local* branch-ek a fejlesztést teszik lehetővé, s a commit-ok feltöltése a *remote* tárolóba a *local* branch-ekből történik.

A *remote* branch-ek a *remote* repository-nak (összes branch-ének) az állapotát tárolják, míg a *local* branch-ek a sajátgépen történő fejlesztést tartalmazzák.

### 2.4. Saját adatok beállítása

A Git commit-ok létrehozásakor eltárolja a parancsot kiadó felhasználó nevét, e-mail címét, dátumot és egyéb információkat. A következő parancsokkal a saját név és e-mail cím állíthatóak be:

```
git config --global user.name "Név"  
git config --global user.email "e-mail"
```

### 2.5. .gitignore fájl

A *.gitignore* fájl tartalmazza azon, a repository-ban lévő fájlok és könyvek nevét, amelyeket a Git nem követ nyomon, nem figyeli a változásait, kihagyja a verzió nyomon követésből. A fájlok és mappák neveiben reguláris kifejezések is szerepelhetnek. Általában a projekt gyökérkönyvtárában helyezik el. Példa a tartalmára:

```
# Debug és debug mappák kihagyása
```

```
[Dd]ebug/
# az összes .txt kiterjesztésű fájl kihagyása
*.txt
# obj könyvtár kihagyása
obj/
# config.c fájl kihagyása
config.c
# lib/main.c fájlt kövesse, nem lesz a kihagyott fájlok között
!lib/main.c
```

### 3. Git parancsok ismertetése

A következő fejezetekben a Git parancsainak ismertetése történik.

#### 3.1.1. *clone*

A *clone* egy olyan parancs, mely segítségével egy repository lemasolható a sajátgépre.

```
git clone git@host:felhasznalonev/repo-neve.git repo-mas-neve
```

A Git az aktuális könyvtárban létrehoz egy *repo-mas-neve* nevezetű mappát, majd ebbe a könyvtárba tölti le a tároló adatait, a teljes repo-t. A *repo-mas-neve* paraméter elhagyható, mely esetén a repo nevével megegyező, .git kiterjesztés nélkül hozza létre a könyvtárat az aktuális mappában.

#### 3.1.2. *status*

Az index tartalmának és a working directory állapotának megjelenítésére szolgál. Az *untracked* fájlok, ill. a *modified* és *staged* adatok listázására ad lehetőséget.

```
git status
```

#### 3.1.3. *add*

Az *add* parancs a módosult adatokat az index-be helyezi, snapshot-ot készít róluk. Az összes fájl cache-be történő helyezése rekurzívan:

```
git add .
```

Csak a *config.c* fájl helyezése a cache-be:

```
git add config.c
```

#### 3.1.4. *commit*

A módosítások, snapshot-ok eltárolására szolgál a lokális adatbázisban. minden commit rendelkezik egy rövid szöveggel, leírással, mely arra utal, hogy milyen módosításokat tartalmaz. A commit parancs futtatása csak az add parancs kiadása után lehetséges!

```
git add .
git commit -m "a commit szövege"
```

Az *add* parancs elhagyható az *-a* kapcsolóval csakis akkor, ha az összes módosult fájl *tracked* minősítésű. Egyéb esetben a Git hibát ad:

```
git commit -am "a commit szövege"
```

A *-m* kapcsoló elhagyásával az alapértelmezett szövegszerkesztő ugrik fel, s abban írható meg a commit szövege:

```
git commit
```

### 3.1.5. *checkout*

A *checkout* parancs a branch-ek közötti váltásra szolgál. Ilyenkor a Git az aktuális branch commit-jain visszafelé haladva megkeresi a legelső, cél branch-csel ugyanarra hivatkozó commit-ot, s attól fogva a cél branch commit-jainak változtatásait leírja a fájlrendszerre. A következő példában a develop branch-ra történik váltás:

```
git checkout develop
```

### 3.1.6. *fetch*

A *fetch* parancs segítségével a remote repo-n lévő commit-ok importálásra kerülnek a lokális adatbázisba. A parancs csak a lokális adatbázist frissíti, a working directory-t és a staging area-t érintetlenül hagyja. A letöltött commit-ok a *remote* branch-ekben tárolódnak a *local* branch-ek helyett, ezáltal lehetőség van a szerveren történt módosításokat áttekinteni, mielőtt azok integrálva (*merge*) lesznek a lokális branch-be.

```
git fetch <remote_repo> [<branch_neve>]
```

Például:

```
git fetch origin  
git fetch origin master
```

### 3.1.7. *push*

Az aktuális branch commit-jainak (snapshot-jainak) feltöltése a remote repository egy meghatározott branch-ébe. *Remote* repository-nak nevezzük a távoli szerveren lévő tárolót, mely segítségével tartják egymással a kapcsolatot a fejlesztők. A *remote* szerverre alapértelmezetten *origin* névvel hivatkoznak.

A *fetch* parancs párjaként is emlegetik, mert a *fetch* importálja, a *push* pedig exportálja a commit-okat a *local* és *remote* repository-k között.

Használata:

```
git push -u <remote_repo> <branch_neve>
```

Például: az aktuális branch feltöltése a remote szerver *master* branch-ébe:

```
git push -u origin master
```

A branch nevét azért szükséges megadni, mert előfordulhat, hogy jelenleg a *master* branch az aktuális ág, míg a módosításokat a távoli szerver *production* branch-ébe szeretnénk feltölteni, nem a *master* branch-ébe:

```
git checkout master          # master ágba lépés
git push -u origin production # snapshot-ok feltöltése nem a master
                               # ágba, hanem a production-be
```

### 3.1.8. *pull*

A központi szerverre felküldött változtatások, commit-ok letöltése és merge-ölése az aktuális branch-be. Az adatbázis frissítése (*fetch*) után a working directory-ba, a fájlrendszerre is leírja (*merge*) a módosításokat.

```
git pull [<remote_repo> <branch_neve>]
```

A *<remote\_repo>* és *<branch\_neve>* paraméterek opcionálisak. Például:

```
git pull
git pull origin
git pull origin master
```

A *git pull origin master* parancs a következőt jelenti:

```
git fetch origin
git merge origin/master
```

A *pull* parancs *rebase* móddal is működtethető. Ez esetben az aktuális branch lineáris marad. Az adatbázis frissítése (*fetch*) után a working directory-ba, a fájlrendszerre is leírja (*rebase*) a módosításokat:

```
git pull --rebase [<repository> <branch_neve>]
```

### 3.1.9. *revert*

A *revert* parancs segítségével egy commit-tált snapshot összes módosítása visszavonható. A parancs esetében a Git nem törli a korábbi commit-ot, hanem az aktuális branch-ben létrehoz egy újat, amely visszavonja a megadott commit változtatásait. Erre az integritás megőrzése miatt van szükség.

```
git revert <commit_sha-1_azonosító>
```

### 3.1.10. *merge*

A *merge* parancs lehetővé teszi önálló fejlesztési ágak integrálását egy ágazatba. A parancs használata során annak a branch-nek a nevét kell megadni, amelyet a jelenleg aktív ágba szükséges integrálni. Tehát a *merge* parancssal másik ág integrálása történik az aktuális ágba.

```
git merge <branch_neve>
```

### 3.1.11. *branch*

A *branch* parancs a branch-ek listázására, létrehozására és törlésére szolgál.

A következő parancs egy új ágat hoz létre úgy, hogy a bázisát majd az a commit képezi, amely az aktuális branch aktuális commit-ja:

```
git branch <branch_neve>
```

Branch-ek listázása a következő módokon tehető meg:

```
git branch  
git branch --list --all
```

Branch törlése kétféle módon lehetséges: a *-d* kapcsolóval biztonságosan törölhető, mert a Git jelez, ha olyan commit-okat tartalmaz, amelyek egyetlen másik ágba sem lettek merge-ölve.

```
git branch -d <branch_neve>
```

A *-D* kacsolóval oly módon törölhető egy branch, hogy lehetnek benne olyan commit-ök, amelyek egyetlen egy másik ágazatnak sem része.

```
git branch -D <branch_neve>
```

Az aktuális branch átnevezése:

```
git branch -m <új_branch_nev>
```

### 3.1.12. diff

A *diff* parancs a változtatásokat mutatja meg a working directory és az index, vagy két commit, branch, esetleg fájl között.

```
git diff  
git diff --cached  
git diff <branch1> <branch2>
```

### 3.1.13. reset

A *reset* parancs használható comit-tált snapshot-ok törlésére, ill. a staging area és az index változtatásainak visszavonására. Mindkét esetben csak lokális változtatások visszaállítására alkalmas. A *remote* repo-ra kiküldött snapshot-okra nem alkalmazható!

```
git reset <commit_sha-1_azonosito>
```

### 3.1.14. tag

A Git lehetőséget ad a history fontosabb pontjainak megcímkezésére. Ezt a leggyakrabban a verziószámok megjelölésére használják.

A következő parancs 'v1.4'-es címkével látja el az aktuális commit-ot, és 'my version 1.4' rövid leírással illeti azt.

```
git tag -a v1.4 -m 'my version 1.4'
```

Címkék listázása:

```
git tag -l
```

Adott címke részletes információi:

```
git show <tag_neve>  
git show v1.4
```

Csak a címkék felküldése a *remote* repo-ra:

```
git push <remote> --tags
```

### 3.1.15. *stash*

A *stash* parancs biztosítja a working directory változtatásainak ideiglenes elmentését egy biztonságos helyre. Erre akkor lehet szükség, amikor pl. branch-váltás történik. Egyik ágról a másikra csak úgy lehet átváltani, hogy ha a working directory „tiszta”, tehát vagy nem történtek változtatások, vagy a változtatások már commit-elve lettek. A másik branch-re történő átállás előtt először a módosításokat stash-elní kell – ha a fejlesztő nem szándékozik még commit-olni –, majd ezután engedélyezett a *checkout* parancs futtatása. Stash-elés nélkül a Git hibát ad a folyamatra.

A még nem snapshot-olt módosítások ideiglenes eltárolása:

```
git stash
```

A még nem snapshot-olt módosítások ideiglenes eltárolása ’message’ néven:

```
git stash save "message"
```

Ideiglenesen eltárolt változtatások listázása:

```
git stash list
```

A *stash@{1}* módosítások visszaállítása a working directory-ba:

```
git stash apply stash@{1}
```

A legutolsó mentett stash állapot visszaállítása:

```
git stash pop
```

### 3.1.16. *log*

A commit logokról és a branch-ekről ad információt.

```
git log  
git log --oneline --graph --decorate  
git log --graph --parents --name-status --oneline
```

### 3.1.17. *rm*

Fájlok törlésére szolgál a working directory-ból és az index-ból. A következő parancs kiadása után minden a fájlrendszerrel, mint az index-ból eltávolításra kerül a fájl:

```
git rm <fajl_nev>
```

Például:

```
git rm readme.txt
```

A *--cached* kapcsolóval csak az index-ből törlődik a fájl, a fájlrendszerrel (working directory) viszont nem:

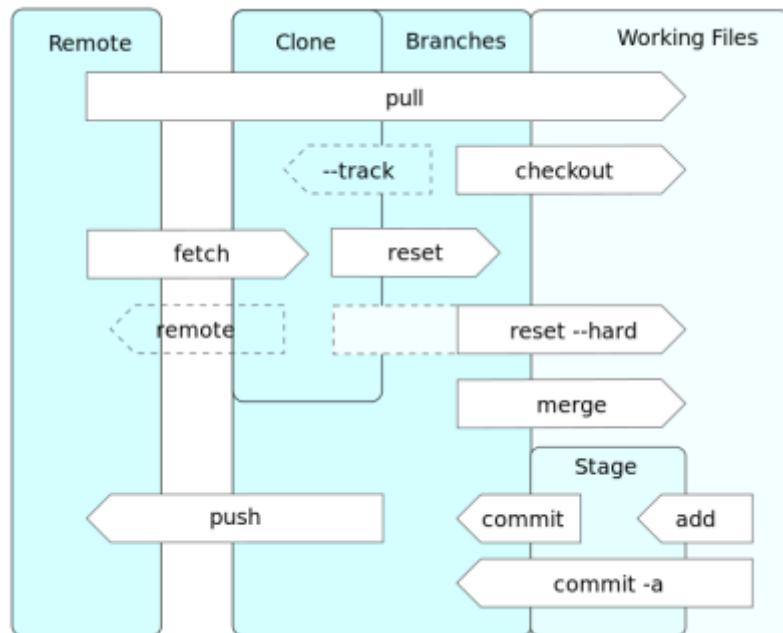
```
git rm --cached readme.txt
```

### 3.1.18. mv

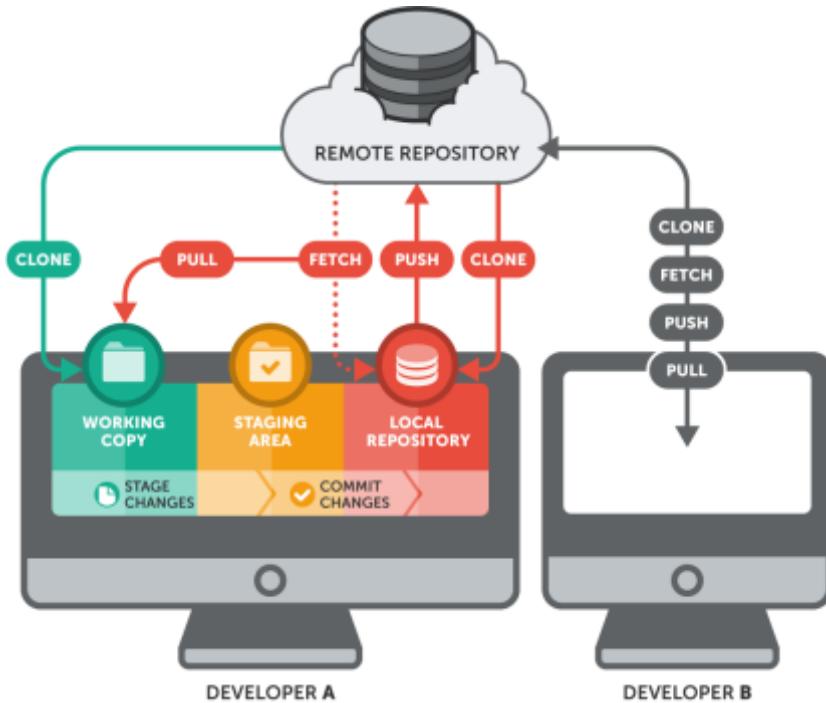
Fájl, könyvtár átnevezésére vagy áthelyezésére szolgál.

```
git mv <forrás> <cél>
```

## 3.2. Néhány ábra a Git parancsokról



5. ábra: Parancsok és szekciók.

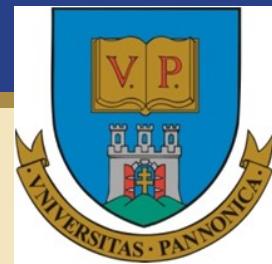


6. ábra: A Git workflow.

### 3.3. checkout vs reset

A *checkout* és *reset* parancsok segítségével korábbi commit-ok állapotára lehetséges visszaállítani a working directory-t. A két parancs használata abban különbözik egymástól, hogy *checkout* alkalmazása esetén az adott commit-ra egy új, ideiglenes branch jön létre (nem változtatja meg az aktuális branch-et), melynek a nevét a commit SHA-1 azonosítója adja. *Reset* használata során az aktuális branch *HEAD*-jét állítja át a Git, ezáltal a branch commit-jai módosulnak.

# A rendszerfejlesztés korszerű módszerei



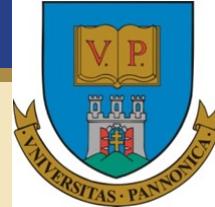
## Csatlakozás adatbázishoz ORM

---

Pannon Egyetem, Műszaki Informatikai Kar

2022

# Bevezetés



- Az adatbázishoz való hozzáférés fontos szempont minden programozási nyelvben.
- minden szoftver nélkülözhetetlen része, hogy elérje adatait.
- Fontos szempont, hogy különböző típusú adatbázisokat lehessen elérni egy szoftverből.
- A különböző keretrendszerek ehhez felületet biztosítanak.
- Példakódok
  - C#: <https://www.codeproject.com/Articles/823854/How-to-Connect-SQL-Database-to-your-Csharp-Program>
  - Java: <https://www.codejava.net/java-se/jdbc/connect-to-microsoft-sql-server-via-jdbc>

# Adatbázis elérése .NET keretrendszerben



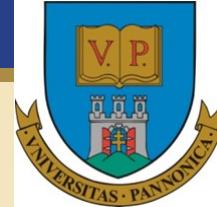
## ■ Kapcsolódás az adatbázishoz

```
using (SqlConnection conn = new SqlConnection())
{
    //Windows authentication
    conn.ConnectionString = "Server=(local);Database=EventAppDb;Trusted_Connection=true";
    //Sql server authentication
    //conn.ConnectionString = "Server=(local);Database=EventAppDb;User ID=sa;Password=PAssword123";
    conn.Open();
    return conn;
}
```

## ■ ConnectionString:

- Server: adatbázis szerver elérhetősége
- Database: adatbázis neve
- Trusted\_Connection=true: windows authentikáció
- User ID + Password: Sql szerver authentikáció

# Adatbázis elérése .NET keretrendszerben



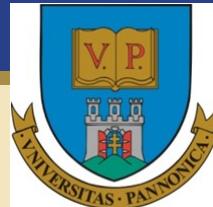
## ■ Adat lekérése

```
// Create the command
SqlCommand command = new SqlCommand("SELECT * FROM People WHERE Id = @0", conn);
// Add the parameters.
command.Parameters.Add(new SqlParameter("@0", 1));

/* Get the rows and display on the screen!
 * This section of the code has the basic code
 * that will display the content from the Database Table
 * on the screen using an SqlDataReader. */

using (SqlDataReader reader = command.ExecuteReader())
{
    Console.WriteLine("Id\tName\t\tDate of Birth\t");
    while (reader.Read())
    {
        Console.WriteLine(String.Format("{0} \t | {1} \t | {2}",
            reader[0], reader[1], reader[2]));
    }
}
```

# Adatbázis elérése .NET keretrendszerben



## ■ Adat beszúrása

```
// Create the command, to insert the data into the Table!
// this is a simple INSERT INTO command!

SqlCommand insertCommand = new SqlCommand("INSERT INTO People (Name, DateOfBirth) VALUES (@0, @1)", conn);

// In the command, there are some parameters denoted by @, you can
// change their value on a condition, in my code they're hardcoded.

insertCommand.Parameters.AddWithValue("@0", "Test Person2");
insertCommand.Parameters.AddWithValue("@1", new DateTime(1976,4,6));

// Execute the command, and print the values of the columns affected through
// the command executed.

Console.WriteLine("Commands executed! Total rows affected are " + insertCommand.ExecuteNonQuery());
```

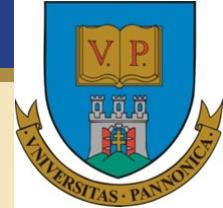
# Adatbázis elérése .NET keretrendszerben



## ■ Kapcsolat bezárása

```
// Close the connection  
conn.Close();
```

# ORM



- Objektum-relációs leképezés
- Programozási technika adatok konvertálására nem kompatibilis típusos rendszerek és objektumorientált programozási nyelvek között.
- Az objektumorientált programozásban a műveleteket általában az objektumok módosításával végezzük, amelyek szinte soha nem skalár változók.
- A legtöbb népszerű adatbázis, mint az SQL DBMS, csak skalár értékeket (pl. stringeket, egészeket) tud tárolni és kezelní.

# ORM



- A probléma lelke az objektum logikai reprezentációjának átalakítása atomi formára, amely alkalmas az adatbázisban való tárolásra.
- A cél az, hogy az objektumok tulajdonságait és kapcsolatait megőrizzük, így helyesen vissza tudjuk tölteni az adatbázisból amikor szükséges.
- ORM eszközök
  - C#: Entity Framework, Entity Framework Core
  - Java: Hibernate
  - PHP: Doctrine, RedBean
  - Stb.

# ORM előnyei és hátrányai



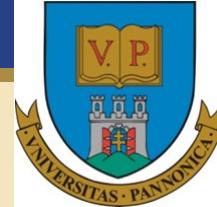
## ■ Előnye

- Gyakran lecsökkenti a megírandó kód mennyiségét, objektumorientált nyelv és relációs adatbázis közötti adatcsere esetén.
- A tárolt eljárások esetleg jobban teljesítenek, de nem hordozhatóak.

## ■ Hátránya

- Hátrányos azokon a területeken, amelyeken az adatbázis specifikus technikák jelentősen optimalizáltak.
- A legtöbb ORM eszköz nem teljesít jól nagy mennyiségű adat törlése esetén, vagy adattáblák összekapcsolásában.
- Az ORM eszközök használata gyakran vezethet rosszul tervezett adatbázisok készítéséhez. (ORM esetén tekintsünk el a normálformáktól.)

# Kitérő: nem SQL adatbázisok



- Egy másik megoldás az objektumorientált adatbázis (OODBMS - Object-Oriented Database Management System) vagy dokumentumorientált adatbázis (pl: natív XML adatbázis) használata.
- OODBMS rendszerek olyan adatbázisok, amelyek objektumorientált változók tárolására alkalmasak.
- OODBMS rendszer használata esetén nincs szükség az objektumok konvertálására SQL formába, mivel az objektumok közvetlenül tárolhatóak az adatbázisban.
- Pl: MongoDB, ArangoDB, Objectivity/DB, ZopeDB stb.

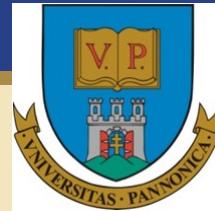
# Kitérő: nem SQL adatbázisok



## ■ Egy objektum MongoDB-ben

```
_id: ObjectId("5addee237d44043570cbb44ef")
__v: 11
config_id: "GTK"
deleted: false
description: "leírás"
email: "info@gtk.uni-pannon.hu"
name: "Gazdaságtudományi Kar"
> news: Array
  picture: "http://www	gtk.uni-pannon.hu/wp-content/uploads/2014/12/GTK_header_hu...."
✓ programmes: Array
  0: ObjectId("5adeedfc44043570cbb44f5")
  1: ObjectId("5adeeee06d44043570cbb44f6")
  2: ObjectId("5adeeee36d44043570cbb44f7")
  3: ObjectId("5aeaad36e5166e45bd859fd0a")
  4: ObjectId("5aeaad3bb5166e45bd859fd0b")
  5: ObjectId("5aeaad4485166e45bd859fd0c")
  6: ObjectId("5aeaad47c5166e45bd859fd0d")
short_name: "gtk"
```

# .NET Core ORM: EF Core



- <https://docs.microsoft.com/en-us/ef/core/get-started/?tabs=netcore-cli>
- <https://www.entityframeworktutorial.net/efcore/entity-framework-core.aspx>
- Entity Framework (EF) Core is a lightweight, extensible, open source and cross-platform version of the popular Entity Framework data access technology.
- EF Core can serve as an object-relational mapper (O/RM), enabling .NET developers to work with a database using .NET objects, and eliminating the need for most of the data-access code they usually need to write.
- EF Core supports many database engines (ex.: Microsoft.EntityFrameworkCore.SqlServer)

# EF Core: the model



- With EF Core, data access is performed using a model.
- A model is made up of entity classes and a context object that represents a session with the database, allowing you to query and save data.
- You can generate a model from an existing database, hand code a model to match your database, or use EF Migrations to create a database from your model, and then evolve it as your model changes over time.

# EF Core: the model



```
public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }
    public int Rating { get; set; }
    public List<Post> Posts { get; set; }
}

public class Post
{
    public int PostId { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }

    public int BlogId { get; set; }
    public Blog Blog { get; set; }
}
```

# EF Core: the model

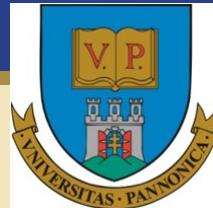


```
public class BloggingContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }
    public DbSet<Post> Posts { get; set; }

    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder.UseSqlServer(
            @"Server=(localdb)\mssqllocaldb;Database=Blogging;Integrated Security=True");
    }
}
```

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<BloggingContext>(options =>
        options.UseSqlServer(Configuration.GetConnectionString("BloggingDatabase")));
}
```

# EF Core: querying



- Instances of your entity classes are retrieved from the database using Language Integrated Query (LINQ).

```
using (var db = new BloggingContext())
{
    var blogs = db.Blogs
        .Where(b => b.Rating > 3)
        .OrderBy(b => b.Url)
        .ToList();
}
```

# EF Core: saving data



- Data is created, deleted, and modified in the database using instances of your entity classes.

```
using (var db = new BloggingContext())
{
    var blog = new Blog { Url = "http://sample.com" };
    db.Blogs.Add(blog);
    db.SaveChanges();
}
```

# Migrations



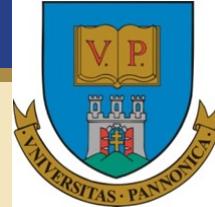
- A data model changes during development and gets out of sync with the database.
- You can drop the database and let EF create a new one that matches the model, but this procedure results in the loss of data.
- The migrations feature in EF Core provides a way to incrementally update the database schema to keep it in sync with the application's data model while preserving existing data in the database.

# Migrations: create



- After you've defined your model, it's time to create/update the database.
- Three files are added to your project under the Migrations directory:
  - XXXXXXXXXXXXXXXX\_MigrationName.cs
    - The main migrations file. Contains the operations necessary to apply the migration (in Up()) and to revert it (in Down()).
  - XXXXXXXXXXXXXXXX\_MigrationName.Designer.cs
    - The migrations metadata file. Contains information used by EF.
  - MyContextModelSnapshot.cs
    - A snapshot of your current model. Used to determine what changed when adding the next migration.
- The timestamp in the filename helps keep them ordered chronologically so you can see the progression of changes.

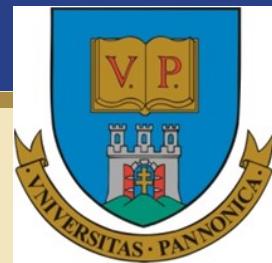
# Migrations



- Update the database: apply the migration to the database to create/update the schema
- Remove migration: sometimes you add a migration and realize you need to make additional changes to your EF Core model before applying it, with this command you can remove the last migration
- Revert a migration: if you already applied a migration (or several migrations) to the database but need to revert it, you can use the same command to apply migrations, but specify the name of the migration you want to roll back to

# A rendszerfejlesztés korszerű módszerei

WebSocket



---

Pannon Egyetem, Műszaki Informatikai Kar

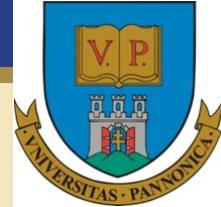
2023

# Bevezetés



- A WebSocket egy protokoll, amely lehetővé teszi a valós idejű kétirányú kommunikációt egy webalkalmazás és egy szerver között.
  - A hagyományos HTTP kérés-válasz modellel szemben a WebSocket lehetővé teszi a szerver számára, hogy elküldje az adatokat a kliensnek bármikor anélkül, hogy a kliensnek először kérnie kellene az azokat.

# Kapcsolat felépítése



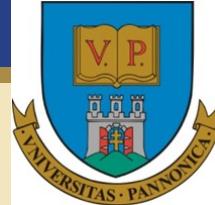
- A kliens HTTP GET kérést küld a szervernek, amely tartalmazza a WebSocket protokoll elérési útját.
  - Ez a GET kérés alapvetően olyan, mint a normál weboldalak betöltésekor használt GET kérés.
- Amikor a szerver megkapja a GET kérést, akkor válaszként küld egy HTTP választ a kliensnek, amely tartalmazza a válasz fejlécet és a 101-es állapotkódot (Switching Protocols), amely azt jelzi, hogy a szerver elfogadta a kliens WebSocket kérését.

# Kapcsolat felépítése



- A kliens és a szerver közötti kapcsolat ezután egy WebSocket kapcsolatba megy át, amely lehetővé teszi a két fél közötti valós idejű adatátvitelt.
  - A kapcsolat SSL/TLS használatával is biztosítható.
- A kapcsolat létrejötte után a kliens és a szerver adatokat küldhet egymásnak a WebSocket protokoll segítségével.
  - Mindkét fél lehetőséget kap arra, hogy adatot küldjön és fogadjon, és minden fél képes lesz az adatokat egyszerre küldeni és fogadni.

# WebSocket előnyei



## ■ Valós idejű kommunikáció

- A WebSocket lehetővé teszi az adatok valós idejű továbbítását, ami különösen fontos az olyan alkalmazásokban, amelyek dinamikusan változnak vagy gyorsan frissülnek.

## ■ Kisebb sávszélesség

- A WebSocket protokoll kisebb sávszélességet használ, mint a HTTP, mivel az adatokat csak akkor küldi, amikor azok ténylegesen szükségesek.

## ■ Egyszerűbb kommunikáció

- A WebSocket egyszerűbb kommunikációt biztosít a kliens és a szerver között, és kevésbé terheli a szerver erőforrásait, mivel nincs szükség felesleges HTTP kérés-válasz ciklusokra.

## ■ A WebSocket használata segít az alkalmazások gyorsabbá és hatékonyabbá tételeben, és lehetővé teszi az alkalmazások számára, hogy jobban alkalmazkodjanak az ügyfelek igényeihez.

# Websocket hátrányai



## ■ Korlátozott támogatottság

- Bár a WebSocket protokoll egyre elterjedtebb, még mindig vannak olyan böngészők, amelyek nem támogatják a protokollt. Az ilyen böngészőkben az alkalmazásnak alternatív kommunikációs csatornákat kell használnia.

## ■ Biztonsági problémák

- A WebSocket protokoll használata növeli a biztonsági kockázatot, ha nem megfelelően konfigurálják.

## ■ Túlzott terhelés

- A WebSocket protokoll használata növelheti a szerver terhelését, ha a szerver nem megfelelően van konfigurálva, és nem képes kezelni a nagy mennyiségű adatforgalmat.

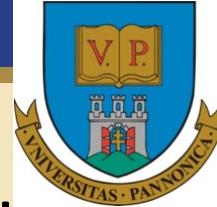
## ■ Többlet kód

- A WebSocket protokoll használata növeli a kód bonyolultságát azáltal, hogy a kódba be kell illeszteni a WebSocket kezelőlogikáját. Ez további tesztelést és karbantartást igényelhet.

## ■ Korlátozott hibakeresési lehetőségek

- A WebSocket protokoll használatakor a hibakeresés és a hibajavítás nehezebb lehet, mivel a hibák azonnal jelentkeznek, és nem lehet lépésről-lépésre követni az adatforgalmat, mint az HTTP kérés-válasz esetén.

# Websocket példa (.NET)



- Telepítenünk kell a Microsoft.AspNetCore.WebSockets csomagot a NuGet csomagkezelő segítségével.
- Hozzunk létre egy WebSocketHandler osztályt, amely a beérkező üzenetek kezelésére szolgál.
- Készítsünk egy WebSocketMiddleware osztályt, amely a WebSocket kapcsolatok kezelésére szolgál.
- Készítsünk egy WebSocketManager osztályt, amely a WebSocket-k nyilvántartására szolgál.
- Végül készítsünk egy példa alkalmazást, amely a WebSocket szerverünket használja.

# Websocket példa (.NET)



- A kód használja a Microsoft.AspNetCore.WebSockets csomagot, amely lehetővé teszi a WebSocket kapcsolatok kezelését a .NET Core platformon.
- A WebSocketHandler absztrakt osztály az üzenetek fogadásáért, a kapcsolatok kezeléséért és az üzenetek továbbításáért felelős.
- A WebSocketMiddleware osztály a WebSocketHandler osztály használatával hajtja végre a WebSocket kommunikációt.
- A WebSocketManager osztály az összes aktív WebSocket kapcsolat nyilvántartását végzi.
- A ChatHandler osztály a WebSocketHandler absztrakt osztály leszármazottja, és implementálja az üzenetek fogadásáért, a kapcsolatok kezeléséért és az üzenetek továbbításáért felelős metódusokat.
- A Startup osztály konfigurálja az alkalmazást, és engedélyezi a WebSocket kommunikációt a UseWebSockets metódus használatával.

# WebSocket példa (.NET)



## ■ Az alkalmazás működése a következő:

- Az alkalmazás engedélyezi a WebSocket kommunikációt a UseWebSockets metódus használatával.
- Az alkalmazás Startup osztálya létrehozza a ChatHandler objektumot, amely a WebSocket kommunikáció kezeléséért felelős.
- Amikor egy kliens csatlakozik, az alkalmazás WebSocketMiddleware osztálya létrehoz egy WebSocketHandler objektumot, és hozzáadja a kapcsolatot a WebSocketManager osztályhoz.
- Amikor egy üzenet érkezik, az alkalmazás ChatHandler osztálya kezeli az üzenetet, és továbbítja az összes csatlakoztatott kliensnek.
- Amikor egy kliens kapcsolatot bont, az alkalmazás WebSocketMiddleware osztálya eltávolítja a kapcsolatot a WebSocketManager osztályból.

# Mi történik az alkalmazásszerver indításakor?



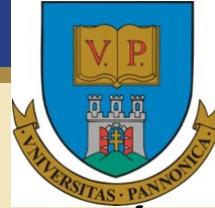
- Amikor a szerver elindul, létrehoz egy WebSocket szerver alkalmazást, amely figyeli a bejövő kapcsolatokat.
- Amikor egy kliens csatlakozik, a szerver létrehoz egy WebSocket csatornát, amelyen keresztül a kliens és a szerver kommunikálhat egymással.
- A szerver külön szálakat használhat a WebSocket kérések kezeléséhez, így több kapcsolat kezelése is lehetséges.
  - A szerveren futó alkalmazásnak nyitva kell tartania a kapcsolatokat, és figyelni a bejövő adatokat.
- Amikor a szerver adatot kap a kliensektől, azt feldolgozza, majd elküldi az adatot az összes kapcsolatban lévő kliensnek.
  - A WebSocket segítségével a szerver és a kliens között valós idejű kommunikáció lehetséges, és az adatok átvitele gyors és hatékony.

# Mi történik a kliens oldalon?



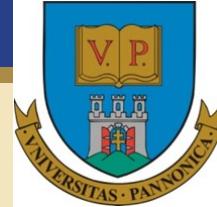
- A kliensoldali WebSockets alkalmazások általában JavaScript nyelven íródnak, és a HTML5 API részét képezik.
- Amikor a kliensoldali alkalmazás futtatása elindul, a böngészőben futó JavaScript kód kérésben küldi a szerver felé a WebSocket kapcsolat kezdeményezését.
- Amikor a szerver engedélyezi a WebSocket kapcsolatot, a kliensoldali JavaScript alkalmazás megnyit egy WebSocket objektumot, amelyen keresztül a kommunikáció történik.
  - A kliensoldali alkalmazás a WebSocket objektum segítségével küldhet adatokat a szervernek, és fogadhat adatokat a szerverről.

# Mi történik a kliens oldalon?



- A kliensoldali alkalmazás figyeli az érkező adatokat, és amikor azok megérkeznek, az alkalmazás feldolgozza az adatokat, majd a logikájának megfelelően válaszol a szervernek (vagy nem).
- Amikor a kapcsolatot le szeretné zárni a kliens, az alkalmazás bezárja a WebSocket objektumot. Ekkor a szerver értesül arról, hogy a kapcsolat megszűnt.
- A kliensoldali alkalmazások használatakor fontos figyelembe venni a WebSockets által felmerülő biztonsági kockázatokat, és megfelelően védeni az alkalmazást a rosszindulatú felhasználóktól.

# Mi történik, ha megszakad a kapcsolat?



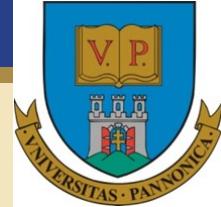
- Amikor megszakad a kapcsolat, akár a szerver, akár a kliens oldalán, akkor a WebSocket objektum bezáródik, és az alkalmazás értesül arról, hogy a kapcsolat megszűnt.
- A WebSocket API támogatja a kapcsolat megszakadásának két fő típusát:
  - az egyik, amikor a kapcsolat hirtelen megszakad (pl. a szerver leállása, a hálózati hiba stb. miatt),
  - a másik pedig, amikor a kapcsolatot a kliens vagy a szerver oldalán szándékosan zárják be.

# Mi történik, ha megszakad a kapcsolat?



- Amikor a kapcsolat hirtelen megszakad, a WebSocket objektum azonnal bezárul.
  - Ebben az esetben az alkalmazásnak észre kell vennie a kapcsolatvesztést, és újra kell kezdenie a kapcsolatfelvételt, ha továbbra is szeretne kommunikálni a szerverrel.
- Ha a kapcsolatot szándékosan zárják be, akkor a szerver vagy a kliens oldalán az alkalmazás hív egy "close" metódust a WebSocket objektumon, hogy jelezze a szándékos kapcsolatbontást.
  - Ebben az esetben a kliensnek szabadon kell kezelnie a kapcsolat bontását, és elvégeznie a szükséges utómunkálatokat (pl. adatok mentése, a felhasználói felület frissítése stb.).
  - A szervernek is értesülnie kell arról, hogy a kapcsolat megszűnt, és gondoskodnia kell arról, hogy az erőforrásokat felszabadítsa, amelyeket a kapcsolat kezeléséhez használt.

# A websocket alkalmazásai



- Chat alkalmazások
  - A WebSocket segítségével lehetőség van valós időben kommunikálni a felhasználókkal.
- Online játékok
  - A WebSocket protokoll lehetővé teszi a valós idejű kommunikációt a játékosok között.
- Online kereskedési platformok
  - A WebSocket lehetővé teszi a valós idejű adatok átvitelét a tőzsdei adatokról.
- Online sportközvetítések
  - A WebSocket protokoll lehetővé teszi a valós idejű sportesemények követését, az eredmények frissítését és a felhasználói élmény javítását.
- IoT (Internet of Things) alkalmazások
  - A WebSocket protokoll lehetővé teszi az IoT eszközökkel való valós idejű kommunikációt és azok felügyeletét.
- Videokonferencia alkalmazások
  - A WebSocket protokoll lehetővé teszi a valós idejű video- és hangátvitelt, ami alapvető fontosságú a videokonferencia alkalmazások számára.
- A protokoll rendkívül sokoldalú, és lehetővé teszi a valós idejű kommunikációt a böngésző és a szerver között, amelyet számos egyéb területen is alkalmaznak.

# A rendszerfejlesztés korszerű módszerei

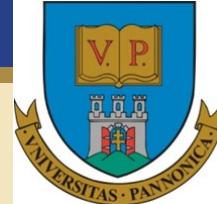
## Message Queue



Pannon Egyetem, Műszaki Informatikai Kar

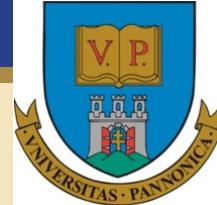
2023

# Bevezetés



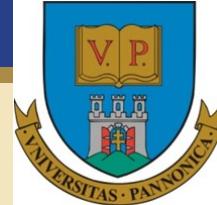
- A message queue egy olyan kommunikációs mechanizmus, amely lehetővé teszi a különböző folyamatok vagy szolgáltatások közötti üzenetek továbbítását és fogadását.
- A message queue egy sorban tárolja az üzeneteket, amelyeket az egyik folyamat küld a másiknak.
  - A küldő folyamat elhelyezi az üzenetet a sor végére, majd a fogadó folyamat az üzenetet kiolvassa a sor elejéről.
  - Az üzeneteket általában aszinkron módon küldik és fogadják, ami azt jelenti, hogy a küldő és a fogadó folyamatok nem kell, hogy ugyanabban az időben fussanak.

# Bevezetés



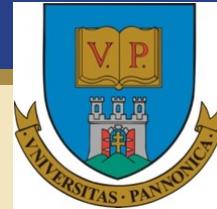
- A message queue fontos jellemzője, hogy a küldő és fogadó folyamatok közötti kapcsolat lazább, mint más kommunikációs mechanizmusoknál, például a szinkronizált csatornáknál vagy a közös memórián alapuló megosztott adatoknál.
- A message queue használatával a folyamatok függetlenek maradhatnak egymástól, és az üzeneteket továbbíthatják, amikor készen állnak arra.

# Üzenetek küldése és fogadása



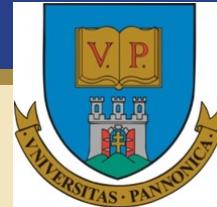
- A küldő folyamat létrehoz egy üzenetet, és elhelyezi azt a message queue-ban.
- A message queue elmenti az üzenetet a sorban, és visszaigazolja a küldő folyamatnak, hogy az üzenet sikeresen bekerült a sorba.
- A fogadó folyamat folyamatosan figyeli a message queue-t, hogy megkapja az új üzeneteket.
- Amikor az üzenet sorra kerül, a message queue elküldi azt a fogadó folyamatnak.
- A fogadó folyamat fogadja az üzenetet, és feldolgozza a benne található információkat.
- A fogadó folyamat visszaigazolja a message queue-nak, hogy az üzenetet sikeresen fogadta.
- A message queue eltávolítja az üzenetet a sorból.

# Üzenetek küldése és fogadása



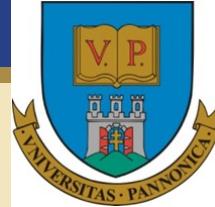
- Ez a folyamat általában aszinkron módon történik, amely azt jelenti, hogy a küldő és fogadó folyamatok függetlenek egymástól, és nem kell, hogy ugyanabban az időben fussanak.
- Emellett a message queue biztosítja a folyamatok közötti biztonságos és megbízható kommunikációt, mivel az üzeneteket megbízhatóan kézbesíti, és a küldő és fogadó folyamatok csak akkor kapcsolódnak egymáshoz, amikor szükség van rá.

# Különböző típusú üzenetek kezelése



- Amikor egy alkalmazásból többféle üzenetet szeretnénk küldeni másikba, akkor általában az üzeneteket különböző típusokra osztjuk fel, és minden típushoz külön message queue-t használunk.
  - Például, ha egy rendszerben két különböző alkalmazás van, és az egyik alkalmazásnak két különböző típusú üzenetet kell küldenie a másiknak, akkor két külön message queue-t hozunk létre, amelyeket az egyik alkalmazás használ a két különböző típusú üzenet elküldésére.

# Különböző típusú üzenetek kezelése



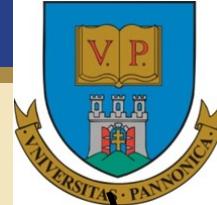
- Az üzenetek típusát általában az üzenet tartalma határozza meg.
  - Például, ha az egyik alkalmazás üzenetet küld a másiknak a felhasználói adatok frissítéséről, akkor az üzenet típusa lehet "UserUpdate", míg ha egy másik alkalmazás üzenetet küld a másiknak egy új rendelésről, akkor az üzenet típusa lehet "NewOrder".
- Az üzenet típusokat gyakran olyan adatszerkezetekkel definiálják, mint például az XML vagy a JSON, amelyek lehetővé teszik az üzenetek konzisztens és strukturált kezelését.

# Különböző típusú üzenetek kezelése



- Az üzenet típusok elkülönítése több előnnyel is jár.
  - Például könnyűvé teszi az üzenetek kezelését a fogadó alkalmazás számára, mivel a fogadó alkalmazás pontosan tudja, hogy milyen típusú üzeneteket kell kezelnie, és hogyan kell azokat feldolgozna.
- Emellett segít az alkalmazások közötti kommunikáció strukturáltabbá tételeben, és lehetővé teszi az alkalmazások számára, hogy különböző fajta üzeneteket küldjenek egymásnak anélkül, hogy azok összekeverednének vagy összejavarának a fogadó alkalmazást.

# Topic



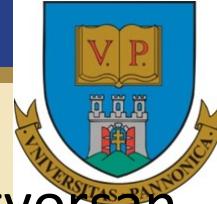
- A topic egy olyan fogalom, amely gyakran kapcsolódik az üzenetsorokhoz (message queue) és a publikálás-feliratkozás (publish-subscribe) kommunikációs mintához.
- A topic (magyarul témakör) egy szabadon választható azonosító, amely egy adott témakört vagy témaköröket jelöl.
- A topic alapú kommunikációban a feladók egy vagy több témakört határoznak meg, amelyekhez az üzeneteket küldik. A fogadók pedig a témakörök alapján iratkoznak fel, így csak azokat az üzeneteket kapják meg, amelyek érdeklik őket.

# Topic



- Például, egy valós idejű adatokat feldolgozó rendszerben az adatszolgáltatók adataikat különböző téma-körökre küldik, például "stock\_prices", "weather\_data" vagy "traffic\_updates".
- Az adatfogyasztók pedig feliratkoznak a különböző téma-körökre annak érdekében, hogy csak azokat az adatokat kapják meg, amelyeket valóban szükségük van.
- A topic alapú kommunikáció használata lehetővé teszi az alkalmazások számára, hogy hatékonyan kommunikáljanak egymással, miközben minimalizálják a fölösleges adatforgalmat és az üzenetek felesleges kezelését.

# Message queue előnyei



## ■ A rendszer leterheltségének kezelése

- A message queue segítségével az alkalmazások képesek gyorsan küldeni és fogadni üzeneteket, ami lehetővé teszi a rendszer számára, hogy hatékonyan kezelje a terhelést és az időbeli igényeket.
- Az üzenetek átmeneti tárolása és továbbítása lehetővé teszi az alkalmazások számára, hogy a saját tempójukban dolgozzanak, miközben a rendszer dinamikusan reagál a változó igényekre.

## ■ Rugalmasság

- A message queue segítségével az alkalmazások egymással kommunikálhatnak anélkül, hogy szinkronizálniuk kellene a folyamataikat vagy ismerniük kellene egymás pontos működési mechanizmusait.
- Az üzenetek egyfajta köztes réteget képeznek az alkalmazások között, amely lehetővé teszi a rugalmas és hatékony kommunikációt.

# Message queue előnyei



## ■ Hibatűrés

- A message queue használata lehetővé teszi az alkalmazások számára, hogy toleránsabbak legyenek a hibákkal szemben, mivel az üzenetek tárolása és újraküldése könnyen implementálható.
- Ha például az alkalmazás egy hálózati hibával találkozik, az üzenetek továbbra is továbbíthatók a queue-on keresztül, és az alkalmazások folytathatják a működést, amint a hiba elhárult.

## ■ Skálázhatóság

- A message queue lehetővé teszi az alkalmazások számára, hogy könnyen skálázzák a folyamataikat és az erőforrásait, mivel az üzenetek feldolgozásának terhe szétteríthető több feldolgozó egység között.
- Az üzeneteket szét lehet osztani több szerver között, így az alkalmazások képesek kezelni a nagyobb terhelést és a folyamatos növekedést.

# Message queue előnyei



## ■ Multiplatform

- A message queue rendszerek több platformon és nyelven használhatóak, így lehetővé teszik az alkalmazások számára, hogy hatékonyan kommunikáljanak egymással akár különböző környezetekben is.

## ■ Adatvédelem

- A message queue használata segíti az adatok biztonságos átvitelét, mivel az üzenetek titkosított formában kerülnek továbbításra.

## ■ Összességében a message queue használata lehetővé teszi a fejlesztők számára, hogy megbízható, skálázható és könnyen karbantartható alkalmazásokat hozzanak létre, amelyek hatékonyan kommunikálnak egymással.

# Message queue hátrányai



## ■ Komplexitás

- A message queue használata bonyolultabbá teheti az alkalmazás fejlesztését, különösen akkor, ha nagy mennyiségű adatot kell átvinni.
- Az üzenetek kezelése és azok átvitele időigényes lehet, és a fejlesztőknek meg kell érteniük a message queue rendszer működését.

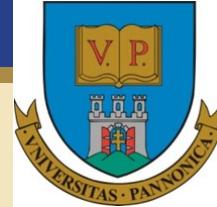
## ■ Rendszerfüggetlenség

- A message queue rendszerek általában nem teljesen függetlenek a rendszerkörnyezettől, így a fejlesztőknek meg kell vizsgálniuk, hogy az adott rendszer milyen platformokat és operációs rendszereket támogat.

## ■ Skálázhatóság

- A message queue használata ugyan lehetővé teszi a skálázhatóságot, de a túl sok üzenet továbbítása vagy az üzenetek átfogó feldolgozása problémákat okozhat a rendszer teljesítményében, és előfordulhat, hogy további hardvereszközökre van szükség a teljesítmény növelése érdekében.

# Message queue hátrányai



## ■ Túlzott terhelés

- Az üzenetek túl gyakori továbbítása vagy az üzenetek túl nagy mérete a message queue rendszerek túlterheléséhez vezethet, amelyek csökkenthetik a rendszer hatékonyságát és teljesítményét.

## ■ Időbeli követelmények

- Az üzeneteknek valós időben kell továbbítaniuk az információt a fogadó alkalmazásokhoz, ami időbeli követelményeket jelenthet az alkalmazások számára, különösen ha nagy adatmennyiséget kell átvinni.

## ■ Adatvédelem

- Habár a message queue rendszerek lehetővé teszik az adatok titkosítását és biztonságos átvitelét, de az adatok továbbra is sebezhetők lehetnek a hackerek és a kártevők támadásai ellen.
- Ezért fontos megfelelő biztonsági intézkedéseket hozni a rendszer védelme érdekében.

## ■ Összességében a message queue rendszerek előnyei messze felülmúlják a hátrányokat, de a fejlesztőknek figyelembe kell venniük ezeket a tényezőket a tervezési és fejlesztési folyamat során.

# Message queue implementációk



## ■ Broker alapú megoldások

- A broker alapú message queue rendszerekben az üzeneteket a központi brókernek kell továbbítani, amely továbbítja azokat a megfelelő vevőnek. Ezek a rendszerek általában jól skálázhatók, és támogatják a többféle nyelvű üzeneteket is.

## ■ Publish-subscribe rendszerek

- A publish-subscribe rendszerekben az üzeneteket előfizetőknek továbbítják, akik azokat megkapják és feldolgozzák. Ezek a rendszerek skálázhatóak és rugalmasak, és lehetővé teszik az üzenetek több felhasználónak való továbbítását is.

## ■ Message queue protokollok

- A message queue protokollok olyan egyszerű, lightweight rendszerek, amelyek szinkronizálni tudják a különböző szolgáltatásokat. A protokollok a REST API-kkal együttműködve könnyen integrálhatóak az alkalmazásokba.

## ■ In-memory message queue rendszerek

- Az in-memory message queue rendszerek a memóriában tárolják az üzeneteket, így rendkívül gyorsak és hatékonyak. Ezek a rendszerek kis mennyiségű adat továbbítására javasoltak, és általában jól alkalmazhatóak a tesztelési és fejlesztési környezetekben.

## ■ Cloud message queue szolgáltatások

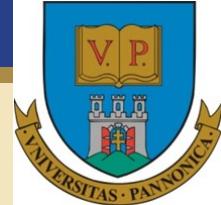
- A cloud message queue szolgáltatások felhőalapúak és skálázhatóak, így kiválóan alkalmasak a nagy adatmennyiségek kezelésére és az alkalmazások közötti integrációra.

# Message queue implementációk



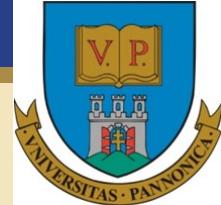
- Ezenkívül a message queue rendszerek különböző nyelveken és keretrendszerekben is implementálhatóak, például
  - a RabbitMQ,
  - az Apache Kafka,
  - a Redis,
  - az Amazon SQS vagy
  - az Azure Service Bus.
- Az alkalmazás specifikus követelmények és igények határozzák meg, hogy melyik megoldás a legmegfelelőbb.

# RabbitMQ



- A RabbitMQ egy nyílt forráskódú, üzenetközvetítő szoftver, amelyet a message queue rendszerekhez használnak.
- A RabbitMQ egy nagyon skálázható és megbízható rendszer, amely támogatja az üzenetek továbbítását több protokollon keresztül, többek között AMQP, MQTT és STOMP.
- A RabbitMQ számos kiegészítő modult és plugin-t tartalmaz, amelyekkel bővíthető a funkcionalitása.
- A rendszer támogatja az üzenetek átvitelét és feldolgozását a felhőben és helyi környezetben is.
- A RabbitMQ könnyen integrálható más alkalmazásokba és környezetekbe, például a Docker és a Kubernetes is támogatja.

# Apache Kafka



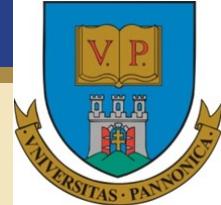
- Az Apache Kafka egy nyílt forráskódú, elosztott streaming platform, amely lehetővé teszi az adatfolyamok hatékony kezelését és feldolgozását.
- A Kafka alapvetően egy publish-subscribe rendszer, amelynek célja az adatok folyamatos, valós idejű átvitele és feldolgozása nagy mennyiségű adatok esetén is.
- A rendszer nagyon skálázható és megbízható, és támogatja a több száz ezer üzenet per másodperc kezelését is.
- A Kafka API-kat számos nyelven írt alkalmazásokkal lehet használni, és könnyen integrálhatóak az Apache Spark, a Storm vagy a Flink adatfeldolgozási keretrendszerekkel.
- A Kafka rendszer különösen alkalmas a nagyvállalatok számára, amelyeknek nagy mennyiségű adatot kell kezelniük és feldolgozniuk.

# Redis



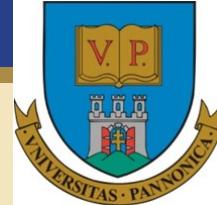
- A Redis egy nyílt forráskódú, memóriában tároló adatstruktúra-tároló rendszer, amely a szerver oldali gyorsítótárazást, a session nyilvántartást, az üzenetsorozatot, a valós idejű adatfolyamok kezelését és más feladatokat támogat.
- A Redis lehetővé teszi az adatok gyors olvasását és írását a memóriában, és így nagyon gyors teljesítményt biztosít.
- A rendszer támogatja az összetett adatstruktúrákat, mint például a listák, a halmazok és a térképek, és a számos kiegészítő modulnak köszönhetően bővíthető a funkcionalitása.
  - Például a RedisGraph modul lehetővé teszi a gráfadatok tárolását és feldolgozását.
- A Redis nagyon könnyen integrálható más alkalmazásokkal és környezetekkel, és támogatja a klaszterezést, így nagyobb skálázhatóságot biztosít.

# Amazon SQS



- Az Amazon SQS (Simple Queue Service) egy teljesen kezelt MQ szolgáltatás, amelyet az Amazon Web Services biztosít.
- Az SQS lehetővé teszi az alkalmazások számára az üzenetek feldolgozását aszinkron módon, és skálázható megoldást biztosít az üzenetek tárolására és továbbítására.
- A rendszer támogatja a standard és a FIFO sorokat, amelyek közül az utóbbi garantálja az üzenetek szigorú időrendi sorrendjét.
- Az SQS lehetővé teszi a könnyű integrációt más AWS szolgáltatásokkal, például az Amazon S3, az Amazon EC2 és az AWS Lambda szolgáltatásokkal.
- A rendszer biztonságos és megbízható, és a felhasználók csak az általuk használt erőforrásokért fizetnek, így a költségek is rugalmasan skálázhatóak.

# Azure Service Bus



- Az Azure Service Bus egy teljesen kezelt, felhő alapú üzenetküldő és fogadó szolgáltatás az Azure felhőplatformon.
- A Service Bus lehetővé teszi az üzenetek feldolgozását aszinkron módon és támogatja a FIFO-t, a topic-okat és feliratkozásokat, a partíciókat, a beágyazott üzeneteket és más összetett üzenetküldő és fogadó scenáriókat.
- A rendszer biztonságos és megbízható, támogatja a tranzakciókat és az üzenetek titkosítását.
- Az Azure Service Bus lehetővé teszi a könnyű integrációt más Azure szolgáltatásokkal, például az Azure Functions, az Azure Logic Apps és az Azure Kubernetes Service szolgáltatásokkal.
- A rendszernek számos árazási lehetősége van, amelyek lehetővé teszik a felhasználók számára a rugalmas költség tervezést.

# A MQ valós alkalmazásai



## ■ Elosztott rendszerek

- A message queue alkalmazása lehetővé teszi a komponensek közötti kommunikációt az elosztott rendszerekben, és lehetővé teszi a részrendszerek független skálázását.

## ■ Mikroszolgáltatás-architektúrák

- A message queue használata lehetővé teszi a különböző mikroszolgáltatások közötti kommunikációt és integrációt, így támogatva az alkalmazások független skálázását és karbantarthatóságát.

## ■ Késleltetett feladatok

- A message queue lehetővé teszi az időzített feladatok beállítását, például az e-mailek késleltetett elküldését vagy a folyamatok újraindítását hiba esetén.

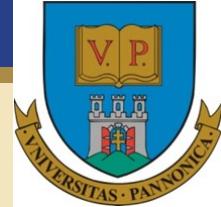
## ■ Nagy adatforgalmú alkalmazások

- A message queue lehetővé teszi az üzenetek gyors és hatékony továbbítását és feldolgozását a nagy adatforgalmú alkalmazásokban.

## ■ Fogadó oldali webes alkalmazások

- A message queue lehetővé teszi a webes alkalmazások számára az aszinkron feldolgozást és az erőforrások hatékony kihasználását, így javítva a felhasználói élményt és a rendszer skálázhatóságát.

# Hitelesítés a MQ-ban



- A message queue rendszerek különböző hitelesítési mechanizmusokat támogatnak, attól függően, hogy melyiket használjuk.
  - Az általános hitelesítési mechanizmusok közé tartozik például a felhasználónév/jelszó alapú hitelesítés, a token alapú hitelesítés, az X.509 tanúsítványok használata, stb.
- Amikor a szoftver csatlakozik a message queue-hoz, meg kell adni az azonosítási adatokat a hitelesítéshez.
  - Azonosító adatokat általában a felhasználónév, a jelszó, a kulcsok vagy a tanúsítványok jelenthetik.
  - Ezeket az adatokat általában az alkalmazás konfigurációs fájljában vagy a környezeti változókban lehet tárolni.
- A message queue-k további biztonsági mechanizmusokat is kínálnak, például az SSL/TLS titkosítást, amely segít védeni a kommunikációt a külső támadásoktól.
- A message queue-kban általában szerepelnek adminisztrációs felületek is, amelyeken keresztül lehet felhasználókat létrehozni, jogosultságokat kezelni, figyelmeztetéseket konfigurálni és más biztonsági beállításokat végrehajtani.

# MQ szerver beüzemelése



## ■ Válaszd ki a szerver szoftvert

- Az előzőekben már bemutattam néhányat a message queue szerverek közül, például a RabbitMQ, az Apache Kafka vagy a Redis. Válassz egyet ezek közül a szoftverek közül, amely megfelel az igényeidnek.

## ■ Telepítsd a szoftvert a szerveredre

- A kiválasztott szoftver telepítése a szerverre a szoftver dokumentációja szerint történik. Előfordulhat, hogy a telepítéshez szükséges egy adatbázis vagy más külső szoftver telepítése is.

## ■ Konfiguráld a szoftvert

- A szerver beállításai a szoftver dokumentációja szerint módosíthatóak. Általában a szerver konfigurációs fájljában lehet megadni a beállításokat.

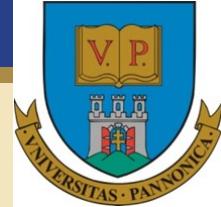
## ■ Teszteld a szoftvert

- A szoftver tesztelése az alkalmazásban történik. Hozz létre egyszerű példaprogramokat, amelyek küldenek és fogadnak üzeneteket a szerverrel.

## ■ A pontos lépések a kiválasztott message queue szervertől függően változhatnak. Érdemes alaposan tanulmányozni a szerver dokumentációját, és követni a lépéseit.

## ■ Azt is fontos megemlíteni, hogy a message queue szerverek általában erőforrásigényesek, ezért fontos biztosítani a megfelelő hardvert és szerverinfrastruktúrát a szerver futtatásához.

# Elérhető MQ szerverek



- RabbitMQ: <https://www.rabbitmq.com/>
- Apache ActiveMQ: <https://activemq.apache.org/>
- Apache Kafka: <https://kafka.apache.org/>
- Redis: <https://redis.io/>
- Amazon Simple Queue Service (SQS):  
<https://aws.amazon.com/sqs/>
- Google Cloud Pub/Sub: <https://cloud.google.com/pubsub>
- Microsoft Azure Service Bus:  
<https://azure.microsoft.com/en-us/services/service-bus/>
- Ezen kiszolgálók mindegyike különböző erősségekkel és korlátokkal rendelkezik, így fontos az adott felhasználási esetnek megfelelő kiszolgáló kiválasztása.
- További információk és telepítési útmutatók megtalálhatók a fenti linkeken.



EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

# ENTITY FRAMEWORK CORE

<https://www.entityframeworktutorial.net/efcore/entity-framework-core.aspx>  
<https://www.learnentityframeworkcore.com/>  
<https://docs.microsoft.com/en-us/ef/core/>

SZÉCHENYI 2020



MAGYARORSZÁG  
KORMÁNYA

Európai Unió  
Európai Strukturális  
és Beruházási Alapok



BEFETKTETÉS A JÖVŐBE

# Introduction

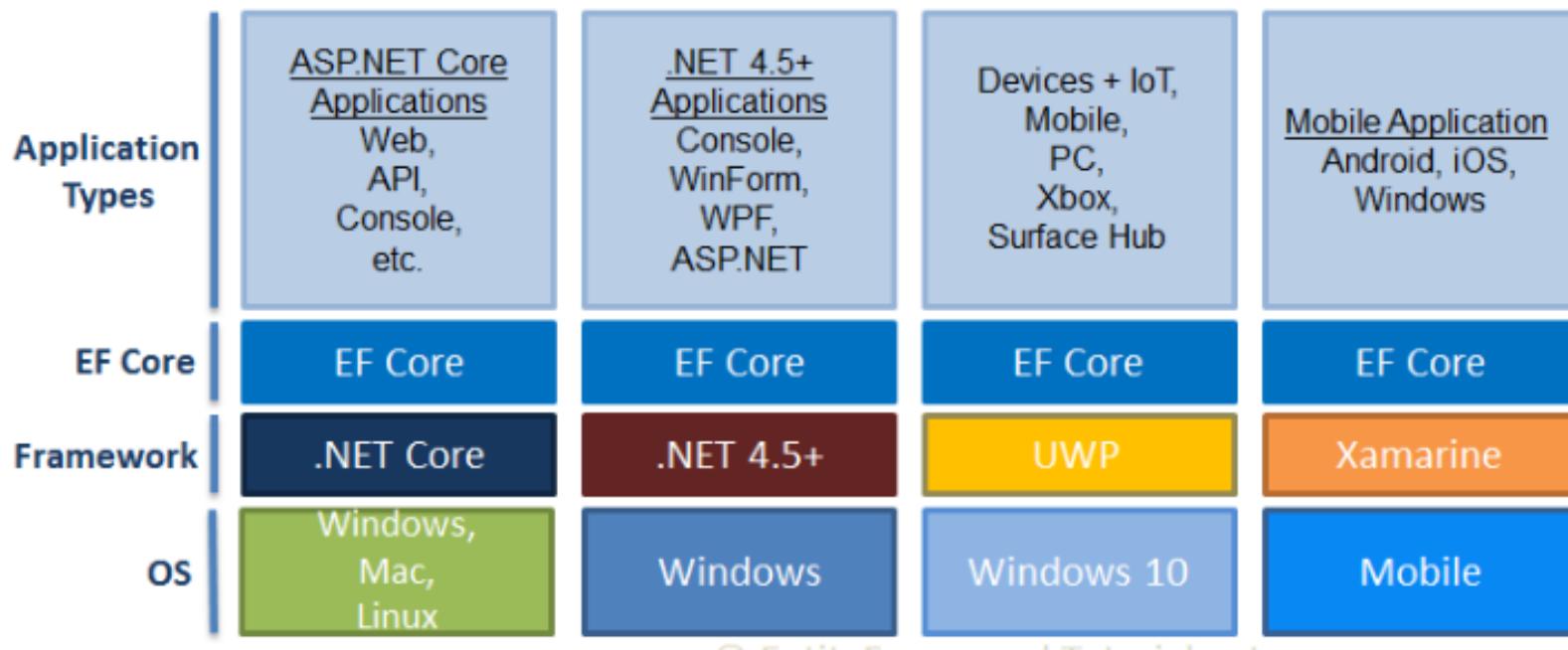
EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

- Entity Framework Core open-source, lightweight, extensible and a cross-platform version of Entity Framework data access technology.
- Entity Framework is an Object/Relational Mapping (O/RM) framework. It is an enhancement to ADO.NET that gives developers an automated mechanism for accessing & storing the data in the database.
- EF Core is intended to be used with .NET Core applications. However, it can also be used with standard .NET 4.5+ framework based applications.

# Introduction

- The following figure illustrates the supported application types, .NET Frameworks and OSs.

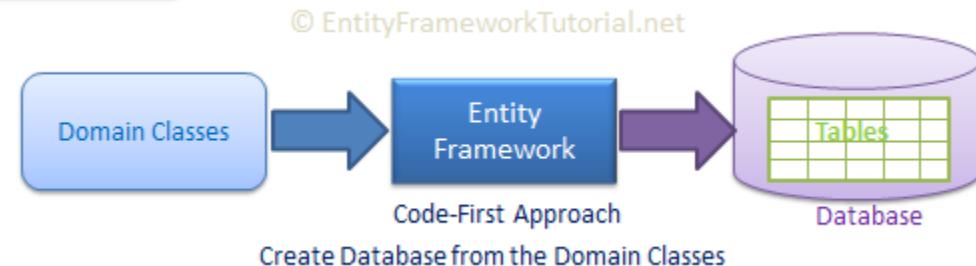
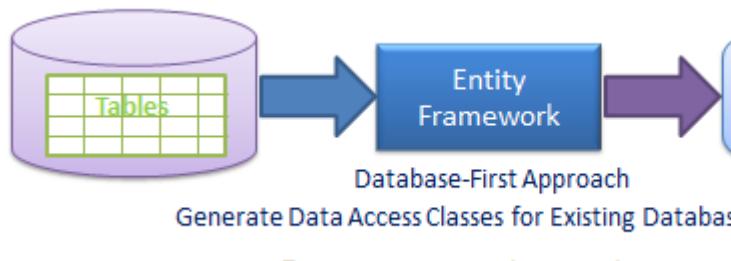


# EF Core Development Approaches

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

- EF Core supports two development approaches
  - Code-First: EF Core API creates the database and tables using migration based on the conventions and configuration provided in your domain classes. This approach is useful in Domain Driven Design (DDD).
  - Database-First: EF Core API creates the domain and context classes based on your existing database using EF Core commands. This has limited support in EF Core as it does not support visual designer or wizard.



# EF Core Database Providers

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- Entity Framework Core uses a provider model to access many different databases. EF Core includes providers as NuGet packages which you need to install.

Database	NuGet Package
SQL Server	<a href="#">Microsoft.EntityFrameworkCore.SqlServer</a>
MySQL	<a href="#">MySql.Data.EntityFrameworkCore</a>
PostgreSQL	<a href="#">Npgsql.EntityFrameworkCore.PostgreSQL</a>
SQLite	<a href="#">Microsoft.EntityFrameworkCore.SQLite</a>
SQL Compact	<a href="#">EntityFrameworkCore.SqlServerCompact40</a>
In-memory	<a href="#">Microsoft.EntityFrameworkCore.InMemory</a>

- EF Core is not a part of .NET Core and standard .NET framework. It is available as a NuGet package. You need to install NuGet packages for the following two things to use EF Core in your application:
  - EF Core DB provider
    - Use the following .NET Core CLI command from the operating system's command line to install or update the EF Core SQL Server provider:
    - [`dotnet add package Microsoft.EntityFrameworkCore.SqlServer`](#)
  - EF Core tools
    - Use the following .NET Core CLI command from the operating system's command line to install or update the EF Core Tools:
    - [`dotnet add package Microsoft.EntityFrameworkCore.Tools`](#)

- The DbContext class is an integral part of Entity Framework. An instance of DbContext represents a session with the database which can be used to query and save instances of your entities to a database. DbContext is a combination of the Unit Of Work and Repository patterns.
- DbContext in EF Core allows us to perform following tasks:
  - Manage database connection
  - Configure model & relationship
  - Querying database
  - Saving data to the database
  - Configure change tracking
  - Caching
  - Transaction management

- To use DbContext in our application, we need to create the class that derives from DbContext, also known as context class. This context class typically includes DbSet< TEntity > properties for each entity in the model. Consider the following example of context class in EF Core.

```
public class SchoolContext : DbContext
{
    public SchoolContext()
    {

    }

    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
    }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
    }
    //entities
    public DbSet<Student> Students { get; set; }
    public DbSet<Course> Courses { get; set; }
}
```

- In the example above, the SchoolContext class is derived from the DbContext class and contains the DbSet< TEntity > properties of Student and Course type.
- It also overrides the OnConfiguring and OnModelCreating methods.
  - The OnConfiguring() method allows us to select and configure the data source to be used with a context using DbContextOptionsBuilder.
  - The OnModelCreating() method allows us to configure the model using ModelBuilder Fluent API.
- We must create an instance of SchoolContext to connect to the database and save or retrieve Student or Course data.

# Migration

- Migration is a way to keep the database schema in sync with the EF Core model by preserving data.



- As per the above figure, EF Core API builds the EF Core model from the domain (entity) classes and EF Core migrations will create or update the database schema based on the EF Core model.
- Whenever you change the domain classes, you need to run migration to keep the database schema up to date.

# Migration

- EF Core migrations are a set of commands which you can execute in NuGet Package Manager Console or in dotnet Command Line Interface (CLI).
- The following table lists important migration commands in EF Core.

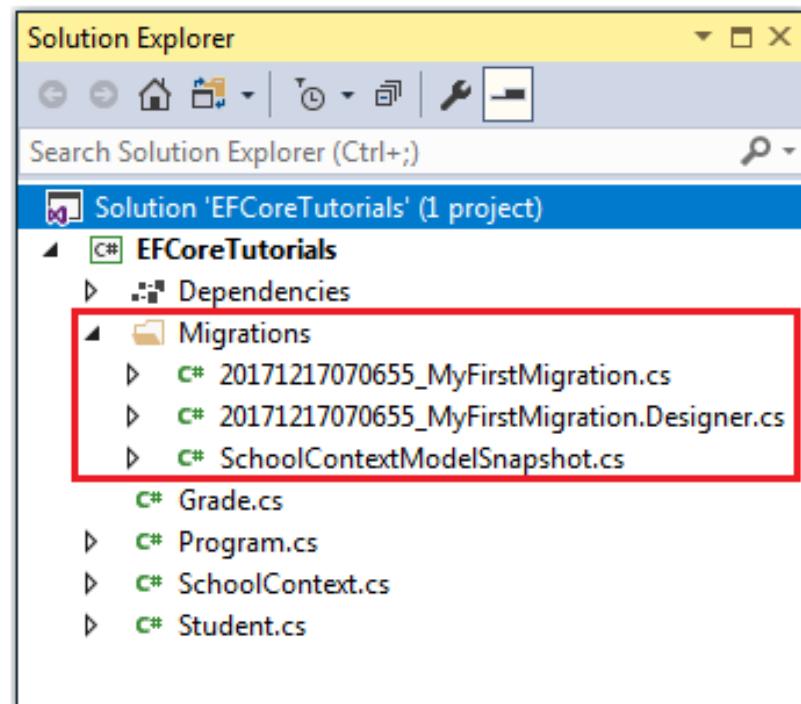
PMC Command	dotnet CLI command	Usage
add-migration <migration name>	Add <migration name>	Creates a migration by adding a migration snapshot.
Remove-migration	Remove	Removes the last migration snapshot.
Update-database	Update	Updates the database schema based on the last migration snapshot.
Script-migration	Script	Generates a SQL script using all the migration snapshots.

- Adding a Migration

- At the very first time, you defined the initial domain classes. At this point, there is no database for your application which can store the data from your domain classes. So, firstly, you need to create a migration.
  - `dotnet ef migrations add MyFirstMigration`
- In the above commands, MyFirstMigration is the name of a migration. This will create three files in the Migrations folder of your project, as shown below.

# Migration

- `<timestamp>_<Migration Name>.cs`: The main migration file which includes migration operations in the `Up()` and `Down()` methods. The `Up()` method includes the code for creating DB objects and `Down()` method includes code for removing DB objects.
- `<timestamp>_<Migration Name>.Designer.cs`: The migrations metadata file which contains information used by EF Core.
- `<contextclassname>ModelSnapshot.cs`: A snapshot of your current model. This is used to determine what changed when creating the next migration.



- Creating or Updating the Database
  - Use the following command to create or update the database schema.
    - `dotnet ef database update`
  - The Update command will create the database based on the context and domain classes and the migration snapshot, which is created using the add command.
  - If this is the first migration, then it will also create a table called `__EFMigrationsHistory`, which will store the name of all migrations, as and when they will be applied to the database.

- Removing a Migration

- You can remove the last migration if it is not applied to the database. Use the following remove commands to remove the last created migration files and revert the model snapshot.
  - `dotnet ef migrations remove`
- If a migration is already applied to the database, then it will throw an exception.

# Migration

- Reverting a Migration

- Suppose you changed your domain class and created the second migration named MySecondMigration using the add-migration command and applied this migration to the database using the Update command. But, for some reason, you want to revert the database to the previous state. In this case, use the database update <migration name> command to revert the database to the specified previous migration snapshot.
  - `dotnet ef database update MyFirstMigration`
- The above command will revert the database based on a migration named MyFirstMigration and remove all the changes applied for the second migration named MySecondMigration. This will also remove MySecondMigration entry from the `__EFMigrationsHistory` table in the database.

# Migration

- Generating a SQL Script
  - Use the following command to generate a SQL script for the database.
    - `dotnet ef migrations script`
  - The above script command will include a script for all the migrations by default. You can specify a range of migrations by using the `-to` and `-from` options.

# Creating a Model for an Existing Database

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- Use `dotnet ef dbcontext scaffold` command to create a model based on your existing database.
  - `dotnet ef dbcontext scaffold "Server=.\SQLEXPRESS;Database=SchoolDB;Trusted_Connection=True;" Microsoft.EntityFrameworkCore.SqlServer -o Models`
- Here, Server=.\SQLEXPRESS; refers to local SQLEXPRESS database server. Database=SchoolDB; specifies the database name "SchoolDB" for which we are going to create classes. Trusted\_Connection=True; specifies the Windows authentication.
- The second parameter is the provider name.
- The -o parameter specifies the directory where we want to generate all the classes which is the Models folder in this case.

# Conventions

- Conventions are default rules using which Entity Framework builds a model based on your domain (entity) classes.
- Consider the following sample entities and context class to understand the default conventions.

```
public class Student
{
    public int StudentId { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public DateTime DateOfBirth { get; set; }
    public byte[] Photo { get; set; }
    public decimal Height { get; set; }
    public float Weight { get; set; }

    public int GradeId { get; set; }
    public Grade Grade { get; set; }
}
```

```
public class Grade
{
    public int Id { get; set; }
    public string GradeName { get; set; }
    public string Section { get; set; }

    public IList<Student> Students { get; set; }
}
```

```
public class SchoolContext : DbContext
{
    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder.UseSqlServer(@"Server=.\SQLEXPRESS;Database=SchoolDB;Trusted_Connection=True;");
    }

    public DbSet<Student> Students { get; set; }
}
```

# Conventions:

## Table

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

- EF Core will create all the database objects in the dbo schema by default.
- EF Core will create database tables for all DbSet< TEntity > properties in a context class with the same name as the property.
- It will also create tables for entities which are not included as DbSet properties but are reachable through reference properties in other DbSet entities.
- For the above example, EF Core will create the Students table for DbSet< Student > property in the SchoolContext class and the Grade table for a Grade property in the Student entity class, even though the SchoolContext class does not include the DbSet< Grade > property.

# Conventions: Table

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

```
public class Student
{
    public int StudentId { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public DateTime DateOfBirth { get; set; }
    public byte[] Photo { get; set; }
    public decimal Height { get; set; }
    public float Weight { get; set; }

    public int GradeId { get; set; }
    public Grade Grade { get; set; }
}

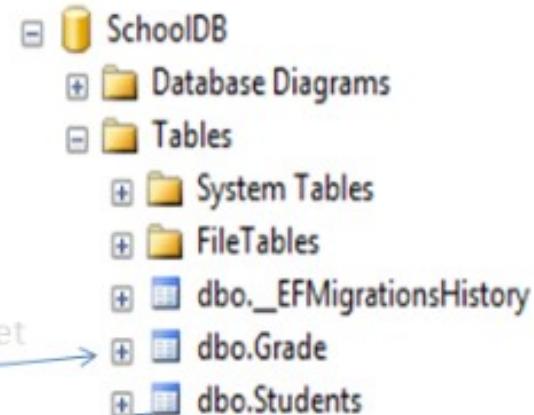
public class SchoolContext : DbContext
{
    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder.UseSqlServer(@"Server=.\SQLEXPRESS;Database=SchoolDB;Trusted;
    }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {

    }

    public DbSet<Student> Students { get; set; }
}
```

© EntityFrameworkTutorial.net



# Conventions: Column

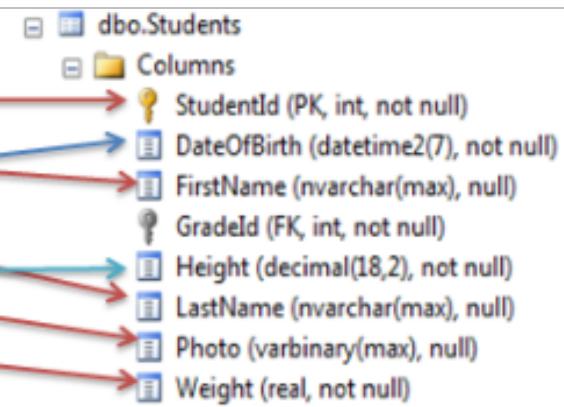
EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

- EF Core will create columns for all the scalar properties of an entity class with the same name as the property, by default.
- It uses the reference and collection properties in building relationships among corresponding tables in the database.

```
public class Student
{
    public int StudentId { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public DateTime DateOfBirth { get; set; }
    public byte[] Photo { get; set; }
    public decimal Height { get; set; }
    public float Weight { get; set; }

    public int GradeId { get; set; }
    public Grade Grade { get; set; }
}
```



# Conventions:

## Column

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

- EF Core creates null columns for all reference data type and nullable primitive type properties.
- EF Core creates NotNull columns in the database for all primary key properties, and primitive type properties.
- EF Core will create the primary key column for the property named Id or <Entity Class Name>Id (case insensitive).
- As per the foreign key convention, EF Core API will create a foreign key column for each reference navigation property in an entity with one of the following naming patterns.
- EF Core creates a clustered index on Primarykey columns and a non-clustered index on ForeignKey columns, by default.

# Conventions: Data Type

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

- The data type for columns in the database table is depending on how the provider for the database has mapped C# data type to the data type of a selected database.

C# Data Type	Mapping to SQL Server Data Type
int	int
string	nvarchar(Max)
decimal	decimal(18,2)
float	real
byte[]	varbinary(Max)
datetime	datetime
bool	bit
byte	tinyint
short	smallint
long	bigint
double	float
char	No mapping
sbyte	No mapping (throws exception)
object	No mapping

# Configurations

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

- Many times we want to customize the entity to table mapping and do not want to follow default conventions.
- EF Core allows us to configure domain classes in order to customize the EF model to database mappings.
- There are two ways to configure domain classes in EF Core.
  - By using Data Annotation Attributes
  - By using Fluent API

# Data Annotation

## Attributes

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

- Data Annotations is a simple attribute based configuration method where different .NET attributes can be applied to domain classes and properties to configure the model.
- Data annotation attributes are not dedicated to Entity Framework, as they are also used in ASP.NET MVC.
  - This is why these attributes are included in separate namespace System.ComponentModel.DataAnnotations.
  - `dotnet add package System.ComponentModel.DataAnnotations`
- The following example demonstrates how the data annotations attributes can be applied to a domain class and properties to override conventions.

# Data Annotation Attributes

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

```
[Table("StudentInfo")]
public class Student
{
    public Student() { }

    [Key]
    public int SID { get; set; }

    [Column("Name", TypeName="ntext")]
    [MaxLength(20)]
    public string StudentName { get; set; }

    [NotMapped]
    public int? Age { get; set; }

    public int StdId { get; set; }

    [ForeignKey("StdId")]
    public virtual Standard Standard { get; set; }
}
```

# Data Annotation

## Attributes

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

Attribute	Description
Key	You can use a Key attribute to configure a single property to be the key of an entity.
NotMapped	You can use a NotMapped attribute to exclude a type from the model or any property of the entity.
Required	You can use a Required attribute to indicate that a property is required.
MaxLength	You can use the MaxLength attribute to configure a maximum length for a property.
MinLength	You can use the MinLength attribute to configure a minimum length for a property.
ForeignKey	You can use the Data ForeignKey attribute to configure which property should be used as the foreign key property for a given relationship.
Table	You can use Table attribute to map the class name which is different from the table name in the database.
Column	You can use Column attribute to map the property name which is different from the column name in the database.

- Fluent API is used to configure domain classes to override conventions.
- EF Fluent API is based on a Fluent API design pattern (a.k.a Fluent Interface) where the result is formulated by method chaining.
- In Entity Framework Core, the ModelBuilder class acts as a Fluent API. By using it, we can configure many different things, as it provides more configuration options than data annotation attributes.
- You can use Data Annotation attributes and Fluent API at the same time.
- Note: Fluent API configurations have higher precedence than data annotation attributes.

- Entity Framework Core Fluent API configures the following aspects of a model:
  - Model Configuration: Configures an EF model to database mappings. Configures the default Schema, DB functions, additional data annotation attributes and entities to be excluded from mapping.
  - Entity Configuration: Configures entity to table and relationships mapping e.g. PrimaryKey, AlternateKey, Index, table name, one-to-one, one-to-many, many-to-many relationships etc.
  - Property Configuration: Configures property to column mapping e.g. column name, default value, nullability, Foreignkey, data type, concurrency column etc.

# Fluent API

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

Configurations	Fluent API Methods	Usage
Model-wide Configurations	HasDefaultSchema()	Specifies the default database schema.
	ComplexType()	Configures the class as complex type.
Entity Configurations	HasIndex()	Configures the index property for the entity type.
	HasKey()	Configures the primary key property for the entity type.
	HasMany()	Configures the Many relationship for one-to-many or many-to-many relationships.
	HasOptional()	Configures an optional relationship which will create a nullable foreign key in the database.
	HasRequired()	Configures the required relationship which will create a non-nullable foreign key column in the database.
	Ignore()	Configures that the class or property should not be mapped to a table or column.
	Map()	Allows advanced configuration related to how the entity is mapped to the database schema.
	MapToStoredProcedures()	Configures the entity type to use INSERT, UPDATE and DELETE stored procedures.
	ToTable()	Configures the table name for the entity.

# Fluent API

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

Property Configurations	HasColumnAnnotation()	Sets an annotation in the model for the database column used to store the property.
	IsRequired()	Configures the property to be required on SaveChanges().
	IsConcurrencyToken()	Configures the property to be used as an optimistic concurrency token.
	IsOptional()	Configures the property to be optional which will create a nullable column in the database.
	HasParameterName()	Configures the name of the parameter used in the stored procedure for the property.
	HasDatabaseGeneratedOption()	Configures how the value will be generated for the corresponding column in the database e.g. computed, identity or none.
	HasColumnOrder()	Configures the order of the database column used to store the property.
	HasColumnType()	Configures the data type of the corresponding column of a property in the database.
	HasColumnName()	Configures the corresponding column name of a property in the database.
	IsConcurrencyToken()	Configures the property to be used as an optimistic concurrency token.

- Override the `OnModelCreating` method and use a parameter `modelBuilder` of type `ModelBuilder` to configure domain classes, as shown below.

```
public class SchoolDBContext: DbContext
{
    public DbSet<Student> Students { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        //Write Fluent API configurations here

        //Property Configurations
        modelBuilder.Entity<Student>()
            .Property(s => s.StudentId)
            .HasColumnName("Id")
            .HasDefaultValue(0)
            .IsRequired();
    }
}
```

# Configure One-to-Many Relationships

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

- Generally, you don't need to configure one-to-many relationships because EF Core includes enough conventions which will automatically configure them.
- However, you can use Fluent API to configure the one-to-many relationship if you decide to have all the EF configurations in Fluent API for easy maintenance.
- Entity Framework Core made it easy to configure relationships using Fluent API.

# Configure One-to-Many Relationships

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

- Configure the one-to-many relationship for the above entities using Fluent API by overriding the OnModelCreating method in the context class, as shown below.

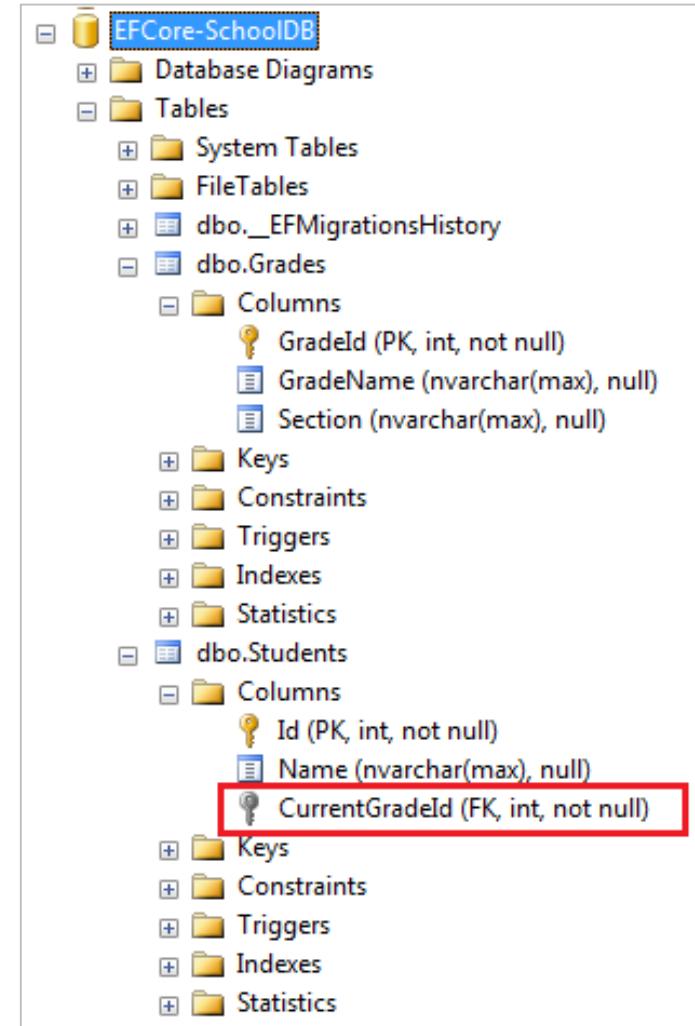
```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Student>()
        .HasOne<Grade>(s => s.Grade)
        .WithMany(g => g.Students)
        .HasForeignKey(s => s.CurrentGradeId);
}
```

# Configure One-to-Many Relationships

- Now, to reflect this in the database, execute migration commands.
- The database will include two tables with One-to-Many relationship.

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen



# Configure One-to-Many Relationships

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

- First, we need to start configuring with one entity class, either Student or Grade. So, modelBuilder.Entity<Student>() starts with the Student entity.
- Then, .HasOne<Grade>(s => s.Grade) specifies that the Student entity includes a Grade type property named Grade.
- Now, we need to configure the other end of the relationship, the Grade entity. The .WithMany(g => g.Students) specifies that the Grade entity class includes many Student entities. Here, WithMany infers collection navigation property.
- The .HasForeignKey<int>(s => s.CurrentGradeId); specifies the name of the foreign key property CurrentGradeId. This is optional. Use it only when you have the foreign key Id property in the dependent class.

# Configure One-to-Many Relationships

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

- Alternatively, you can start configuring the relationship with the Grade entity instead of the Student entity, as shown below.

```
modelBuilder.Entity<Grade>()
    .HasMany<Student>(g => g.Students)
    .WithOne(s => s.Grade)
    .HasForeignKey(s => s.CurrentGradeId);
```

# Configure One-to-Many Relationships

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

- Cascade delete automatically deletes the child row when the related parent row is deleted.
- Use the OnDelete method to configure the cascade delete between Student and Grade entities, as shown below.

```
modelBuilder.Entity<Grade>()
    .HasMany<Student>(g => g.Students)
    .WithOne(s => s.Grade)
    .HasForeignKey(s => s.CurrentGradeId)
    .OnDelete(DeleteBehavior.Cascade);
```

# Configure One-to-Many Relationships

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

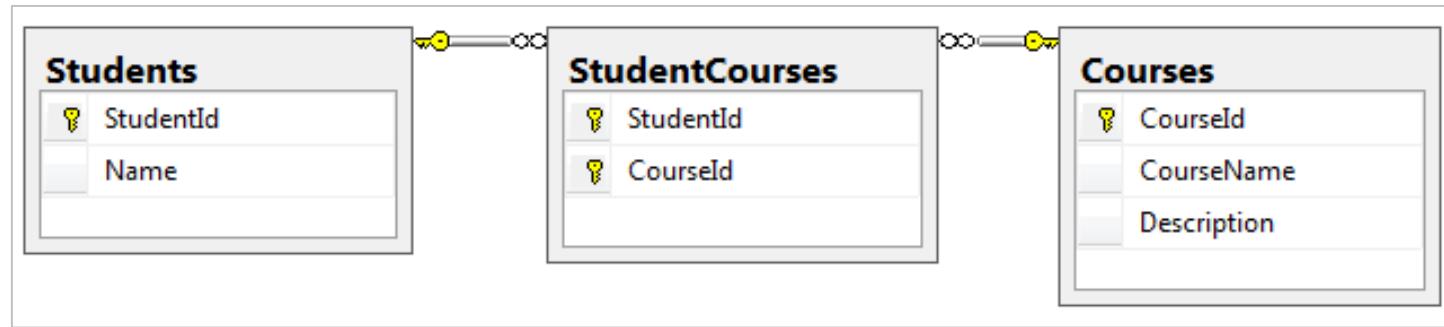
- The OnDelete() method cascade delete behaviour uses the DeleteBehavior parameter.
- You can specify any of the following DeleteBehavior values, based on your requirement.
  - Cascade : Dependent entities will be deleted when the principal entity is deleted.
  - Restrict: Prevents Cascade delete.
  - SetNull: The values of foreign key properties in the dependent entities will be set to null.

# Configure Many-to-Many Relationships

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

- The many-to-many relationship in the database is represented by a joining table which includes the foreign keys of both tables. Also, these foreign keys are composite primary keys.



- There are no default conventions available in Entity Framework Core which automatically configure a many-to-many relationship. You must configure it using Fluent API.

# Configure Many-to-Many Relationships

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

- We must create a joining entity class for a joining table. The joining entity for the above Student and Course entities should include a foreign key property and a reference navigation property for each entity.
- The steps for configuring many-to-many relationships would be the following:
  - Define a new joining entity class which includes the foreign key property and the reference navigation property for each entity.
  - Define a one-to-many relationship between other two entities and the joining entity, by including a collection navigation property in entities at both sides (Student and Course, in this case).
  - Configure both the foreign keys in the joining entity as a composite key using Fluent API.

# Configure Many-to-Many Relationships

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- So, first of all, define the joining entity StudentCourse, as shown below.
- The above joining entity StudentCourse includes reference navigation properties Student and Course and their foreign key properties StudentId and CourseId respectively (foreign key properties follow the convention).

```
public class StudentCourse
{
    public int StudentId { get; set; }
    public Student Student { get; set; }

    public int CourseId { get; set; }
    public Course Course { get; set; }
}
```

# Configure Many-to-Many Relationships

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- Now, we also need to configure two separate one-to-many relationships between Student -> StudentCourse and Course -> StudentCourse entities.
- As you can see above, the Student and Course entities now include a collection navigation property of StudentCourse type.
- The StudentCourse entity already includes the foreign key property and navigation property for both, Student and Course.

```
public class Student
{
    public int StudentId { get; set; }
    public string Name { get; set; }

    public IList<StudentCourse> StudentCourses { get; set; }
}

public class Course
{
    public int CourseId { get; set; }
    public string CourseName { get; set; }
    public string Description { get; set; }

    public IList<StudentCourse> StudentCourses { get; set; }
}
```

# Configure Many-to-Many Relationships

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- This makes it a fully defined one-to-many relationship between Student & StudentCourse and Course & StudentCourse.
- Now, the foreign keys must be the composite primary key in the joining table. This can only be configured using Fluent API.

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<StudentCourse>().HasKey(sc => new { sc.StudentId, sc.CourseId });
}
```

- modelBuilder.Entity<StudentCourse>().HasKey(sc => new { sc.StudentId, sc.CourseId }) configures StudentId and CourseId as the composite key.

# Configure Many-to-Many Relationships

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- Suppose that the foreign key property names do not follow the convention (e.g. SID instead of StudentId and CID instead of CourseId), then you can configure it using Fluent API.

```
modelBuilder.Entity<StudentCourse>().HasKey(sc => new { sc.SId, sc.CId });

modelBuilder.Entity<StudentCourse>()
    .HasOne<Student>(sc => sc.Student)
    .WithMany(s => s.StudentCourses)
    .HasForeignKey(sc => sc.SId);

modelBuilder.Entity<StudentCourse>()
    .HasOne<Course>(sc => sc.Course)
    .WithMany(s => s.StudentCourses)
    .HasForeignKey(sc => sc.CId);
```

- Entity Framework Core uses Language Integrated Query (LINQ) to query data from the database.
- LINQ allows you to use C# (or your .NET language of choice) to write strongly typed queries. It uses your derived context and entity classes to reference database objects.
- EF Core passes a representation of the LINQ query to the database provider. Database providers in turn translate it to database-specific query language (for example, SQL for a relational database).

- In addition to LINQ extension methods, we can use the Find() method of DbSet to search the entity based on the primary key value.

```
var ctx = new SchoolDBEntities();
var student = ctx.Students.Find(1);
```

- In the example, ctx.Student.Find(1) returns a student record whose StudentId is 1 in the database. If no record is found, then it returns null.

# Querying

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

- If you want to get a single student object, when there are many students, whose name is "Bill" in the database, then use First or FirstOrDefault.

```
using (var ctx = new SchoolDBEntities())
{
    var student = ctx.Students
        .Where(s => s.StudentName == "Bill")
        .FirstOrDefault<Student>();
}
```

- EF Core executes the following query in the database.

```
SELECT TOP (1)
[s].[StudentId], [s].[DoB], [s].[FirstName], [s].[GradeId],
[s].[LastName], [s].[MiddleName]
FROM [Students] AS [s]
WHERE [s].[FirstName] = N'Bill'
```

# Querying

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

- The difference between First and FirstOrDefault is that First() will throw an exception if there is no result data for the supplied criteria, whereas FirstOrDefault() returns a default value (null) if there is no result data.

- The `ToListAsync` method returns the collection result. If you want to list all the students with the same name then use `ToListAsync()`.

```
using (var ctx = new SchoolDBEntities())
{
    var studentList = ctx.Students.Where(s => s.StudentName == "Bill").ToList();
}
```

- We may also use `ToArray`, `ToDictionary` or `ToLookup`.

# Eager Loading

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

- Entity Framework Core supports eager loading of related entities, using the `Include()` extension method and projection query.
- In addition to this, it also provides the `ThenInclude()` extension method to load multiple levels of related entities.

```
var context = new SchoolContext();

var studentWithGrade = context.Students
    .Where(s => s.FirstName == "Bill")
    .Include(s => s.Grade)
    .FirstOrDefault();
```

# Eager Loading

- In the above example, `.Include(s => s.Grade)` passes the lambda expression `s => s.Grade` to specify a reference property to be loaded with Student entity data from the database in a single SQL query.
- The above query executes the following SQL query in the database.

```
SELECT TOP(1) [s].[StudentId], [s].[DoB], [s].[FirstName], [s].[GradeId], [s].[LastName],  
    [s].[MiddleName], [s.Grade].[GradeId], [s.Grade].[GradeName], [s.Grade].[Section]  
FROM [Students] AS [s]  
LEFT JOIN [Grades] AS [s.Grade] ON [s].[GradeId] = [s.Grade].[GradeId]  
WHERE [s].[FirstName] = N'Bill'
```

# Eager Loading

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

- Use the `Include()` method multiple times to load multiple navigation properties of the same entity. For example, the following code loads Grade and StudentCourses related entities of Student.

```
var context = new SchoolContext();

var studentWithGrade = context.Students.Where(s => s.FirstName == "Bill")
    .Include(s => s.Grade)
    .Include(s => s.StudentCourses)
    .FirstOrDefault();
```

# Eager Loading

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

- The ThenInclude() extension method to load multiple levels of related entities.

```
var context = new SchoolContext();

var student = context.Students.Where(s => s.FirstName == "Bill")
    .Include(s => s.Grade)
    .ThenInclude(g => g.Teachers)
    .FirstOrDefault();
```

- In the above example, .Include(s => s.Grade) will load the Grade reference navigation property of the Student entity. .ThenInclude(g => g.Teachers) will load the Teacher collection property of the Grade entity.
- The ThenInclude method must be called after the Include method.

# Projection Query

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

- We can also load multiple related entities by using the projection query instead of `Include()` or `ThenInclude()` methods.
- The following example demonstrates the projection query to load the Student, Grade, and Teacher entities.

```
var context = new SchoolContext();

var stud = context.Students.Where(s => s.FirstName == "Bill")
    .Select(s => new
{
    Student = s,
    Grade = s.Grade,
    GradeTeachers = s.Grade.Teachers
})
    .FirstOrDefault();
```

# Projection Query

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

- In the above example, the .Select extension method is used to include the Student, Grade and Teacher entities in the result. This will execute the same SQL query as the above ThenInclude() method.

# Lazy Loading

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

- Lazy loading of data is a pattern whereby the retrieval of data from the database is deferred until it is actually needed.
- This sounds like a good thing, and in some scenarios, this can help to improve the performance of an application. In other scenarios, it can degrade the performance of an application substantially, particularly so in web applications.
- For this reason, lazy Loading was introduced in EF Core 2.1 as an opt-in feature.
- Lazy loading can be enabled in two ways:
  - Using Proxies
  - Using the ILazyLoader service

# Lazy Loading

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

- Proxies
  - Proxies are objects deriving from your entities that are generated at runtime by Entity Framework Core.
  - These proxies have behaviour added to them that result in database queries being made as required to load navigation properties on demand.
  - This was the default mechanism used to provide lazy loading in previous version of Entity Framework.
- Enabling lazy loading by proxies requires three steps.

# Lazy Loading

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

- Install the Microsoft.EntityFrameworkCore.Proxies package.
  - add package Microsoft.EntityFrameworkCore.Proxies
- Use the UseLazyLoadingProxies method to enable the creation of proxies in the OnConfiguring method of the DbContext.

```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{
    optionsBuilder.UseLazyLoadingProxies();
}
```

- Make all navigation properties virtual.

# Lazy Loading

```
public class Author
{
    public int AuthorId { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public virtual List<Book> Books { get; set; } = new List<Book>();
}

public class Book
{
    public int BookId { get; set; }
    public string Title { get; set; }
    public int AuthorId { get; set; }
    public virtual Author Author { get; set; }
}
```

- This last step is the key to allowing EF Core to override your entities to create proxies. In addition, all entity types must be public, unsealed, and have a public or protected constructor.

- **ILazyLoader**

- The **ILazyLoader** interface represents a component that is responsible for loading navigation properties if they haven't already been loaded.
- This approach circumvents the generation of proxies which isn't supported on all platforms.
- **ILazyLoader** can be used in one of two ways.
  - It can be injected into the principal entity in the relationship, where it is used to load dependants. This requires that your model class(es) take a dependency on `Microsoft.EntityFrameworkCore.Infrastructure`, which is available in the `Microsoft.EntityFrameworkCore.Abstractions` package.
  - Or you can use a convention-based delegate.
- The following steps detail how to employ the first approach.

# Lazy Loading

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

- Install the `Microsoft.EntityFrameworkCore.Abstractions` package into the project containing your model classes.
  - add package `Microsoft.EntityFrameworkCore.Abstractions`
- Alter the principal entity to include
  - a using directive for `Microsoft.EntityFrameworkCore.Infrastructure`
  - a field for the `ILazyLoader` instance
  - an empty constructor, and one that takes an `ILazyLoader` as a parameter (which can be private, if you prefer)
  - a field for the collection navigation property
  - a getter in the public property that uses the `ILazyLoader.Load` method

# Lazy Loading

```
using Microsoft.EntityFrameworkCore.Infrastructure;
public class Author
{
    private readonly ILazyLoader _lazyLoader;
    public Author()
    {
    }
    public Author(ILazyLoader lazyLoader)
    {
        _lazyLoader = lazyLoader;
    }
    private List<Book> _books;
    public int AuthorId { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public List<Book> Books
    {
        get => _lazyLoader.Load(this, ref _books);
        set => _books = value;
    }
}
```

# Lazy Loading

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

- Whether you have used proxies or the `ILazyLoader` interface, lazy loading is now enabled in your application, and will take place as soon as you reference dependent entities in a relationship.

```
using(var db = new BookContext())
{
    var authors = db.Authors;
    foreach(var author in authors)
    {
        Console.WriteLine($"Name: {author.FirstName} {author.LastName}");
        foreach(var book in author.Books) // lazy loading initiated
        {
            Console.WriteLine($"{book.Title}");
        }
    }
}
```

# Lazy Loading

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

- The first query retrieves the authors, and then there are n more queries, where n represents the number of results from the first query.
- This is known as the n+1 query pattern, or probably more accurately, the n+1 problem. Flooding the database with unnecessary queries can cause performance problems.
- The same result set can be obtained with two queries using Include.

```
var authors = db.Authors.Include(a => a.Books);
foreach(var author in authors)
{
    Console.WriteLine($"Name: {author.FirstName} {author.LastName}");
    foreach(var book in author.Books)
    {
        Console.WriteLine($"{book.Title}");
    }
}
```

- The advice is not to use lazy loading unless you are certain that it is the better solution. This is why (unlike in previous versions of EF) lazy loading is not enabled by default in Entity Framework Core.

# Explicit Loading

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

- Even with lazy loading disabled (in EF 6), it is still possible to lazily load related entities, but it must be done with an explicit call. Use the Load() method to load related entities explicitly.

```
using (var context = new SchoolContext())
{
    var student = context.Students
        .Where(s => s.FirstName == "Bill")
        .FirstOrDefault<Student>();

    context.Entry(student).Reference(s => s.StudentAddress).Load(); // loads StudentAddress
    context.Entry(student).Collection(s => s.StudentCourses).Load(); // loads Courses collection
}
```

# Explicit Loading

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

- In the above example, `context.Entry(student).Reference(s => s.StudentAddress).Load()` loads the `StudentAddress` entity. The `Reference()` method is used to get an object of the specified reference navigation property and the `Load()` method loads it explicitly.
- In the same way, `context.Entry(student).Collection(s => s.Courses).Load()` loads the collection navigation property `Courses` of the `Student` entity. The `Collection()` method gets an object that represents the collection navigation property.
- The `Load()` method executes the SQL query in the database to get the data and fill up the specified reference or collection property in the memory.

# Explicit Loading

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

- You can also write LINQ-to-Entities queries to filter the related data before loading. The `Query()` method enables us to write further LINQ queries for the related entities to filter out related data.

```
using (var context = new SchoolContext())
{
    var student = context.Students
        .Where(s => s.FirstName == "Bill")
        .FirstOrDefault<Student>();

    context.Entry(student)
        .Collection(s => s.StudentCourses)
        .Query()
        .Where(sc => sc.CourseName == "Maths")
        .FirstOrDefault();
}
```

- In the above example, `.Collection(s => s.StudentCourses).Query()` allows us to write further queries for the `StudentCourses` entity.

# Explicit Loading

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

- In the above example, `context.Entry(student).Reference(s => s.StudentAddress).Load()` loads the `StudentAddress` entity. The `Reference()` method is used to get an object of the specified reference navigation property and the `Load()` method loads it explicitly.
- In the same way, `context.Entry(student).Collection(s => s.Courses).Load()` loads the collection navigation property `Courses` of the `Student` entity. The `Collection()` method gets an object that represents the collection navigation property.
- The `Load()` method executes the SQL query in the database to get the data and fill up the specified reference or collection property in the memory.

# Asynchronous Queries

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

- Asynchronous queries avoid blocking a thread while the query is executed in the database. Async queries are important for keeping a responsive UI in thick-client applications.
- They can also increase throughput in web applications where they free up the thread to service other requests in web applications.
- Entity Framework Core provides a set of async extension methods similar to the LINQ methods, which execute a query and return results. Examples include `ToListAsync()`, `ToArrayAsync()`, `SingleAsync()`.

# Asynchronous Queries

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

- There are no async versions of some LINQ operators such as `Where(...)` or `OrderBy(...)` because these methods only build up the LINQ expression tree and don't cause the query to be executed in the database.

```
public async Task<List<Blog>> GetBlogsAsync()
{
    using (var context = new BloggingContext())
    {
        return await context.Blogs.ToListAsync();
    }
}
```

# Global Query Filters

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

- Global query filters are LINQ query predicates (a boolean expression typically passed to the LINQ Where query operator) applied to Entity Types in the metadata model (usually in OnModelCreating).
- Such filters are automatically applied to any LINQ queries involving those Entity Types, including Entity Types referenced indirectly, such as through the use of Include or direct navigation property references.
- Some common applications of this feature are:
  - Soft delete: An Entity Type defines an IsDeleted property.
  - Multi-tenancy: An Entity Type defines a TenantId property.

# Global Query Filters

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

- The following example shows how to use Global Query Filters to implement soft-delete and multi-tenancy query behaviors in a simple blogging model.
- First, define the entities:

```
public class Blog
{
    private string _tenantId;

    public int BlogId { get; set; }
    public string Name { get; set; }
    public string Url { get; set; }

    public List<Post> Posts { get; set; }
}
```

```
public class Post
{
    public int PostId { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }
    public bool IsDeleted { get; set; }

    public int BlogId { get; set; }
    public Blog Blog { get; set; }
}
```

# Global Query Filters

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

- Note the declaration of a tenantId field on the Blog entity. This will be used to associate each Blog instance with a specific tenant.
- Also defined is an IsDeleted property on the Post entity type. This is used to keep track of whether a Post instance has been "soft-deleted". That is, the instance is marked as deleted without physically removing the underlying data.

# Global Query Filters

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

- Next, configure the query filters in `OnModelCreating` using the `HasQueryFilter` API.

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Blog>().Property<string>("_tenantId").HasColumnName("TenantId");

    // Configure entity filters
    modelBuilder.Entity<Blog>().HasQueryFilter(b => EF.Property<string>(b, "_tenantId") == _tenantId);
    modelBuilder.Entity<Post>().HasQueryFilter(p => !p.IsDeleted);
}
```

- The predicate expressions passed to the `HasQueryFilter` calls will now automatically be applied to any LINQ queries for those types.

# Global Query Filters

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

- Filters may be disabled for individual LINQ queries by using the `IgnoreQueryFilters()` operator.

```
blogs = db.Blogs
    .Include(b => b.Posts)
    .IgnoreQueryFilters()
    .ToList();
```

- Global query filters have the following limitations:
  - Filters can only be defined for the root Entity Type of an inheritance hierarchy.

# Tracking vs. No-Tracking Queries

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

- By default, queries that return entity types are tracking. Which means you can make changes to those entity instances and have those changes persisted by `SaveChanges()`.
- No tracking queries are useful when the results are used in a read-only scenario.
- If you don't need to update the entities retrieved from the database, then a no-tracking query should be used.
- You can swap an individual query to be no-tracking.

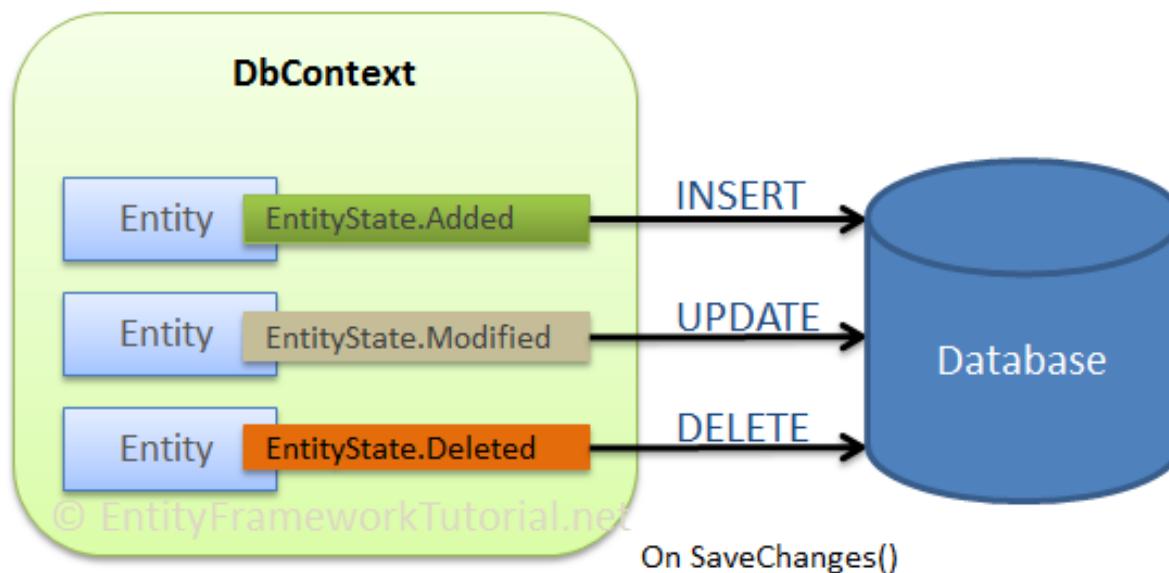
```
var blogs = context.Blogs
    .AsNoTracking()
    .ToList();
```

# Saving data

- Entity Framework Core provides different ways to add, update, or delete data in the underlying database.
- An entity contains data in its scalar property will be either inserted or updated or deleted based on its EntityState.
- There are two scenarios to save an entity data: connected (tracking) and disconnected (no-tracking).
- In the connected scenario, the same instance of DbContext is used in retrieving and saving entities, whereas this is different in the disconnected scenario.

# Saving data

- The following figure illustrates the CUD (Create, Update, Delete) operation in the connected scenario.



# Saving data

- As per the above figure, Entity Framework builds and executes INSERT, UPDATE, or DELETE statements for the entities whose EntityState is Added, Modified, or Deleted when the DbContext.SaveChanges() method is called.
- In the connected scenario, an instance of DbContext keeps track of all the entities and so it automatically sets an appropriate EntityState of each entity whenever an entity is created, modified, or deleted.

- The DbSet.Add and DbContext.Add methods add a new entity to a context (instance of DbContext) which will insert a new record in the database when you call the SaveChanges() method.

```
using (var context = new SchoolContext())
{
    var std = new Student()
    {
        FirstName = "Bill",
        LastName = "Gates"
    };
    context.Students.Add(std);

    // or
    // context.Add<Student>(std);

    context.SaveChanges();
}
```

- In the connected scenario, EF Core API keeps track of all the entities retrieved using a context.
- Therefore, when you edit entity data, EF automatically marks EntityState to Modified, which results in an updated statement in the database when you call the SaveChanges() method.

```
using (var context = new SchoolContext())
{
    var std = context.Students.First<Student>();
    std.FirstName = "Steve";
    context.SaveChanges();
}
```

# Saving data

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- Use the `DbSet.Remove()` or `DbContext.Remove` methods to delete a record in the database table.

```
using (var context = new SchoolContext())
{
    var std = context.Students.First<Student>();
    context.Students.Remove(std);

    // or
    // context.Remove<Student>(std);

    context.SaveChanges();
}
```

- Learn about saving data in the disconnected scenario in EF Core in the following link:
- <https://www.entityframeworktutorial.net/efcore/saving-data-in-disconnected-scenario-in-ef-core.aspx>

# Asynchronous Saving

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

- Asynchronous saving avoids blocking a thread while the changes are written to the database. This can be useful to avoid freezing the UI of a thick-client application.
- Asynchronous operations can also increase throughput in a web application, where the thread can be freed up to service other requests while the database operation completes.
- Entity Framework Core provides `DbContext.SaveChangesAsync()` as an asynchronous alternative to `DbContext.SaveChanges()`.

```
using (var context = new BloggingContext())
{
    var blog = new Blog { Url = url };
    context.Blogs.Add(blog);
    await context.SaveChangesAsync();
}
```

# Transactions

- Transactions allow several database operations to be processed in an atomic manner.
- If the transaction is committed, all of the operations are successfully applied to the database.
- If the transaction is rolled back, none of the operations are applied to the database.
- By default, if the database provider supports transactions, all changes in a single call to `SaveChanges()` are applied in a transaction.
  - If any of the changes fail, then the transaction is rolled back and none of the changes are applied to the database.
  - This means that `SaveChanges()` is guaranteed to either completely succeed, or leave the database unmodified if an error occurs.
- For most applications, this default behavior is sufficient. You should only manually control transactions if your application requirements deem it necessary.

- You can use the `DbContext.Database` API to begin, commit, and rollback transactions.
- Not all database providers support transactions. Some providers may throw or no-op when transaction APIs are called!
- The following example shows two `SaveChanges()` operations and a LINQ query being executed in a single transaction.
- Learn more about transactions in EF Core in the following link:
  - <https://docs.microsoft.com/en-us/ef/core/saving/transactions>

# Transactions

```
using (var context = new BloggingContext())
{
    using (var transaction = context.Database.BeginTransaction())
    {
        try
        {
            context.Blogs.Add(new Blog { Url = "http://blogs.msdn.com/dotnet" });
            context.SaveChanges();

            context.Blogs.Add(new Blog { Url = "http://blogs.msdn.com/visualstudio" });
            context.SaveChanges();

            var blogs = context.Blogs
                .OrderBy(b => b.Url)
                .ToList();

            // Commit transaction if all commands succeed, transaction will auto-rollback
            // when disposed if either commands fails
            transaction.Commit();
        }
        catch (Exception)
        {
            // TODO: Handle failure
        }
    }
}
```

- The `DbContext` in Entity Framework Core includes the `ChangeTracker` class in `Microsoft.EntityFrameworkCore.ChangeTracking` namespace which is responsible of tracking the state of each entity retrieved using the same `DbContext` instance.
- The `ChangeTracker` class in Entity Framework Core starts tracking of all the entities as soon as it is retrieved using `DbContext`, until they go out of its scope.
- EF keeps track of all the changes applied to all the entities and their properties, so that it can build and execute appropriate DML statements to the underlying data source.

- An entity at any point of time has one of the following states which are represented by the enum `Microsoft.EntityFrameworkCore.EntityState` in EF Core.
  - Added: All the new entities without key property value, added in the `DbContext` using the `Add()` or `Update()` method will be marked as Added.
  - Modified: If the value of any property of an entity is changed in the scope of the `DbContext`, then it will be marked as Modified state.
  - Deleted: If any entity is removed from the `DbContext` using the `DbContext.Remove` or `DbSet.Remove` method, then it will be marked as Deleted.
  - Unchanged: First, all the entities retrieved using direct SQL query or LINQ-to-Entities queries will have the Unchanged state.
  - Detached: All the entities which were created or retrieved out of the scope of the current `DbContext` instance, will have the Detached state. They are also called disconnected entities and are not being tracked by an existing `DbContext` instance.



EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

# KÖSZÖNÖM A FIGYELMET!

SZÉCHENYI 2020



MAGYARORSZÁG  
KORMÁNYA

Európai Unió  
Európai Strukturális  
és Beruházási Alapok



BEFETKTETÉS A JÖVŐBE



EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

# WEB API DESIGN: RESTFUL API

SZÉCHENYI 2020



MAGYARORSZÁG  
KORMÁNYA

Európai Unió  
Európai Strukturális  
és Beruházási Alapok



BEFEKETETÉS A JÖVŐBE

<https://docs.microsoft.com/en-us/azure/architecture/best-practices/api-design>

# Bevezetés

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

- A legtöbb modern webalkalmazás olyan API-kat tesz elérhetővé, amelyek segítségével az ügyfelek interakcióba léphetnek az alkalmazással.
- Egy jól megtervezett webes API-nak a következőket kell támogatnia:
  - **Platformfüggetlenség:** Bármely ügyfélnek képesnek kell lennie arra, hogy meghívja az API-t, függetlenül attól, hogy az API-t belsőleg hogyan implementálták. Ehhez szabványos protokollok használatára van szükség, és olyan mechanizmusra, amellyel a kliens és a webszolgáltatás megegyezhet a kicserélendő adatok formátumában.
  - **Szolgáltatásfejlesztés:** A webes API-nak képesnek kell lennie az ügyfélalkalmazásoktól függetlenül fejlődni. Ahogy az API fejlődik, a meglévő kliens alkalmazásoknak továbbra is változtatás nélkül kell működniük.

# REST bevezetés

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

- A webszolgáltatások tervezésének architekturális megközelítése.
- A REST egy architektúra típus hipermédián alapuló elosztott rendszerek építésére.
- A REST független minden mögöttes protokolltól, és nem feltétlenül kapcsolódik a HTTP-hez.
- A legtöbb elterjedt REST megvalósítás azonban HTTP-t használ alkalmazásprotokollként.

# A RESTful API-k tervezési elvei

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

- A REST API-k állapot nélküli kérési modellt használnak. A HTTP kéréseknek függetleneknek kell lenniük, és tetszőleges sorrendben fordulhatnak elő, így az állapotinformációk megőrzése a kérések között nem kivitelezhető. Az információ tárolásának egyetlen helye magukban az erőforrásokban van, és minden kérésnek atomi műveletnek kell lennie.
  - Ez a megszorítás lehetővé teszi, hogy a webszolgáltatások nagymértékben méretezhetőek legyenek.
  - Bármely szerver képes kezelní bármely ügyféltől érkező kérést.
- A REST API-kat a reprezentációban található hipermédia hivatkozások hajtják.

# A web API szintjei

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

- 2008-ban Leonard Richardson a következő szinteket határozta meg a webes API-khoz:
  - 0. szint: Egy Url meghatározása, és minden művelet POST kérés ehhez az URI-hez.
  - 1. szint: Külön URI-ek az egyes erőforrásokhoz.
  - 2. szint: HTTP metódusok használata az erőforrásokon végzett műveletek meghatározásához.
  - 3. szint: Hipermédia használata (HATEOAS).
- A 3. szint egy RESTful API-nak felel meg.
- A gyakorlatban sok webes API valahol a 2. szint környékén mozog.

# Az API erőforrások köré szervezése

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

- Koncentráljon az üzleti entitásokra, ezek a webes API-n keresztül legyenek elérhetőek.
  - Például egy e-kereskedelmi rendszerben az elsődleges entitások az ügyfelek és a megrendelések lehetnek. Megrendelés létrehozása a rendelési információkat tartalmazó HTTP POST kérés elküldésével érhető el. A HTTP-válasz jelzi, hogy a rendelés leadása sikeres volt-e vagy sem.
  - Ha lehetséges, az erőforrás-URI-knak főneveken (az erőforráson) és nem igéken (az erőforráson végzett műveletek) kell alapulniuk.

<https://adventure-works.com/orders> // Good

<https://adventure-works.com/create-order> // Avoid

# Az API erőforrások köré szervezése

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

- Egy erőforrásnak nem kell egyetlen fizikai adatelemen alapulnia.
  - Például egy rendelési erőforrás több táblaként valósítható meg egy relációs adatbázisban, de egyetlen entitásként jeleníthető meg az ügyfél számára.
- Kerülje az olyan API-k létrehozását, amelyek egy az egyben tükrözik az adatbázis belső szerkezetét.

# Az API erőforrások köré szervezése

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

- Az entitásokat gyakran csoportosítják (rendelések, ügyfelek).
- A csoporton belül az egyes entitások különálló erőforrások, és saját URI-val kell rendelkezniük.
- Ha HTTP GET kérésre a rendszer lekéri a csoport elemeinek listáját. A csoport minden elemének saját egyedi URI-ja is van.

<https://adventure-works.com/orders>

# Az API erőforrások köré szervezése

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

- Következetes elnevezési konvenciót alkalmazzunk az URI-ekben. Általában hasznos többes számú főnevek használata a csoportokra hivatkozó URI-k esetében. Bevált gyakorlat a csoportok és az elemek URI-jeit hierarchiába rendezni.
  - Például a /customers az ügyfelek listájához adja vissza, a /customers/5 pedig az 5-ös azonosítójú ügyfél elérési útja.
  - Ez a megközelítés segít abban, hogy a webes API intuitív legyen.
  - Ezenkívül számos webes API-keretrendszer támogatja a paraméterezett Url-eket.

# Az API erőforrások köré szervezése

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

- Bonyolultabb rendszerekben csábító lehet olyan URI-k megadása, amelyek lehetővé teszik az ügyfél számára, hogy a kapcsolatok több szintjén keresztül navigáljon.
- A komplexitás szintjét azonban nehéz lehet fenntartani, és rugalmatlan, ha az erőforrások közötti kapcsolatok a jövőben megváltoznak.
- Ehelyett használunk viszonylag egyszerű URI-ket.
- Tipp: Kerüljük a csoport/elem/csoportnál összetettebb erőforrás-URI-t.

# Az API erőforrások köré szervezése

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

- Egy másik tényező, hogy minden kérés terheli a webszervert.
- Minél több a kérés, annál nagyobb a terhelés. Ezért próbáljuk elkerülni a "csevegő" webes API-kat, amelyek nagyszámú kis erőforrást szolgálnak ki.
- Egy ilyen API megkövetelheti az ügyfélalkalmazástól, hogy több kérést küldjön az összes szükséges adat lekéréséhez.
- Ehelyett érdemes denormalizálni az adatokat, és a kapcsolódó információkat nagyobb erőforrásokká alakítani, amelyek egyetlen kéréssel lekérhetőek.
- Ezt a megközelítést azonban egyensúlyba kell hozni az ügyfélnek nem szükséges adatok lekérésével járó többletköltséggel.
- A nagy objektumok lekérése megnövelheti a kérés késleltetési idejét, és további nagyobb sávszélességet igényelhet.

# Az API erőforrások köré szervezése

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

- Kerüljük el a webes API és a mögöttes adatforrások közötti függőséget.
  - Például ha relációs adatbázist használunk, akkor az API-nak nem kell egy az egyben leképeznie az adatbázis tábláit.
- Ehelyett tekintsük a web API-t az adatbázis absztrakciójának.
  - Ha szükséges, vezessünk be egy leképezési réteget az adatbázis és a webes API közé.

# HTTP metódusok

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

- A HTTP protokoll számos metódust definiál, amelyek szemantikai jelentést rendelnek egy kéréshez. A legtöbb RESTful webes API által használt általános HTTP-metódusok a következők:
  - A GET lekéri az erőforrás reprezentációját a megadott URI-n. A válaszüzenet törzse tartalmazza a kért erőforrást.
  - A POST új erőforrást hoz létre a megadott URI-n. A kérésüzenet törzse az új erőforrás részleteit tartalmazza.
  - A PUT vagy létrehozza vagy felülírja az erőforrást a megadott URI-n. A kérésüzenet törzse meghatározza a létrehozandó vagy frissítendő erőforrást.
  - A PATCH végrehajtja az erőforrás részleges frissítését. A kérés törzse meghatározza az erőforrásra alkalmazandó változtatásokat.
  - A DELETE eltávolítja az erőforrást a megadott URI-n.

# HTTP médiatípusok

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

- A HTTP-protokollban a formátumokat médiatípusok, más néven MIME-típusok használatával határozzák meg.
- A nem bináris adatok esetében a legtöbb webes API támogatja a JSON-t és esetleg az XML-t.
- A kérés vagy válasz Content-Type fejléce határozza meg a formátumot.
  - Ha a kiszolgáló nem támogatja a médiatípust, akkor a 415-ös HTTP-állapotkódot kell visszaadnia (Unsupported Media Type).
- Az ügyfél kérése tartalmazhat egy Accept fejlécet, amely tartalmazza azon médiatípusok listáját, amelyeket az ügyfél a válaszüzenetben elfogad a szervertől.
  - Ha a szerver nem támogatja a felsorolt médiatípusok egyikét sem, akkor a 406-os HTTP-állapotkódot kell visszaadnia (Not Acceptable).

# Aszinkron műveletek

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

- Előfordulhat, hogy egy POST, PUT, PATCH vagy DELETE művelet hosszabb feldolgozást igényel.
  - Ha megvárnánk a művelet befejezését, az elfogadhatatlan hosszú késleltetést okozna.
- Fontoljuk meg a művelet aszinkronizálását.
- Adjuk vissza a 202-es HTTP-állapotkódot (Accepted), jelezve, hogy a kérést elfogadtuk feldolgozásra, de még nem fejeződött be.
- Hozzunk létre egy külön végpontot, amelyen a feldolgozás során lekérdezhető a feldolgozás állapota.

# Adatok szűrése és lapozása

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

- Az erőforrások egyetlen URI-n keresztüli feltárása ahhoz vezethet, hogy az alkalmazások nagy mennyiségű adatot kérnek le, amikor az információknak csak egy részhalmazára van szükség.
  - Az API lehetővé tegye egy szűrő feltétel átadását az URI-ben (paraméter). Az API ezután feldolgozza ezt a paramétert, és a szűrt eredményt adja vissza.
- Az erőforrásokat listázó GET-kérések potenciálisan nagyszámú elemet adhatnak vissza. Meg kell tervezni egy olyan API-t, amely korlátozza az egyetlen kérés által visszaküldött adatok mennyiségét.
  - Fontoljuk meg olyan lekérdezési paraméterek támogatását, amelyek meghatározzák a lekérhető elemek maximális számát és a lista lapozását.

# Adatok szűrése és lapozása

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

- A visszaadott elemek számának maxmilásával kivédhetőek a DoS támadások.
- Az ügyfélalkalmazások segítése érdekében a lapozott adatokat visszaadó GET-kéréseknek tartalmazniuk kell valamilyen metaadatokat is, amelyek jelzik a gyűjteményben elérhető erőforrások teljes számát.
- Hasonló stratégiát használhatunk az adatok lekéréskor történő rendezésére, ha megadunk egy rendezési paramétert, amely egy mezőnevet vesz fel értékként, például /orders?sort=ProductID.

# Részleges válaszok támogatása nagy bináris erőforrások esetén

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

- Egy erőforrás tartalmazhat nagy bináris mezőket, például fájlokat vagy képeket.
- A megbízhatatlan kapcsolatok okozta problémák leküzdése és a válaszidő javítása érdekében fontoljuk meg az ilyen erőforrások darabokban történő lekérését.
  - Ehhez a webes API-nak támogatnia kell az Accept-Ranges fejlécet a nagy erőforrásokhoz tartozó GET-kérésekknél.
  - Ez a fejléc azt jelzi, hogy a GET művelet támogatja a részleges kéréseket.
- Az ügyfélalkalmazás olyan GET-kéréseket küldhet be, amelyek az erőforrások egy bájttartományként meghatározott részhalmazát adják vissza.

# Részleges válaszok támogatása nagy bináris erőforrások esetén

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

- Ezenkívül fontoljuk meg a HTTP HEAD kérések megvalósítását ezekhez az erőforrásokhoz.
  - A HEAD kérés hasonló a GET kéréshez, azzal a különbséggel, hogy csak az erőforrást leíró HTTP-fejléceket adja vissza, üres üzenettörzsel.
- Az ügyfélalkalmazások HEAD kérést adhatnak ki annak meghatározására, hogy le kell-e kérni az erőforrást részleges GET kérések használatával.  
Például:

```
HEAD https://adventure-works.com/products/10?fields=productImage HTTP/1.1
```

# Részleges válaszok támogatása nagy bináris erőforrások esetén

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

- Egy példa a válaszra:

HTTP/1.1 200 OK

Accept-Ranges: bytes  
Content-Type: image/jpeg  
Content-Length: 4580

- A Content-Length fejléc az erőforrás teljes méretét adja meg, az Accept-Ranges fejléc pedig azt jelzi, hogy a megfelelő GET művelet támogatja a részleges eredményeket.
- Az ügyfélalkalmazás felhasználhatja ezeket az információkat a kép kisebb darabokban való lekérésére. Az első kérés lekéri az első 2500 bájtot a Range fejléc használatával:

GET <https://adventure-works.com/products/10?fields=productImage> HTTP/1.1  
Range: bytes=0-2499

# Részleges válaszok támogatása nagy bináris erőforrások esetén

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

- A válaszüzenet a 206-os HTTP-állapotkóddal jelzi, hogy ez egy részleges válasz.
- A Content-Length fejléc az üzenettörzsben visszaadott bájtok tényleges számát határozza meg (nem az erőforrás méretét), a Content-Range fejléc pedig azt, hogy ez az erőforrás melyik része (0-2499 bájt a 4580-ból):

HTTP/1.1 206 Partial Content

Accept-Ranges: bytes  
Content-Type: image/jpeg  
Content-Length: 2500  
Content-Range: bytes 0-2499/4580

- Az ügyfélalkalmazástól érkező későbbi kérés lekérheti az erőforrás fennmaradó részét.

# További információk

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

- <https://docs.microsoft.com/en-us/azure/architecture/best-practices/api-design>
- <https://restfulapi.net/>



EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

# KÖSZÖNÖM A FIGYELMET!

SZÉCHENYI 2020



MAGYARORSZÁG  
KORMÁNYA

Európai Unió  
Európai Strukturális  
és Beruházási Alapok



BEEFEKTETÉS A JÖVŐBE

# A rendszertervezés korszerű módszerei

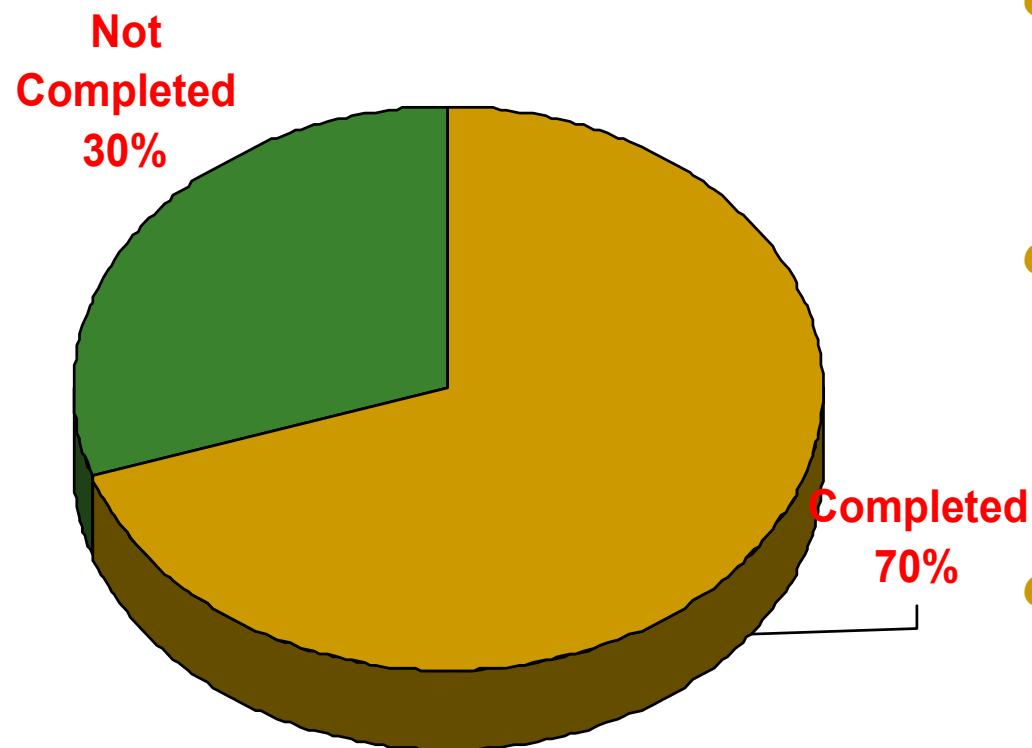
---

Rational Unified Process

# Fejlesztési módszertanok

- Szoftverfejlesztési módszertan:  
hogyan célszerű szoftver rendszereket
  - Építeni
  - Telepíteni
  - Karbantartani
- Pl.
  - Vízesés modell
  - Iteratív modellek

# A szoftverfejlesztés kockázatos...

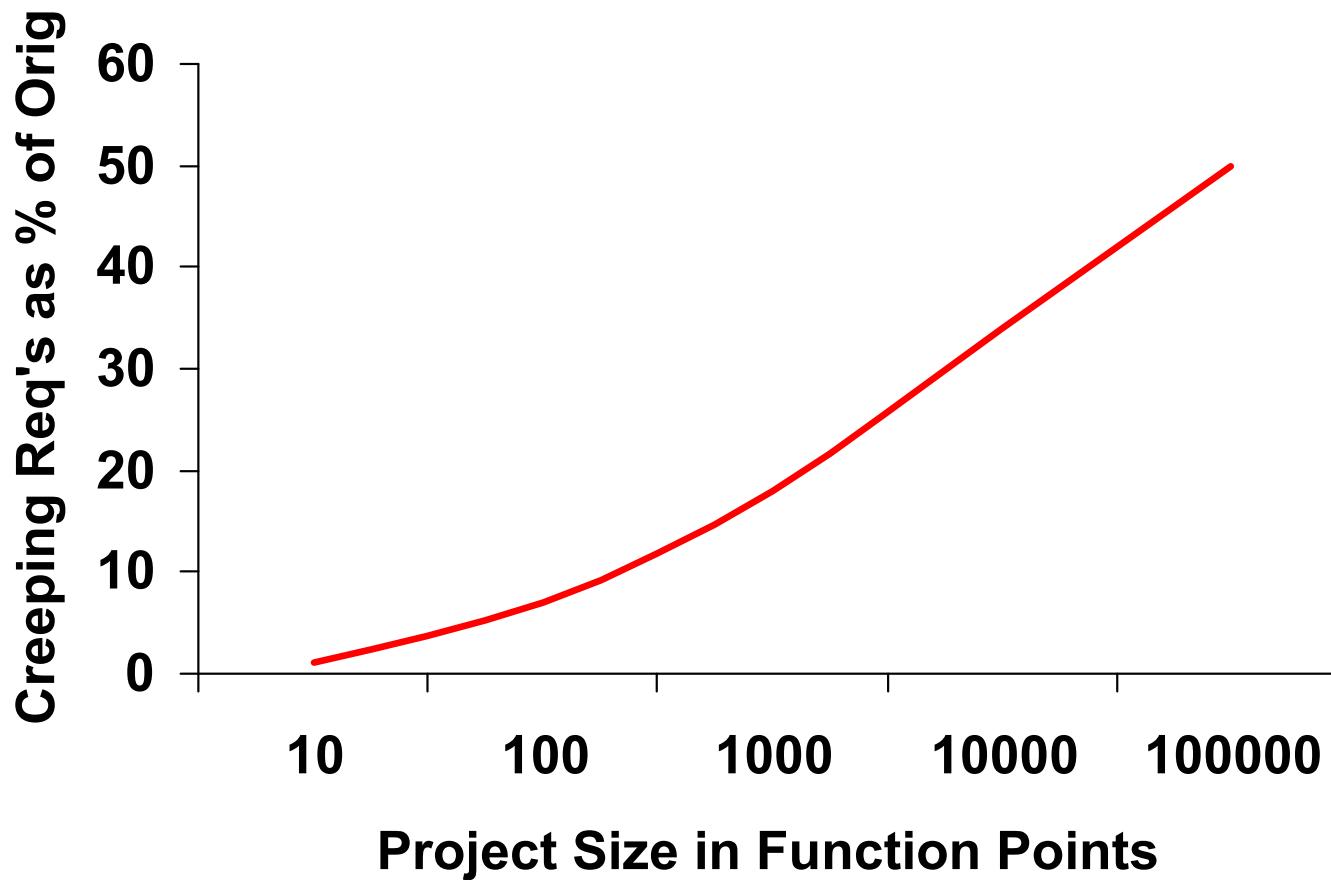


Forrás: Standish Report, 1994

- Vízesés modell alapján...
- Az esetek 53%-ában a költség a tervezett 200%-a lett
- 1995-ben becslések szerint 81 milliárd dollárt költöttek megbukott szoftverprojektekre.

# Tévhít #1:

## A követelmények elég pontosak



- Applied Software Measurement, Capers Jones, 1997. Based on 6,700 systems.

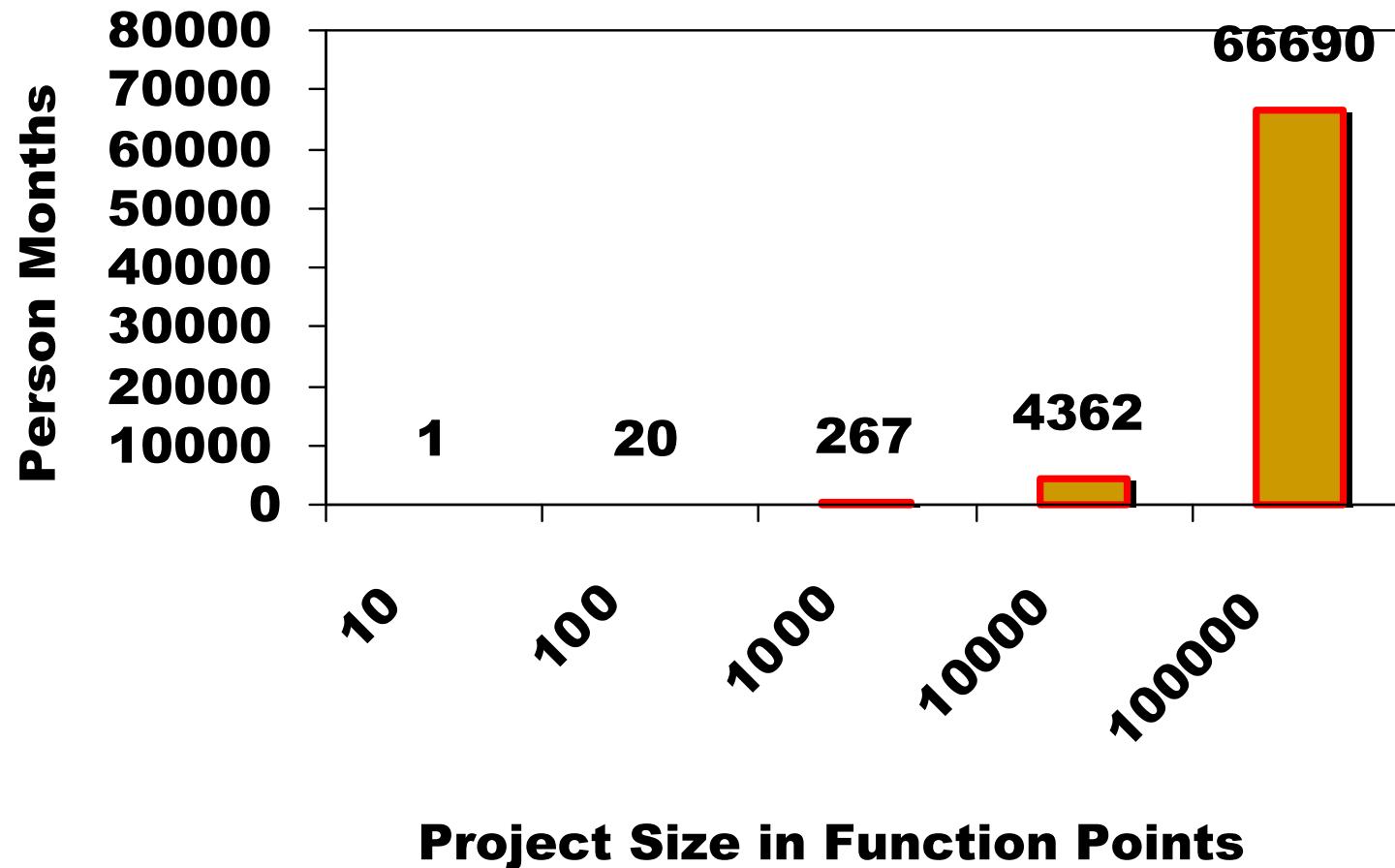
## Tévhít #2: A követelmények stabilak

- A piac **folyamatosan** változik
- A technológia változik
- A részvényszek céljai változnak

## Tévhít #3: A tervezés a programozás elkezdése előtt befejezhető

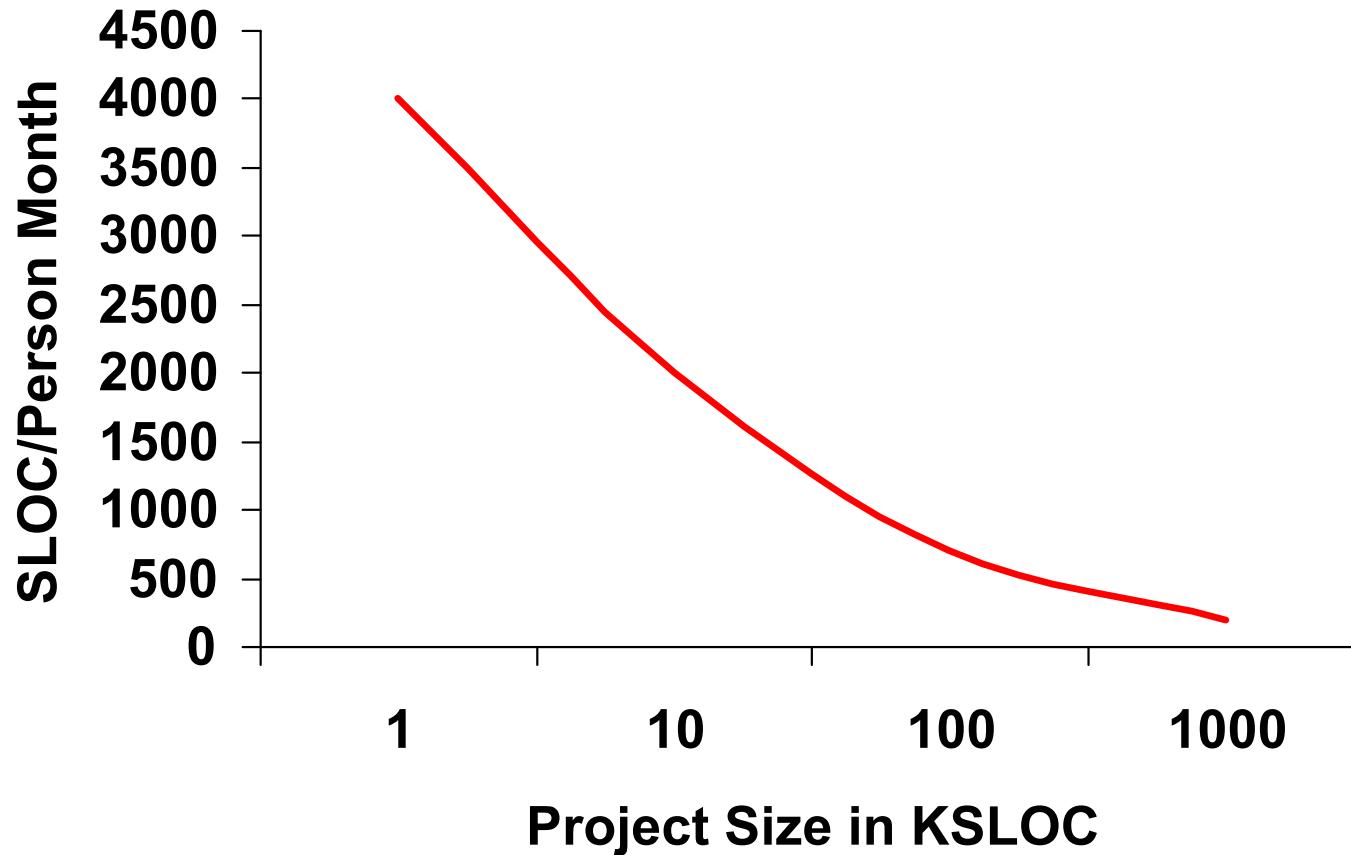
- A követelmények nem teljesek és változnak.
- Túl sok változó, ismeretlen, újdonság, ...
- A teljes specifikáció olyan részletes lenne, mint a kód maga.
- Szoftvert készíteni nehéz:
  - Discover Magazine, 1999: A szoftver az emberiség által készített legbonyolultabb „gép”.

# A nagy lépések veszélyei: ráfordítás



Applied Software Measurement, Capers Jones, 1997. Based on 6,700 systems.

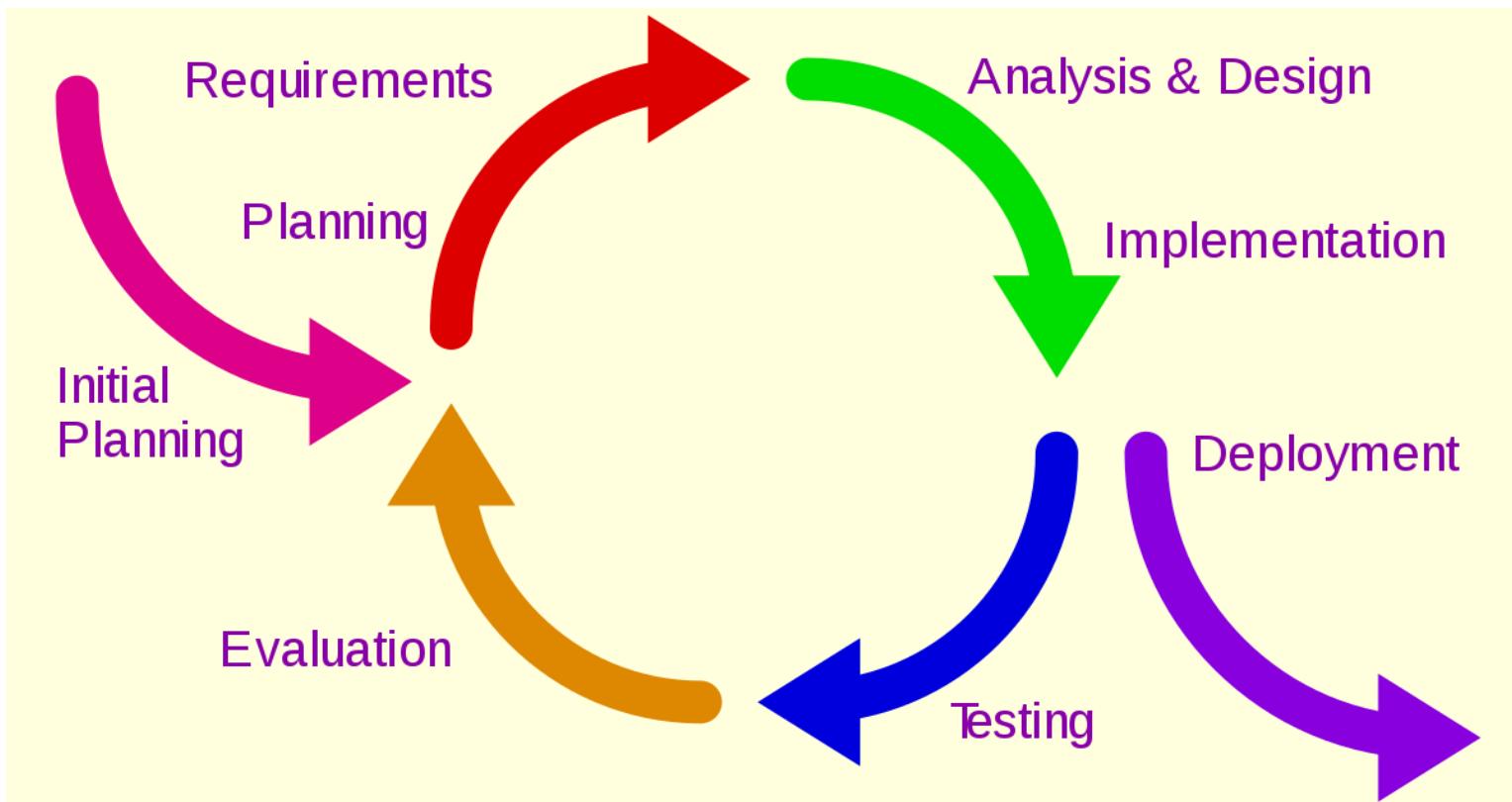
# A nagy lépések veszélyei: produktivitás



Measures For Excellence, Putnam, 1992. Based on 1,600 systems.

# Megoldás?

- Iteratív fejlesztés...



# Ismétlés: a vízesés modell

- Alapelv:

- Követelmények tisztázása és rögzítése (befagyasztása)
- A követelmények alapján a rendszer tervezése
- A tervezés alapján implementálás

# Unified Process

- Korábbi iteratív módszertanok: spirális, evolúciós
- A UP a '90-es évek végén keletkezett, OO rendszerek fejlesztésére szolgáló módszertan
- A Rational Unified Process ennek továbbfejlesztett, finomított változata
- Széles körben használt, sikeres

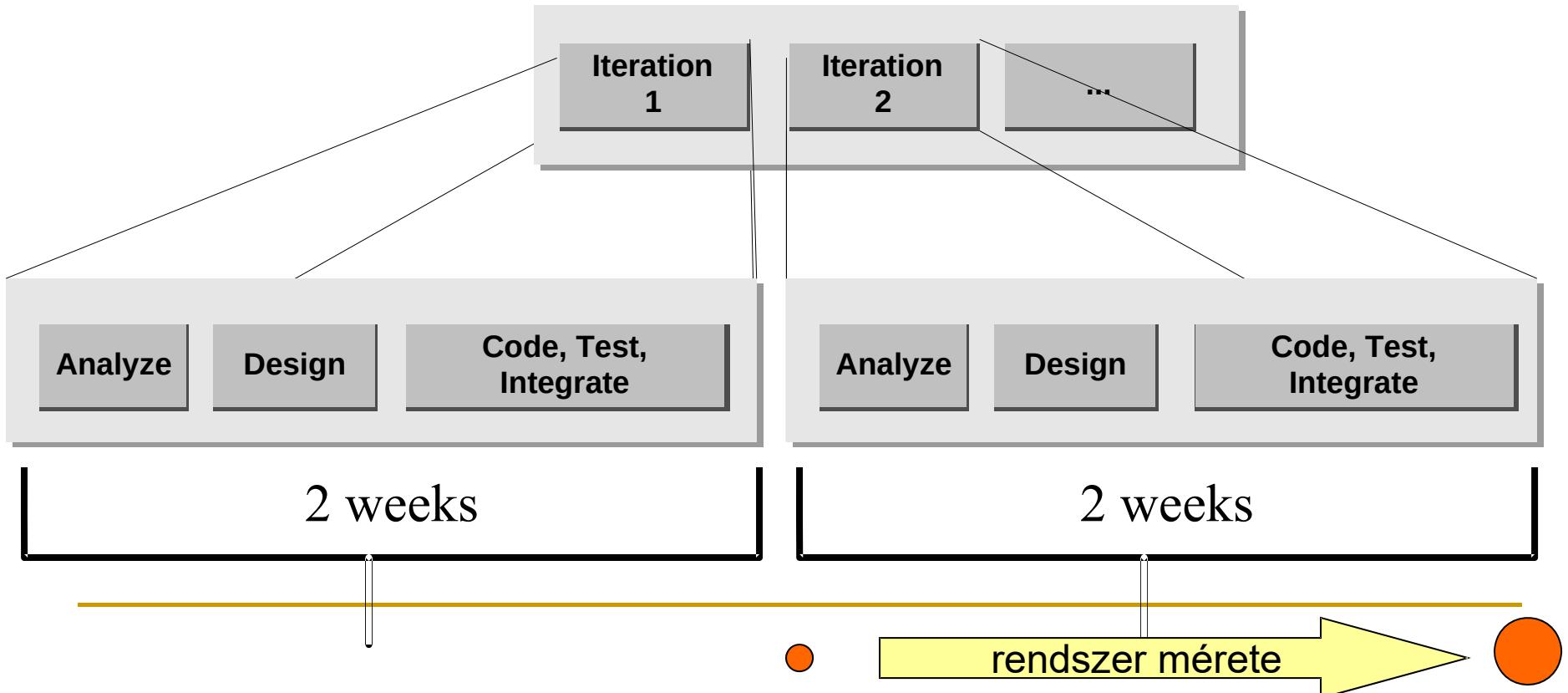
# Iteráció

- Iteráció = mini-projekt
  - rövid (általában 2-4 hét)
  - előre meghatározott hosszúságú
- A fejlesztés ilyen iterációk sorozata
- A rendszer fokozatosan nő és finomodik
- Fontos a
  - visszacsatolás és az
  - adaptáció
- Ezek az **iteratív inkrementális fejlesztés** alapelvei

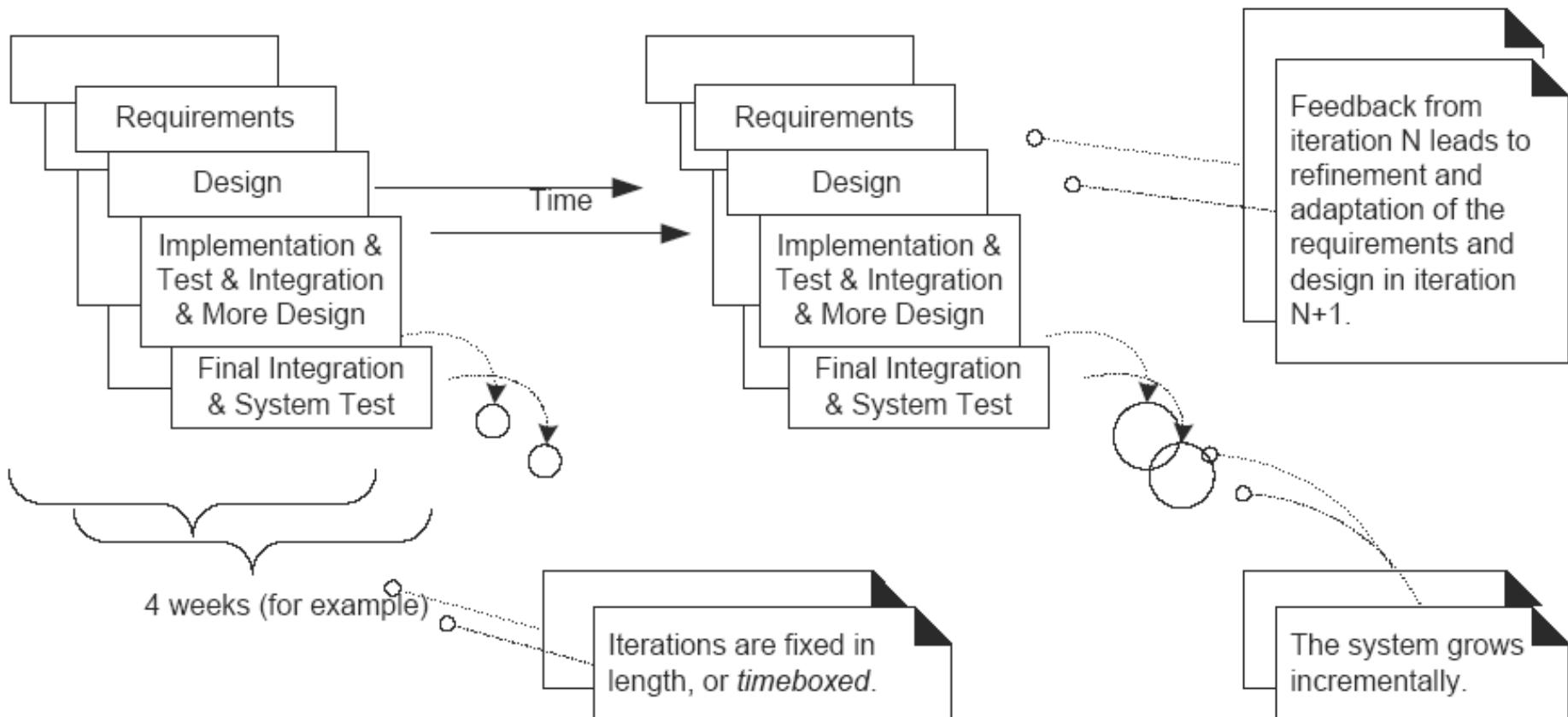
# Iteratív fejlesztés

- Kulcsszavak:

- Kis lépések, visszacsatolás, pontosítás/átdolgozás
- Iteratív, inkrementális, időkorlátozott.



# Iteratív inkrementális fejlesztés



# Az iteráció eredménye

- Végrehajtható, de még nem teljes rendszer
- A végső termék része
- Nem gyártásra kész
- Még nem telepíthető (akár 10-15 iteráció is szükséges lehet ehhez)
- De **nem** kísérleti rendszer, **nem** prototípus, **nem** eldobandó!
- Általában újabb követelmények megvalósítása minden iterációban, de elképzelhető „javító” iteráció is (pl. sebesség növelése érdekében)

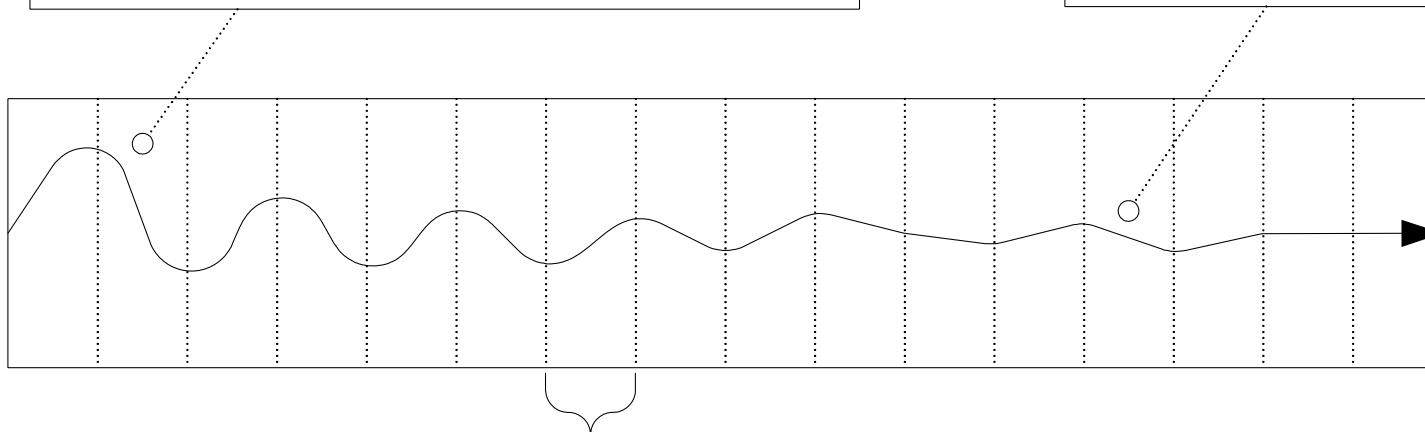
# Változások elfogadása

- Tipikus mondat a fejlesztés során:
- „Igen, ilyesmire gondoltam, **de ...**”
- Sajnálatos tény: követelmények változnak
- UP filozófia:
  - Ne harcoljunk ellene (pl. komoly specifikációs fázisban mégiscsak próbáljuk meg előre kitalálni...),
  - hanem inkább építsük be a fejlesztési eljárásba.
  - Felépít – Visszacsatol – Adaptál ciklusok

# Visszacsatolás és adaptáció

Early iterations are farther from the "true path" of the system. Via feedback and adaptation, the system converges towards the most appropriate requirements and design.

In late iterations, a significant change in requirements is rare, but can occur. Such late changes may give an organization a competitive business advantage.



# Az iteratív fejlesztés előnyei

- A nagy kockázatú elemek korai ellenőrzése (technikai problémák, követelmények, használhatóság, stb...)
- Korai, jól látható előrehaladás a fejlesztésben
- Korai felhasználói visszacsatolás, ami az igényeknek jobban megfelelő rendszerhez vezet
- Kezelhető komplexitás: a fejlesztőket nem terheli le az „analízis paralízis”, vagy egyéb hosszú és bonyolult fejlesztési lépések
- Az iterációkon belül nyert tapasztalatok felhasználhatók a fejlesztési folyamat javítására

# Az iterációk időkorlátai

- Timeboxing
- Az iterációk javasolt hossza: **2-6 hét**
  - (nagyon kivételes mega-projektnél alkalmaztak már 6 hónapos iterációkat is, de ez a kivétel)
- 2 hétnél rövidebb: rövid idő jelentősebb fejlesztésre és a visszacsatolásra
- 6-8 hétnél hosszabb: túl nagy komplexitás, a visszacsatolás késik
- Mi történik, ha nem lehet tartani az előre meghatározott határidőt?
  - nem módosul a határidő, inkább
  - néhány feladat kivétele az iterációból

# Az eddigiek fő üzenete:

- UP központi gondolata:
  - **rövid**
  - **időkorlátos**
  - **iteratív**
  - **adaptív fejlesztés**
- Fontos elv még az OO alapelvek használata:
  - **OOA**
  - **OOD**
  - **OOP**

# A UP ajánlásai (best practices)

A RUP 6 fő ajánlása:

1. Develop software iteratively
2. Manage requirements
3. Use component-based architectures
4. Visually model software
5. Verify software quality
6. Control changes to software

# 1. Develop Software Iteratively

- Given today's sophisticated software systems, it is not possible to sequentially first define the entire problem, design the entire solution, build the software and then test the product at the end. **An iterative approach is required that allows an increasing understanding of the problem through successive refinements, and to incrementally grow an effective solution over multiple iterations.**

The Rational Unified Process supports an iterative approach to development that addresses the highest risk items at every stage in the lifecycle, significantly reducing a project's risk profile. This iterative approach helps you attack risk through demonstrable progress – frequent, executable releases that enable continuous end user involvement and feedback. Because each iteration ends with an executable release, the development team stays focused on producing results, and frequent status checks help ensure that the project stays on schedule. An iterative approach also makes it easier to accommodate tactical changes in requirements, features or schedule.

---

Forrás:

**Rational Unified Process - Best Practices for Software Development Teams**  
A Rational Software Corporation White Paper

## 2. Manage Requirements

- The Rational Unified Process describes how to elicit, organize, and **document** required functionality and constraints; **track** and document tradeoffs and decisions; and easily capture and communicate **business requirements**. The notions of **use case and scenarios** proscribed in the process has proven to be an excellent way to capture functional requirements and to ensure that these drive the design, implementation and testing of software, making it more likely that the final system fulfills the end user needs. They provide coherent and traceable threads through both the development and the delivered system.

Forrás:

Rational Unified Process - *Best Practices for Software Development Teams*  
A Rational Software Corporation White Paper

### 3. Use Component-based Architectures

- The process focuses on **early development and baselining of a robust executable architecture, prior to committing resources for full-scale development**. It describes how to design a resilient architecture that is flexible, accommodates change, is intuitively understandable, and promotes more effective software reuse. The Rational Unified Process supports *component-based software development*. Components are non-trivial modules, subsystems that fulfill a clear function. The Rational Unified Process provides a systematic approach to defining an architecture using new and existing components. These are assembled in a well-defined architecture, either ad hoc, or in a component infrastructure such as the Internet, CORBA, and COM, for which an industry of reusable components is emerging.

## 4. Visually Model Software

- The process shows you how to visually model software **to capture the structure and behavior of architectures and components**. This allows you to hide the details and write code using “graphical building blocks.” **Visual abstractions help you communicate different aspects of your software**; see how the elements of the system fit together; make sure that the building blocks are consistent with your code; maintain consistency between a design and its implementation; and promote unambiguous communication. The industry-standard Unified Modeling Language (UML), created by Rational Software, is the foundation for successful visual modeling.

---

Forrás:

**Rational Unified Process - Best Practices for Software Development Teams**  
A Rational Software Corporation White Paper

# 5. Verify Software Quality

- Poor application performance and poor reliability are common factors which dramatically inhibit the acceptability of today's software applications. Hence, quality should be reviewed with respect to the requirements based on reliability, functionality, application performance and system performance. The Rational Unified Process assists you in the planning, design, implementation, execution, and evaluation of these test types. **Quality assessment** is built into the process, in all activities, involving all participants, using objective measurements and criteria, and **not treated as an afterthought or a separate activity performed by a separate group.**

---

Forrás:

Rational Unified Process - *Best Practices for Software Development Teams*  
A Rational Software Corporation White Paper

# 6. Control Changes to Software

- The ability to **manage change** – making certain that each change is acceptable, and being able to track changes – is essential in an environment in which change is inevitable. The process describes how to control, track and monitor changes to enable successful iterative development. It also guides you in how to establish secure workspaces for each developer by providing isolation from changes made in other workspaces and by controlling changes of all software artifacts (e.g., models, code, documents, etc.). And it brings a team together to work as a single unit by describing how to automate integration and build management.

---

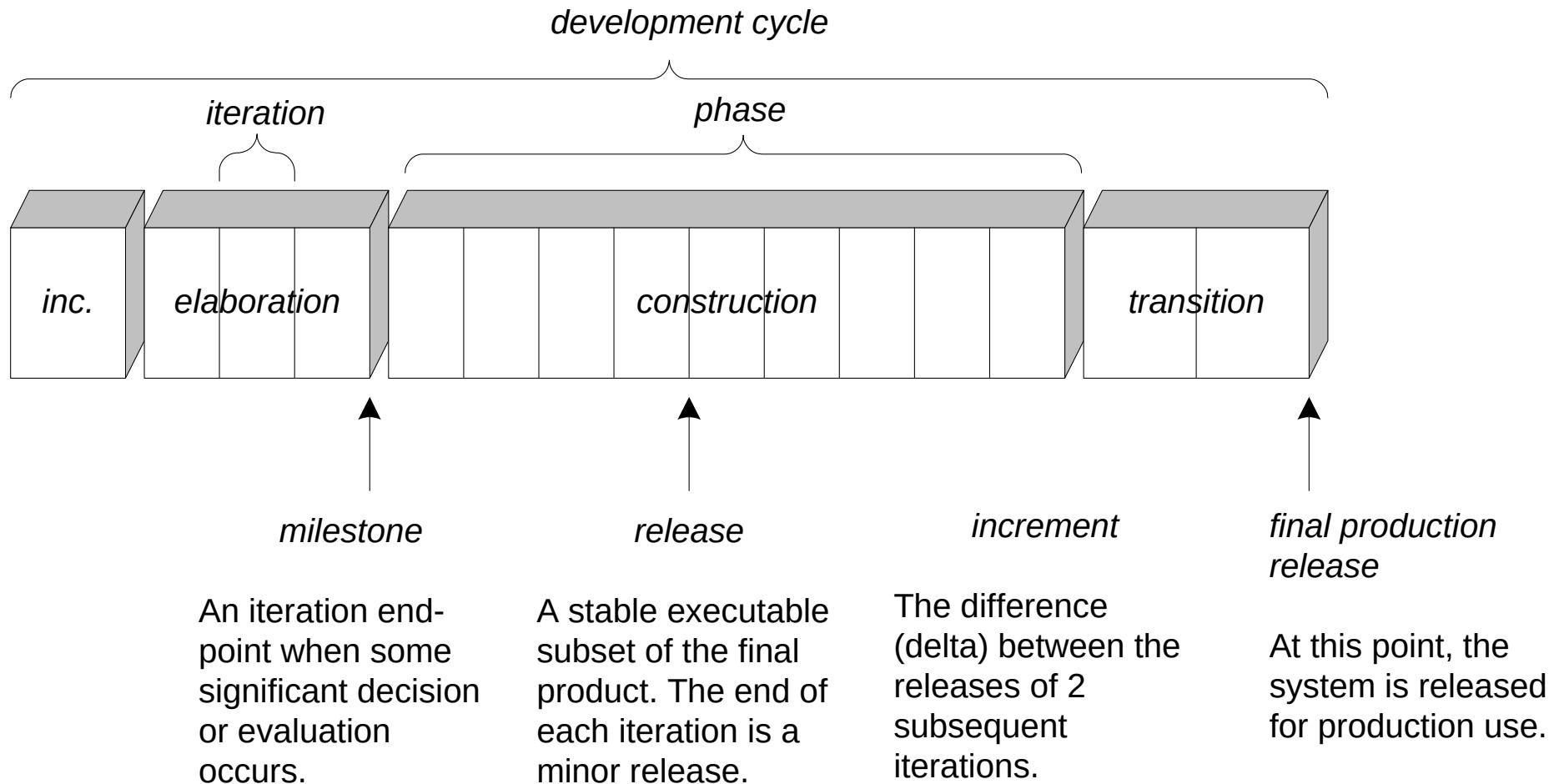
Forrás:

**Rational Unified Process - Best Practices for Software Development Teams**  
A Rational Software Corporation White Paper

# A UP fázisai

1. **Inception**— approximate vision, business case, scope, vague estimates.
2. **Elaboration**—refined vision, iterative implementation of the core architecture, resolution of high risks, identification of most requirements and scope, more realistic estimates.
3. **Construction**—iterative implementation of the remaining lower risk and easier elements, and preparation for deployment.
4. **Transition**—beta tests, deployment.

# Fázisok és egyéb ütemezési fogalmak



# 1. Inception

Inception      Elaboration      Construction      Transition

Eredmény:

- A vision document: a general vision of the core project's requirements, key features, and main constraints.
- An initial use-case model (10%-20% complete).
- An initial project glossary (may optionally be partially expressed as a domain model).
- An initial business case, which includes business context, success criteria (revenue projection, market recognition, and so on), and financial forecast.
- An initial risk assessment.
- A project plan, showing phases and iterations.
- A business model, if necessary.
- One or several prototypes.

## 2. Elaboration

Inception

Elaboration

Construction

Transition

Eredmény:

- A use-case model (at least 80% complete) — all use cases and actors have been identified, and most use-case descriptions have been developed.
- Supplementary requirements capturing the non functional requirements and any requirements that are not associated with a specific use case.
- A Software Architecture Description.
- An executable architectural prototype.
- A revised risk list and a revised business case.
- A development plan for the overall project, including the coarse-grained project plan, showing iterations" and evaluation criteria for each iteration.
- An updated development case specifying the process to be used.
- A preliminary user manual (optional).

# 3. Construction

Inception

Elaboration

Construction

Transition

Eredmény:

- The product ready to put in hands of its end-users.
- At minimum, it consists of:
  - The software product integrated on the adequate platforms.
  - The user manuals.
  - A description of the current release.

---

Forrás:

**Rational Unified Process - Best Practices for Software Development Teams**

A Rational Software Corporation White Paper

# 4. Transition

Inception

Elaboration

Construction

Transition

Tipikus tevékenységek:

- “beta testing” to validate the new system against user expectations
- parallel operation with a legacy system that it is replacing
- conversion of operational databases
- training of users and maintainers
- roll-out the product to the marketing, distribution, and sales teams

---

Forrás:

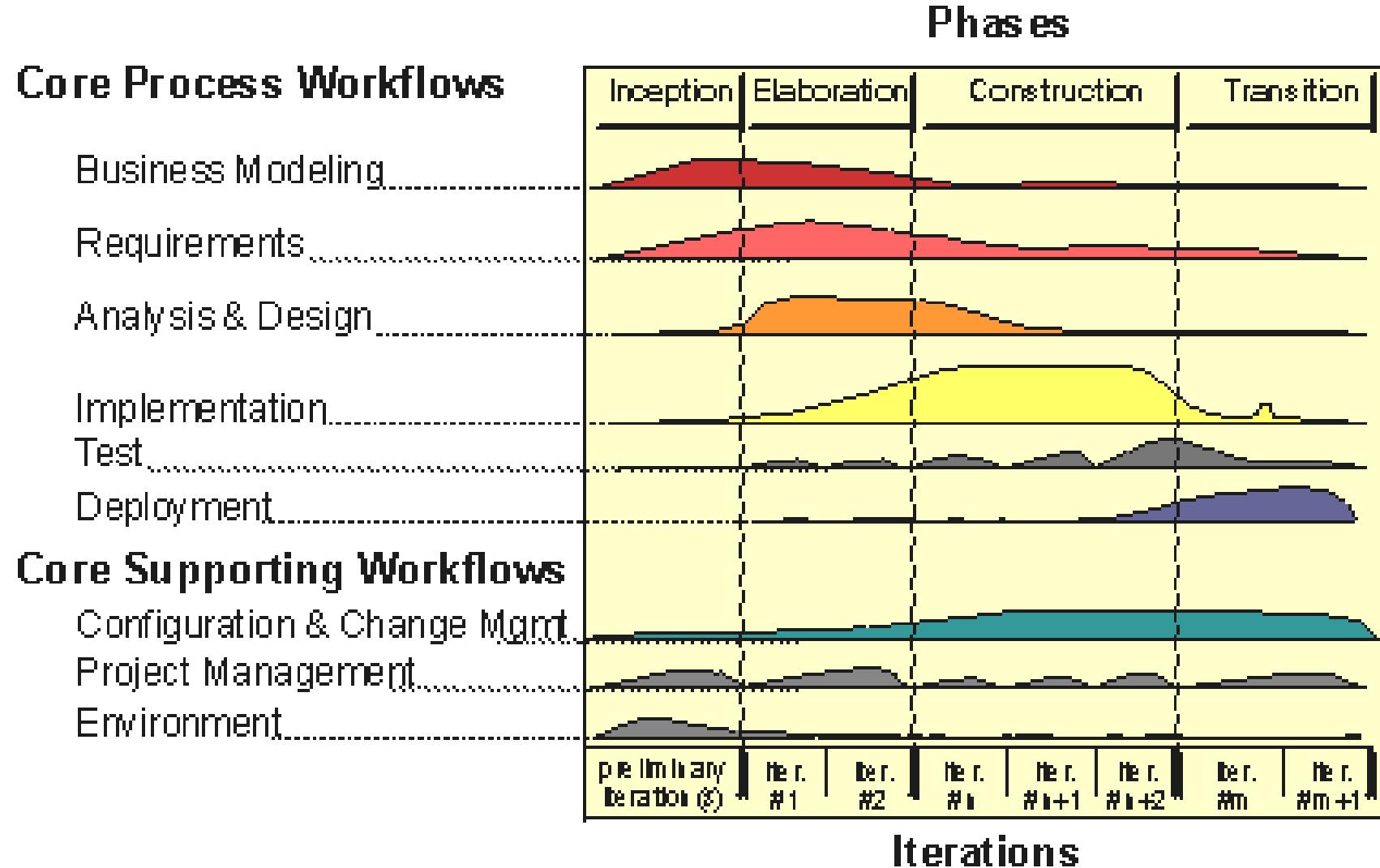
Rational Unified Process - *Best Practices for Software Development Teams*

A Rational Software Corporation White Paper

# A UP diszciplinái

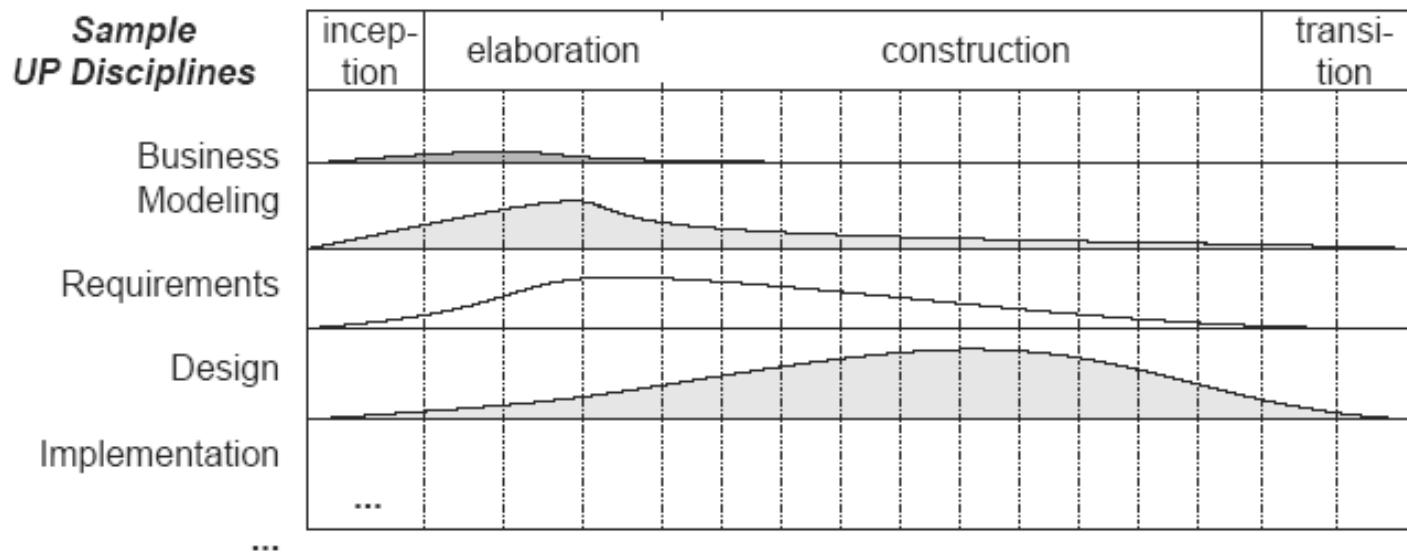
- Régebben munkafolyamnak (workflow) nevezték
- Hat fő diszciplina
  - 1. **Business modeling**
  - 2. **Requirements**
  - 3. **Analysis & Design**
  - 4. **Implementation**
  - 5. **Test**
  - 6. **Deployment**
- Három kiszolgáló diszciplina
  - 1. Project Management
  - 2. Configuration and Change Management
  - 3. Environment

# A UP diszciplinák és a fázisok



Figyelem: a fázisok nem azonosak az iterációkkal!

# A diszciplinák arányai



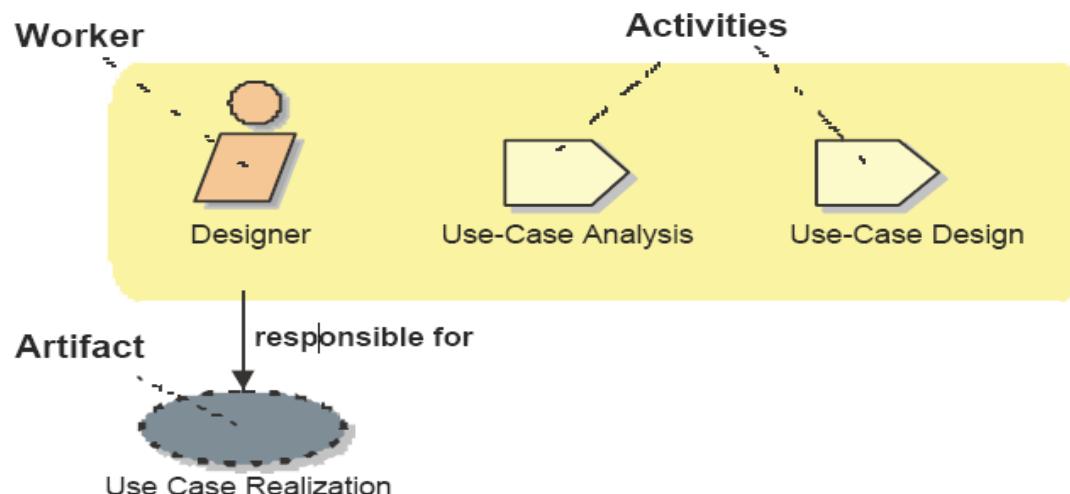
The relative effort in disciplines shifts across the phases.

This example is suggestive, not literal.

- Az egyes diszciplinák közötti arány folyamatosan változik a fázisok (és iterációk) alatt
- Eleinte pl. magasabb a követelménytervezés aránya, később a tervezés, majd az implementálás dominál

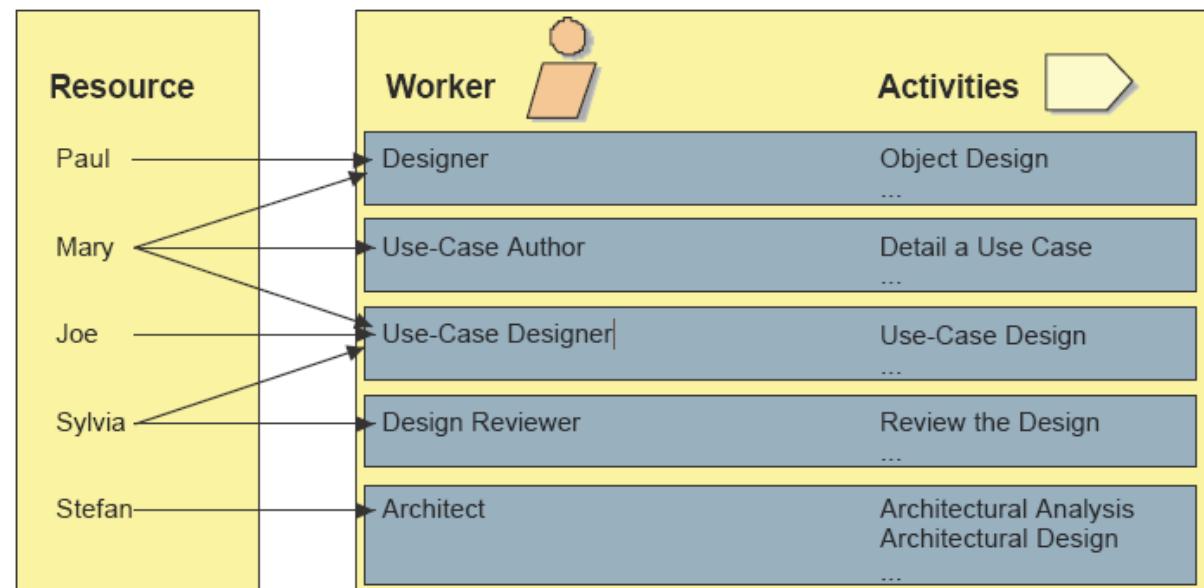
# Egyéb RUP fogalmak

- Mukavégző: „Workers, the ‘who’”
- Tevékenység: „Activities, the ‘how’”
- Termékek: „Artifacts, the ‘what’”
- Munkafolyam (diszciplina): „Workflows, the ‘when’” (erről beszéltünk)



# A munkavégző (worker)

- A munkavégző egy személy viselkedését, szerepét határozza meg („sapka”).
- Egy személy több sapkát is hordhat



# Tevékenységek (activities)

- A munkavégző számára egy munkaegység.
- Világos célja van (pl. készítsen el egy modellt)
- Néhány óra – néhány nap
- Általában egy munkavégzőt érint
- Max. néhány terméket érint
- Példák:
  - Egy iteráció megtervezése (worker: projekt manager)
  - Használati esetek kidolgozása (worker: system analyst)
  - Teljesítmény tesztelése (worker: Performance Tester)

# Termékek (artifacts)

- Egy információs egység, amit a folyamat során létrehozunk, módosítunk, használunk.
- Példák:
  - Modellek (pl. use case)
  - A modell egy eleme (pl. egy osztály, alrendszer)
  - Dokumentum (pl. Szoftver architektúra dokumentum)
  - Forráskód
  - Futtatható kód

# Metodikák jellemzése

- Egy fejlesztési metodika lehet
  - Nehéz vagy könnyű
    - Nehéz: sok termék, bürokratikus atmoszféra, szigorú ellenőrzés, részletes tervezés
    - Könnyű: ennek ellentéte
  - Prediktív vagy adaptív
    - Prediktív: hosszú időre részletesen előre próbál tervezni, általában vízesés-modell alapon
    - Adaptív: változások beépítése, azokhoz adaptálódás
  - **Agilis:** könnyű és adaptív

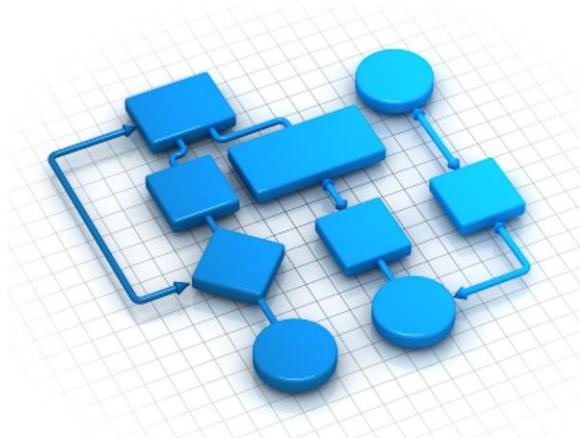
## „Agilis” UP

- Lehetőleg kevés termék: „keep it simple”
- Iteratív követelményfeltárás, tervezés és implementáció, visszacsatoláson alapulva.
- Nincs a projektnek előre meghatározott részletes terve.
  - Phase Plan
  - Iteration Plan

# A rendszerfejlesztés korszerű módszerei



## Scrum



Pannon Egyetem  
2021

# Tartalom



- Mi az a Scrum?
- Scrum csapat jellemző
- Scrum események
- Scrum munkaanyagok



# Tartalom



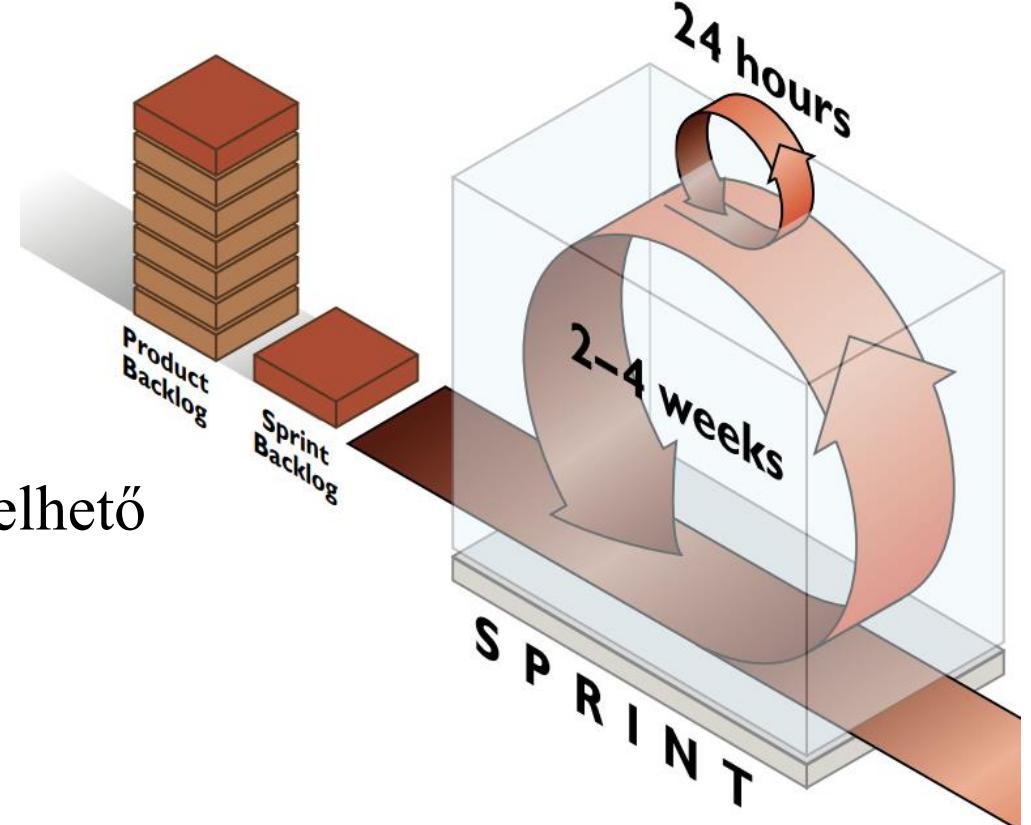
- **Mi az a Scrum?**
- Scrum csapat jellemző
- Scrum események
- Scrum munkaanyagok



# Mi az a Scrum?



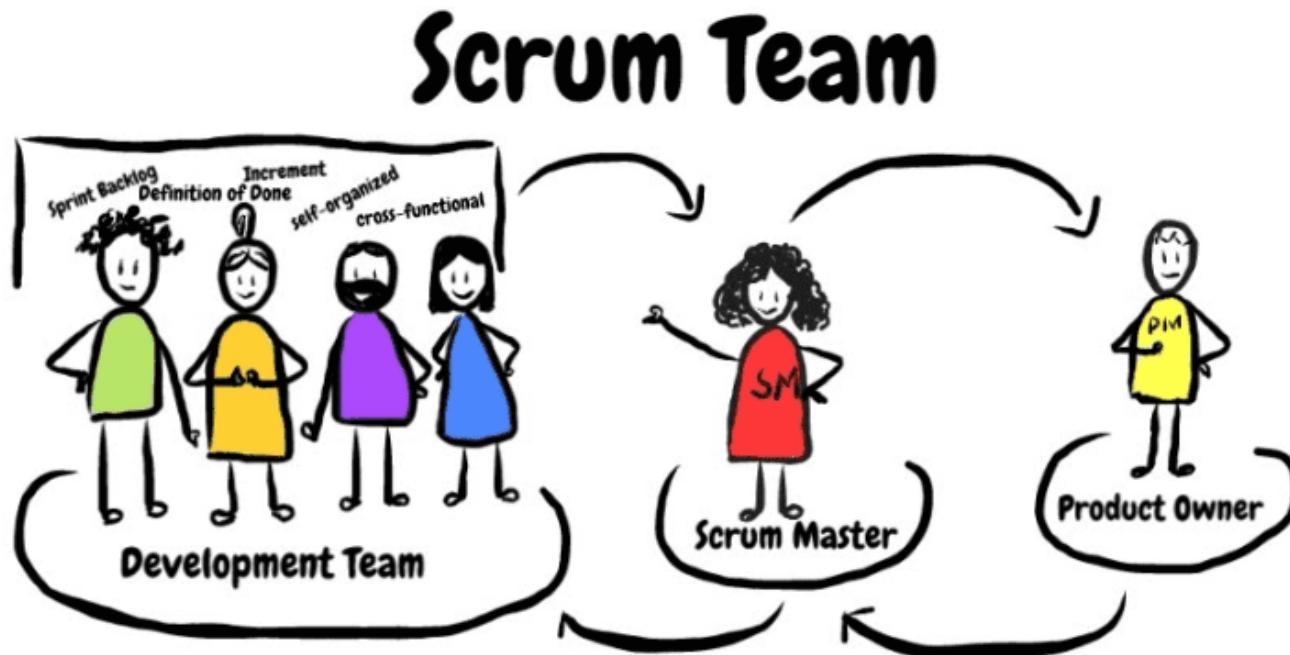
- **Scrum:** Egy olyan keretrendszer, melynek segítségével emberek komplex problémákat tudnak adaptív módon kezelní úgy, hogy közben termelékenyen és kreatívan szállítják le a lehető legértékesebb termékeket.
- Jellemzői:
  - Egyszerű
  - Könnyen érthető
  - Rendkívül nehezen művelhető mesteri szinten



# Mi az a Scrum?



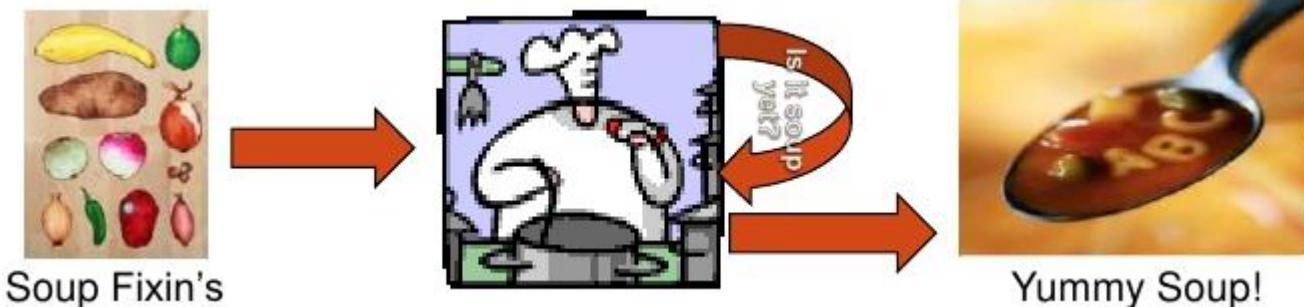
- A Scrum keretrendszer a Scrum Csapatokból, valamint a hozzájuk rendelt szerepekből, eseményekből, munkaanyagokból (artifacts) és szabályokból áll.



# Mi az a Scrum?



- A Scrum a **tapasztaláson alapuló** folyamatellenőrzési elméletben, vagy más néven empirizmuson alapul.
- Az empirizmus azt állítja, hogy a tudás a tapasztalatokból és az adott ismereteken alapuló döntésekből ered.
- A Scrum egy iteratív (ismétlődő), inkrementális megközelítést alkalmaz a kiszámíthatóság optimalizálása és a kockázat kézben tartása érdekében.



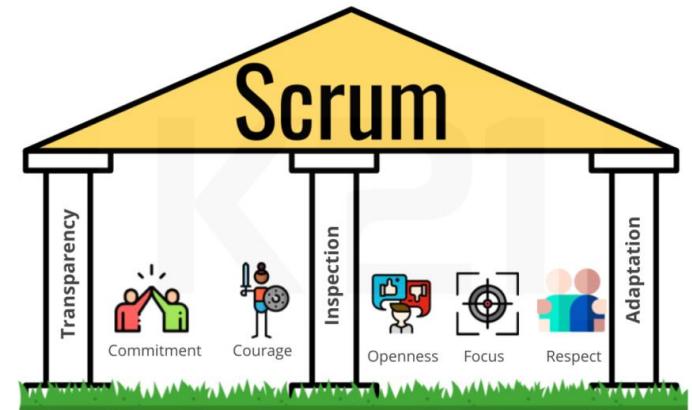
- Az empirikus folyamatellenőrzés megvalósítása három pilléren nyugszik: **transzparencia, ellenőrzés és korrekció**

# Mi az a Scrum?



## 1) Transzparencia (Transparency)

- A folyamat lényeges nézőpontjainak láthatónak kell lenni azok számára, akik felelősek az eredményért.
- A transzparencia elve megköveteli, hogy ezeket a nézőpontokat egy közös szabvány szerint határozzák meg, hogy a minden résztvevő ugyanazzal az értelmezéssel rendelkezzen.



## 2) Ellenőrzés (Inspection)

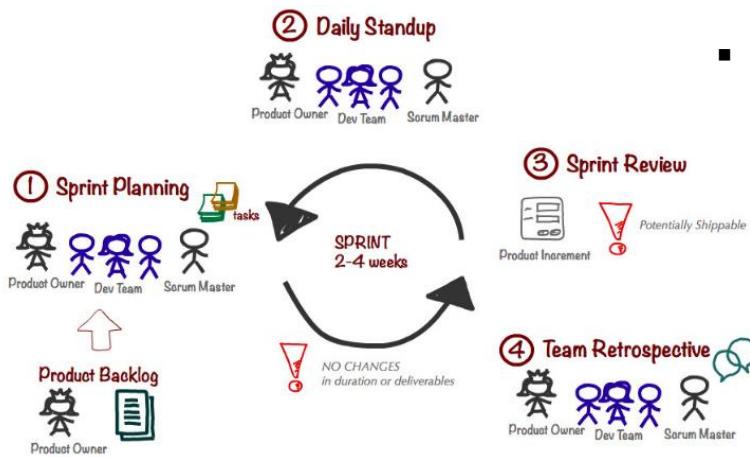
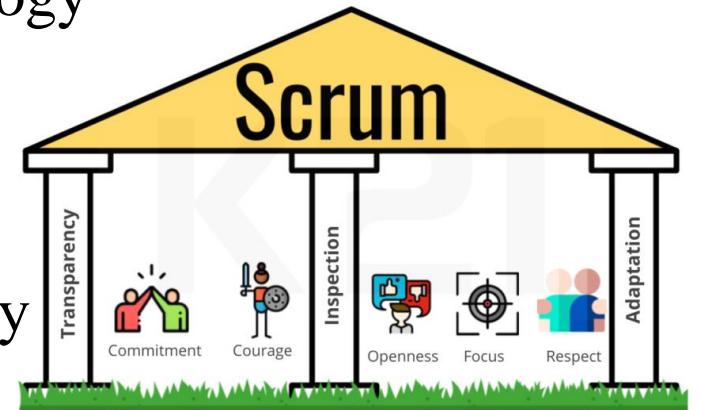
- A Scrum felhasználóknak gyakorta kell ellenőrizniük a Scrum munkaanyagait, és a cél felé történő haladást
- Nem lehet olyan gyakori, hogy akadályozza a munkát
- Akkor a legeredményesebb, ha képzett elemzők hajtják végre éppen a munkafolyamat megkezdése előtt.

# A Scrum elmélet



## 3) Korrekció (Adaptation)

- Amennyiben egy elemző megállapítja, hogy
  - egy folyamat egy vagy több szempontból a megengedett határokon kívül esik,
  - a végtermék nem lesz megfelelő,akkor módosítani kell a folyamaton, vagy kidolgozás alatt lévő anyagon.



- A Scrum négy formális eseményt ír elő a vizsgálatra és az alkalmazkodásra
  - Sprint Tervezés (Sprint Planning)
  - Napi Scrum (Daily Scrum)
  - Sprint Áttekintés (Sprint Review)
  - Sprint Visszatekintés (Sprint Retrospective)

# Tartalom



- Mi az a Scrum?
- **Scrum csapat jellemzői**
- Scrum események
- Scrum munkaanyagok



# A Scrum csapat

V.P.

- **A Scrum Csapat tagjai:**

- Terméktulajdonos
- Fejlesztőcsapat
- Scrum Mester (Scrum Master)



- Önszerveződők és kereszt-funkcionálisak.
- A csapat modellt a Scrumban a rugalmasság, a kreativitás és a produktivitás optimalizálása érdekében tervezték meg
- Iteratív módon és fokozatos lépésekben (inkrementálisan) szállítják a terméket, maximalizálva a visszajelzés lehetőségét

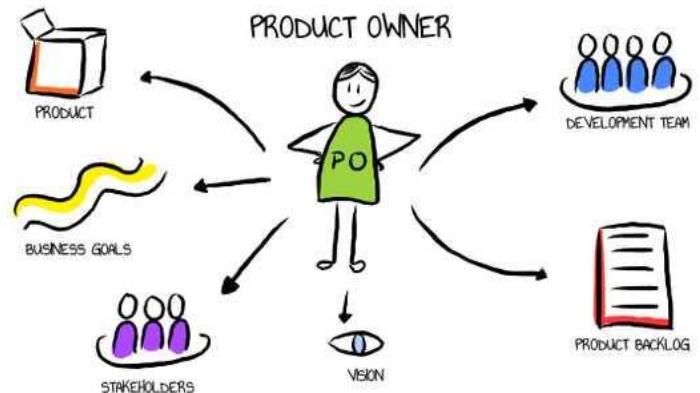
# Product Owner



- A **Terméktulajdonos** (Product Owner) felelős a termék értékének maximalizálásáért és a Fejlesztőcsapat munkájáért
- Nem egy bizottság, hanem **egyetlen személy**.

□ A Terméktulajdonos képviselheti egy bizottság kívánságait a Termék Backlogban,

□ Ha a bizottság meg szeretné változtatni valamelyik Termék Backlog elem prioritását, akkor ezt csak a Terméktulajdonoson keresztül teheti meg.



- Ahhoz, hogy a Terméktulajdonos sikeresen el tudja végezni a feladatát, **a teljes szervezetnek tiszteletben kell tartania a döntéseit.**

# Product Owner

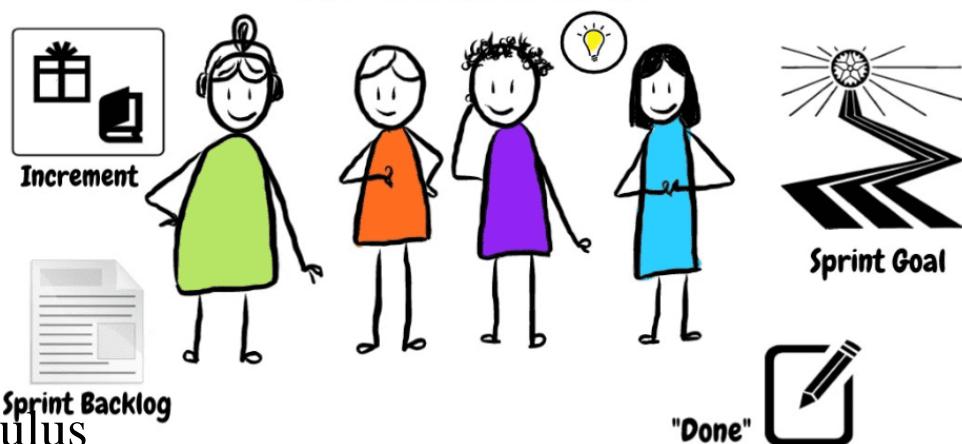


- A **Terméktulajdonos** az egyetlen felelős, aki a Product Backlog (Termék Teendőlista) kezeléséért felelős:
  - 1) Termék Backlog tételeinek egyértelmű leírása;
  - 2) A Termék Backlogban szereplő tételeknek sorba rendezése a célok és küldetések legjobb, leghatékonyabb elérése érdekében;
  - 3) Fejlesztőcsapat által végzett munka értékének optimalizálása;
  - 4) Annak biztosítása, hogy a Termék Backlog elérhető, könnyen áttekinthető és mindenki számára világos legyen
  - 5) Annak biztosítása, hogy a Fejlesztőcsapat legalább a munkavégzéshez szükséges szinten érti a Termék Backlog egyes tételeit

# Development Team



- A **Fejlesztőcsapat** olyan szakemberekből áll, akik azon dolgoznak, hogy minden egyes Sprint végén leszállítható legyen a termék egy “Kész” potenciálisan kibocsátható Inkrementuma.
  - Úgy állítják össze, hogy ők maguk szervezzék és menedzseljék saját munkájukat.
  - Jellemzők:
    - Önszerveződő
    - Kereszt-funkcionálisak
    - „Fejlesztő”-n kívül nincs külön titulus
    - Nincsenek alcsoportok egyes célfeladatok elvégzésére
    - Felelősség az egész Fejlesztőcsapatra, mint egy egységre hárul.



# Development Team

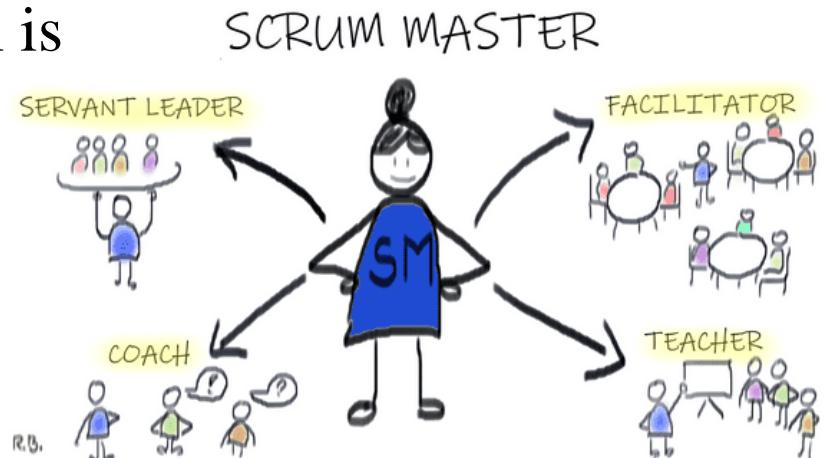


- A Fejlesztőcsapat **optimális mérete**:
  - elég kicsi ahhoz, hogy a csapat gyors reagálású maradjon
  - elég nagy ahhoz, hogy jelentős mennyiségű munkát tudjon végezni egy Sprint alatt
- Kisebb csapat jellemzői:
  - Hárromnál kevesebb tag esetében csökken az interakció mértéke, és ez alacsonyabb termelékenységhez vezet
  - A kisebb csapatok a Sprint során készségkorlátokba ütközhetnek
- Nagyobb csapat jellemzői
  - Kilencnél több tag már túl sok koordinációt igényel.
  - A tapasztalati úton fejlesztett folyamatok kezelése nagy létszámú Fejlesztőcsapatok esetében túl bonyolulttá, nehézkessé válik.

# Scrum Master



- A **Scrum Mester** a Scrum megértéséért és betartásáért felelős, ezt úgy érik el, hogy megbizonyosodnak a csapat Scrum elméleti-, gyakolati- és szabályismeretéről, valamint meggyőződnek elkötelezettségükről is
- A Scrum Csapat szolgáló-vezetője (servent-leader)
- A Scrum Master segít:
  - a Scrum Csapaton kívülieknek megérteni azt, hogy mely Scrum Csapattal való interakciójuk lesz hasznos és melyik nem
  - megváltoztatni ezeket az interakciókat azért, hogy azok a Scrum Csapat által létrehozott értéket maximalizálják.



# Scrum Master



- A **Scrum Mester** szolgáltatásai a Terméktulajdonos felé:
  - Módszereket alakít ki a Termék Backlog hatékony kezelésére;
  - Segít megérteni a Scrum Csapatnak, hogy miért szükséges, hogy a Termék Backlog elemei világosak, tömörek legyenek;
  - Megérti a terméktervezést empirikus környezetben;
  - Biztosítja, hogy a Terméktulajdonos tudja, hogy miként rendezze a Termék Backlogot az érték maximalizálása érdekében
  - Érti és gyakorolja az agilitást; valamint,
  - Kérés illetve szükség esetén előmozdítja a Scrum események lebonyolítását.

# Scrum Master



- A **Scrum Mester** szolgáltatásai a Fejlesztőcsapat felé:
  - Felkészíti, támogatja a Fejlesztőcsapatot az önszerveződésben és a kereszt-funkcionalitás kialakításában;
  - Segíti a Fejlesztőcsapatot magas színvonalú termékek előállításában;
  - Eltávolítja a Fejlesztőcsapat útjába kerülő akadályokat;
  - Kérés illetve szükség esetén előmozdítja a Scrum események lebonyolítását; és,
  - Segíti a Fejlesztőcsapatot olyan szervezeti környezetben, ahol még nem teljes mértékben vezették be és értették meg a Scrumot.

# Scrum Master



- **A Scrum Mester szolgáltatásai a Szervezet felé:**

- Vezeti és képzi a szervezetet a Scrum elsajátításában;
- Megtervezи a Scrum megvalósítását a szervezetben;
- Segít az alkalmazottaknak és az megrendelő oldal szereplőinek megérteni és elfogadni a Scrumot és az empirikus termékfejlesztést;
- Olyan változásokat eszközöl, melyek növelik a Scrum Csapat termelékenységét; és,
- Együttműködik a többi Scrum Mesterrel annak érdekében, hogy növelje a Scrum alkalmazásának hatékonyságát a szervezetben.

# Tartalom



- Mi az a Scrum?
- Scrum csapat jellemzői
- **Scrum események**
- Scrum munkaanyagok



# A Scrum események

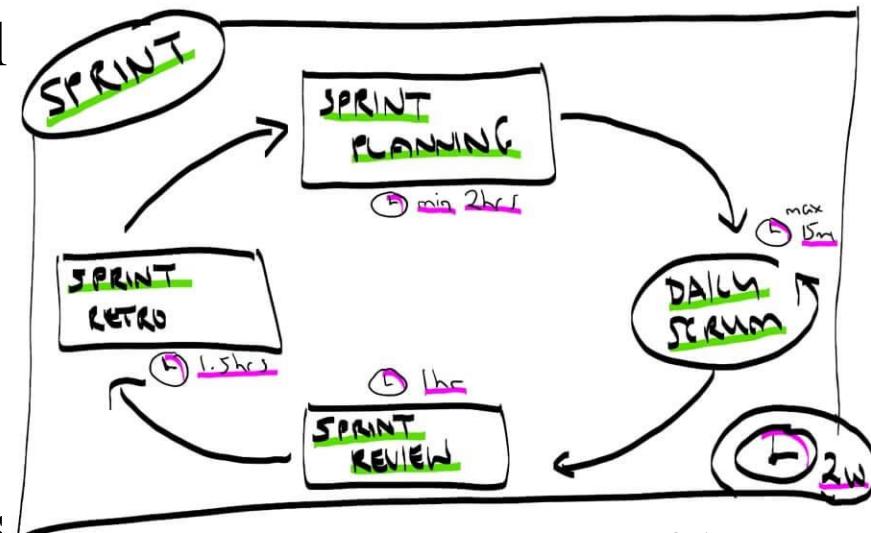


- Cél: rendszerességet, szabályszerűséget teremtenek, és minimalizálják az egyéb, Scrumban nem meghatározott megbeszélések szükségességét
- Időkorlátos (time-boxed) események, vagyis a hossza maximalizálva van
  - a Sprint megkezdődött, annak időtartama rögzített, melyet nem lehet csökkenteni vagy növelni
  - a többi esemény véget érhet, amint az esemény elérte célját
- minden egyes Scrum-esemény (a Sprinten kívül) egy formális lehetőség valaminek az ellenőrzésére és korrekciójára
- Ezeket az eseményeket kifejezetten úgy terveztek meg, hogy biztosítani tudják a kritikus átláthatóságot és ellenőrizhetőséget.

# Sprint



- A Scrum lelke a legfeljebb egy hónapig tartó **Sprint**, amely eredményeként előáll egy “Kész”, használható és potenciálisan kibocsátható termék Inkrementum.
- A Sprintek hossza legjobb esetben a teljes fejlesztési idő során azonos
- Az előző Sprint lezárása után azonnal egy újabb Sprint kezdődik
- A Sprintek a következőkből épül fel
  - Sprint Tervezés
    - Napi Scrumok
    - Fejlesztési munka
    - Sprint Áttekintésből
    - Sprint Visszatekintés



# Sprint



- Minden egyes Sprint egy hónapnál nem hosszabb horizonttal rendelkező projektnek tekinthető.
- Tartalmaz:
  - egy meghatározást, ami leírja, hogy minek kell megvalósulnia,
  - egy modellt és egy rugalmas tervet, ami irányt mutat a megvalósításban.
- A Sprint során:
  - Nem történnek olyan változtatások, melyek veszélyeztetik a Sprint Célját;
  - A minőségi célok nem csökkennek; és,
  - A Terméktulajdonos és a Fejlesztőcsapat újra tárgyalhatja és tisztázhata a Feladatokat (Scope) az időközben szerzett ismeretek alapján

# A Scrum események

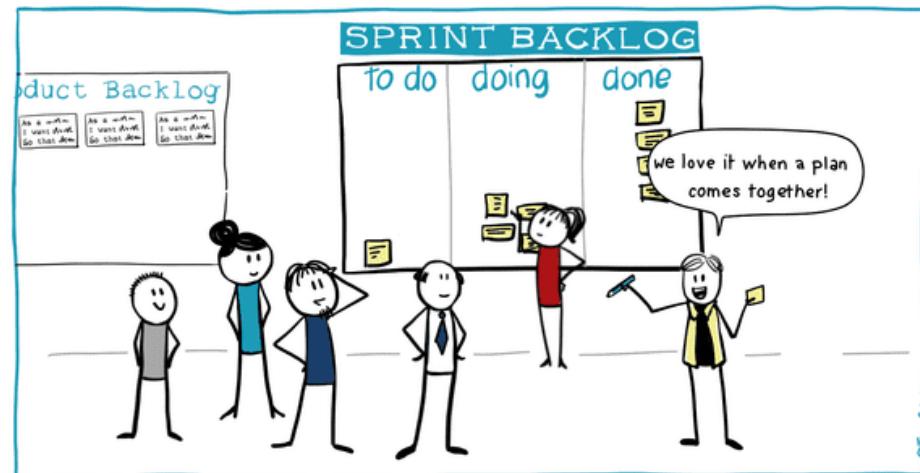


- Ha túl hosszú, megváltozhat a megvalósítandó doleg specifikációja, emelkedhet a komplexitása és nőhet a kockázat.
- Biztosítják a tervezhetőséget, hogy legalább minden naptári hónapban egyszer ellenőrzik a **Sprint Cél** felé haladást, és szükség esetén kiigazítják a folyamatot.
- A kockázatot is egy naptári hónap költségére korlátozzák.
- Sprintet az időkorlát lejárta előtt le lehet fújni, amelyre kizárálag a Terméktulajdonosnak van joga
  - Lehetséges okok: Sprint Cél elavul, okafogyottá válik, a cég irányt változtat, piaci vagy technológiai feltételek megváltoznak

# Sprint Planning



- A Sprintben végzendő munkát a **Sprint Tervezésen** tervezik meg
  - Teljes **Scrum Csapat** közös munkájának eredménye
  - Időtartama 1 hónapos Sprint esetében legfeljebb 8 óra
  - A Scrum Mester biztosítja, hogy az esemény megtörténjen és a résztvevők megértsék annak célját.
- A következő kérdésekre válaszol:
  - Mit fog tartalmazni a következő Sprint eredményeképpen szállítandó Inkrementum?
  - Hogyan lehet elvégezni az Inkrementum előállításához szükséges munkát?



# Sprint Planning



- 1) Mi fog elkészülni ebben a Sprintben?
  - Cél: felvázolni a Sprint során megvalósítandó funkcionálitást
  - A Terméktulajdonos bemutatja a Sprint során elérendő célt és azokat a **Termék Backlog** tételeket, amelyek megvalósításával a Sprint eléri a Sprint Célt.
- Bemeneti elemek:
  - a Termék Backlog,
  - a legutóbbi termék Inkrementum,
  - a Fejlesztőcsapat tervezett kapacitása a Sprint ideje alatt, valamint
  - a Fejlesztőcsapat korábbi teljesítménye

# Sprint Planning



- 1) Mi fog elkészülni ebben a Sprintben?
- Az adott Sprint számára a Termék Backlogból hány tételt választanak ki, egyedül a Fejlesztőcsapaton múlik
- Kizárálag a Fejlesztőcsapat tudhatja, hogy mit képes végrehajtani a soron következő Sprintben
- A Termék Backlog elemeit alapján a Scrum Csapat elkészíti a Sprint Célt
- A Sprint Cél a Sprintben végrehajtásra kiválasztott Termék Backlog elemek megvalósításával elérte célkitűzés, a Sprint során előállított termék Inkrementum tárgyának megfogalmazása

# Sprint Planning



- 2) Hogyan készül el a kiválasztott munka?
- Fejlesztőcsapat eldönti, hogy a Sprint során miként építi be az új funkcionalitást a “Kész” termék inkrementumba
- A Sprintre kiválogatott Termék Backlog tételeket, valamint ezek leszállítási tervét együttesen **Sprint Backlognak** (Sprint Teendőlista) nevezik
- A Fejlesztőcsapat általában a Termék Backlog működő termék inkrementummá konvertálásához szükséges feladatok meghatározásával és a rendszer megtervezésével kezdi meg a munkát.

# Sprint Planning



- A **Sprint Cél** a Sprinthez rendelt célkitűzés, mely a Termék Backlog megvalósításával érhető el.
- Irányt ad a Fejlesztőcsapatnak azzal kapcsolatban, hogy miért fejlesztik az inkrementumot.
- Sprint Cél enged némi rugalmasságot a Fejlesztőcsapatnak a Sprint során megvalósított funkcionálitás kapcsán
- A kiválasztott Termék Backlog tételek egy összefüggő funkcionálitást jelentenek, amelyet a Sprint Cél fogalmaz meg.
- Ha kiderül, hogy a munka eltér attól, mint amire a Fejlesztőcsapat számított, a Fejlesztőcsapat a Terméktulajdonossal újratárgyalja a Sprint Backlog terjedelmét (scope) a Sprintben

# Daily Scrum



- A **Napi Scrum** megbeszélés egy maximum 15 perc időtartamú megbeszélés, ahol a Fejlesztőcsapat összehangolja a tevékenységeket, és megtervezи az elkövetkezendő 24 órát.
  - a legutóbbi Napi Scrum megbeszélés óta elvégzett feladatok elemzésével, majd
  - a következő Napi Scrum előtt elvégezhető feladatok megtervezésével teszi meg
- minden nap ugyanabban az időben, ugyanazon a helyen tartják
- A fejlesztőcsapat minden egyes tagja az alábbiakat fejti ki
  - Mit sikerült elvégeznem tegnap, ami a Fejlesztőcsapatot segítette a Sprint Cél elérésében?
  - Mit fogok tenni ma, ami a Fejlesztőcsapatot segíti a Sprint Cél elérésében?
  - Látok-e akadályozó tényezőt, ami gátol engem vagy a Fejlesztőcsapatot a Sprint Cél elérésében?

# Daily Scrum



- A Napi Scrum során a Fejlesztőcsapat ellenőrzi a Sprint Célhoz vezető folyamat haladását és azt, hogy
- A haladás tendenciája miként változik a Sprint Backlogban szereplő munka teljesítése felé.
- Maximálja annak a valószínűségét, hogy a Fejlesztőcsapat eléri a Sprint Célt.
- A Scrum Mester biztosítja:
  - a Fejlesztőcsapat tagjai minden nap megtartsák a megbeszélést, de a Fejlesztőcsapat felelős a Napi Scrum vezetéséért
  - kizárolag a Fejlesztőcsapat tagjai vegyenek részt a Napi Scrumon



# Sprint Review



- A **Sprint Áttekintést** a Sprint végén tartják azzal a céllal hogy ellenőrizzék az Inkrementumot és szükség esetén módosítsák a Termék Backlogot
- Scrum Csapat tagjai és az megrendelő oldal (stakeholders) egyeztetik, hogy mi történt a Sprint során
- a résztvevők egyeztetik a következő időszakban végrehajtandó, optimális/maximális értéket képviselő teendőket
- Cél: az Inkrementum bemutatásán keresztül visszajelzés érkezzen a megrendelő oldal (stakeholders) részéről
- A megbeszélésnek az időtartama egy 1 hónapos sprint esetén 4 órára korlátozódik.



# Sprint Review



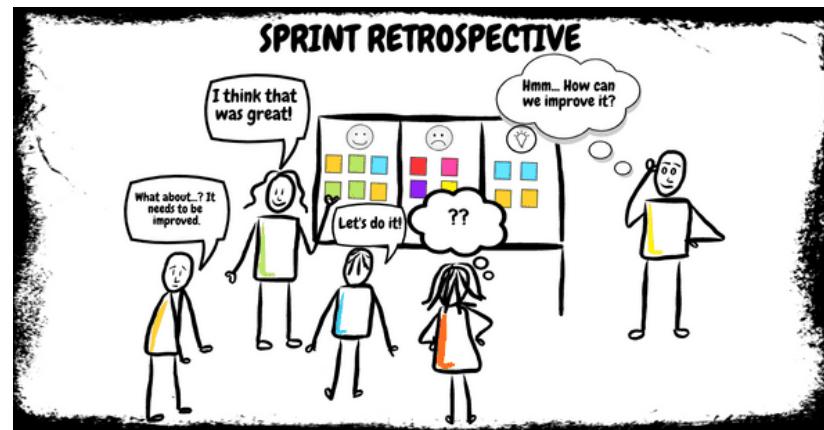
## A Sprint Áttekintés az alábbi elemeket tartalmazza:

- A Terméktulajdonos megállapítja, hogy melyik Termék Backlog térel lett “Kész” és melyik nem lett “Kész”;
- A Fejlesztőcsapat megvitatja, mi ment jól a Sprint során, milyen problémákba futott bele, és hogyan oldotta meg azokat;
- A Fejlesztőcsapat szemlélteti a “Kész” munkát, és válaszol az inkrementummal kapcsolatos kérdésekre;
- A Terméktulajdonos bemutatja a Termék Backlog aktuális állapotát, előrevetíti a várható befejezési dátumokat az addigi haladás alapján (amennyiben ez szükséges);
- Az összes résztvevő közös munkával meghatározza, hogy mik legyenek a következő feladatok, így a Sprint Áttekintés eredménye értékes bemenetként szolgál a következő Sprint Tervezéshez;
- Annak áttekintése, hogy a piac illetve a termék várható használata megváltoztatta-e azt, hogy mik a soron következő, legnagyobb értéket szállító tennivalók; és
- Az ütemezés, költségvetés, várható képességek, funkciók és a piac áttekintése a termék következő release-ére vonatkozóan.

# Sprint Retrospective



- A **Sprint Visszatekintés** (Sprint Retrospective) egy lehetőség a Scrum Csapatnak arra, hogy elemezze saját tevékenységét.
  - A Sprint Áttekintés után, a következő Sprint Tervezés előtt történik
  - Egy hónapos Sprintek esetén ez egy három órás időtartamra korlátozott megbeszélés.
- A cél:
  - Megvizsgálják, hogy mennyire volt sikeres a legutóbbi Sprint az emberek, kapcsolatok, folyamatok és eszközök szempontjából;
  - Azonosításak és sorba rendezzék a jól működő főbb elemeket és a lehetséges javításokat; valamint,
  - Tervet készítsenek a Scrum Csapat működésének javítására.



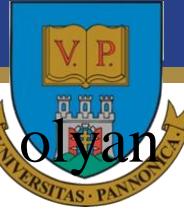
# Tartalom



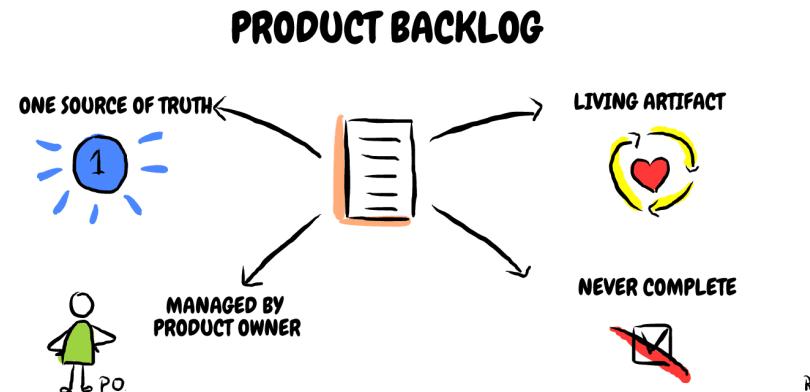
- Scrum elmélet
- Scrum csapat
- Scrum események
- **Scrum munkaanyagok**



# Product Backlog



- A **Termék Backlog** egy sorba rendezett lista, ami minden olyan dolgot tartalmaz, amire szükség lehet a termékben
- A termékkel kapcsolatos változtatási követelmények egyetlen forrása
- Jellemzői:
  - A Terméktulajdonos felelőse
  - Sosem tekinthető teljesnek (kezdetben csak az ismert és legjobban megértett követelményeket fedik le)
  - A termék és a majdani használati környezet változásával összhangban fejlődik
  - Dinamikus; folyamatosan változik annak érdekében, hogy meghatározza azt, hogy mi szükséges ahhoz, hogy a termék megfelelő, versenyképes és hasznos legyen.
  - Ameddig egy termék létezik, a hozzá tartozó Termék Backlog is létezik



# Product Backlog



- A Termék Backlog tartalmazza az összes olyan jellemzők, funkciók, követelmények, tovább fejlesztések és javítások formájában megjelenő változtatást, amiket a termék jövőbeni kibocsátásaiban (**release**) el kell végezni.
- A Termék Backlog tételeihez **leírást, sorrendi helyezést, becslést** és **értéket** rendelnek.
- A Termék Backlog finomítása:
  - további részletekkel, becsléssel egészítjük ki az elemeket
  - változtatjuk azok sorrendjét
- A sorban előbb álló tételek világosabbak és részletesebben kifejtettek, ennek köszönhetően pontosabb becslés készíthető



# Product Backlog

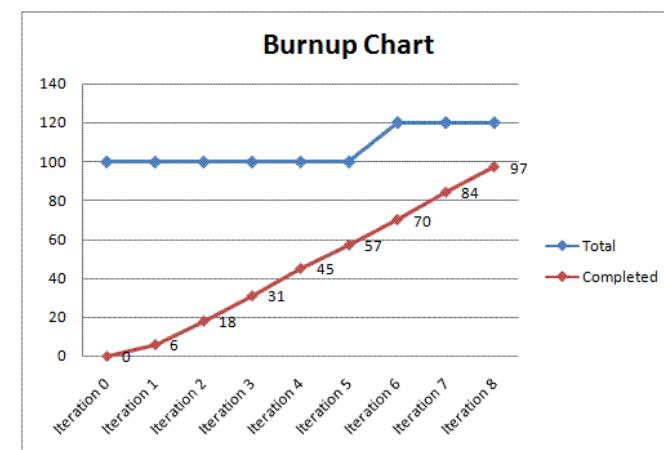


- Azok a Termék Backlog tételek, amelyekkel a Fejlesztőcsapat a soron következő Sprintben foglalkozni fog, kellően részletezettek (finomítottak, granuláltak) ahhoz, hogy bármelyiket „Kész” állapotba lehessen hozni a következő Sprint időtartama alatt.
- Azokat a Termék Backlog tételeket, amiket a Fejlesztőcsapat egy Sprinten belül el tud „Kész”-íteni, a Sprint Tervezésen „Kiválasztható”-nak nyilvánítanak
- A Termék Backlog tételek általában a feljebb leírt Termék Backlog finomítási tevékenységek során érik el ezt a fokú átláthatóságot (készültségi fokot)
- Az összes becslésért a Fejlesztőcsapat felelős.

# Product Backlog



- A Cél felé haladás ellenőrzése:
  - Bármely időpontban összegezhető, hogy mennyi munka szükséges még egy adott cél eléréséhez
  - A Terméktulajdonos legalább minden Sprint Áttekintés alkalmával nyomon követi ezt a hátralévő munkát
- Korábban: burn-up, burn-down és cumulative flow diagramok
  - Hasznosnak bizonyultak, viszont nem helyettesítik a megtapasztalás fontosságát
  - Komplex környezetben nem lehet megjósolni, hogy mi fog történni
  - Csak azokat az információkat lehet hasznosítani a jövőre vonatkozó döntéshozatalhoz, amik már megtörtént eseményeken alapulnak.



# Sprint Backlog

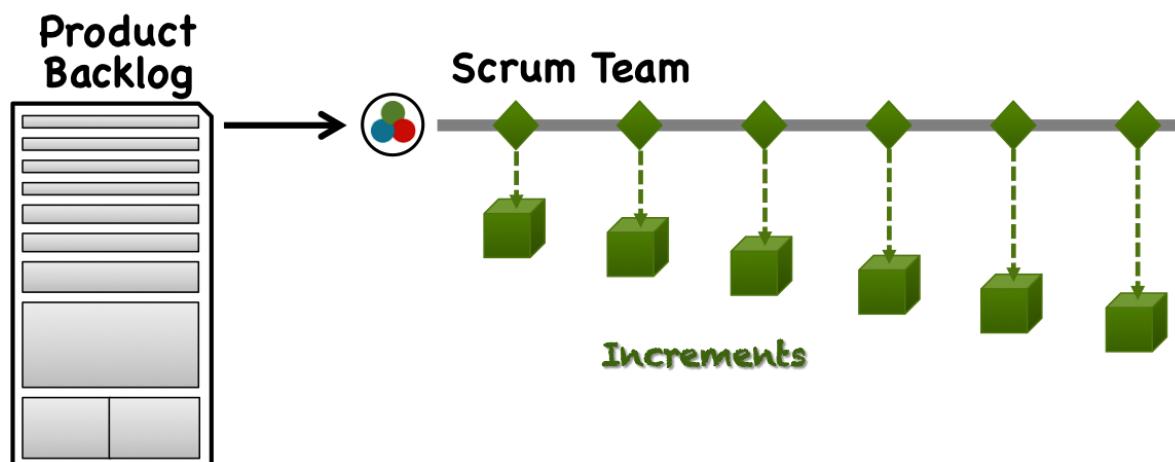


- **Sprint Backlog** a Termék Backlog elemeinek egy, a Sprintre kiválasztott halmazát, plusz a termék Inkrementum leszállítására és a Sprint Cél megvalósítására vonatkozó tervet tartalmaz.
- Előrejelzés arra vonatkozóan, hogy a következő Inkrementum milyen funkcionálitást fog tartalmazni, és mely feladatok végrehajtása szükséges ennek „Kész” inkrementumkénti leszállításához
- Egy terv, ami elég részletes ahhoz, hogy a haladásban bekövetkezett változások a Napi Scrum során érhetők legyenek
  - Sprint során folyamatosan módosul
  - Az új feladatot a Fejlesztőcsapat felveheti a Sprint Backlogba
  - A becsült hátralévő ráfordítást folyamatosan frissítik
  - Sprint alatt kizárolag a Fejlesztőcsapat változtathat a Sprint Backlogon

# Inkrementum



- Az **Inkrementum** a Sprintben leszállított Termék Backlog elemeknek és az összes megelőző Sprint során szállított inkrementumok értékének összessége.
- A Sprint végére az új Inkrementumnak “Kész”-nek, azaz használhatónak kell lennie
- Felhasználható állapotban kell lennie független attól, hogy a Terméktulajdonos úgy dönt, hogy ténylegesen kibocsátja-e azt



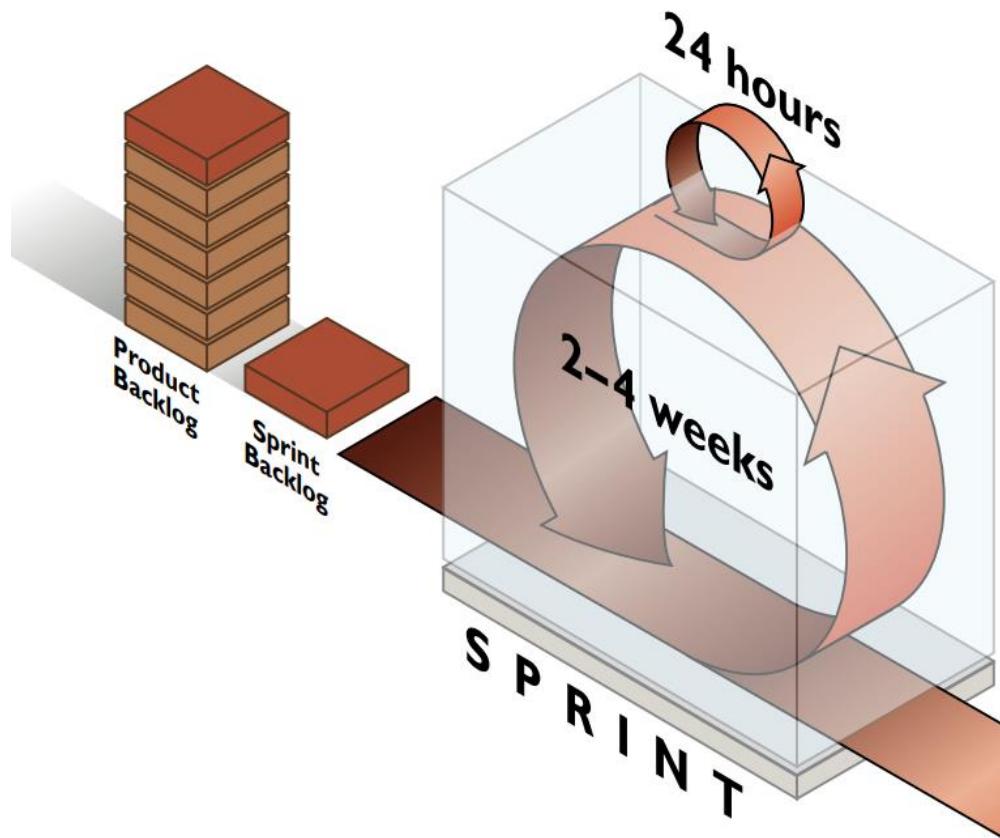
# Definition of “Done”



- Amikor egy Termék Backlog tételt vagy egy Inkrementumot “Kész”-nek nyilvánítanak, mindenkinél pontosan kell tudnia, hogy a “Kész” definíció mit is jelent.
- Segíti a Fejlesztőcsapatot abban, hogy mennyi Termék Backlog tételt válasszon ki a Sprint Tervezésen
- Ahogy a Scrum Csapatok érnek, fejlődnek, általában a “Kész” definíciójuk is velük együtt fejlődik, és még magasabb minőséget biztosító, szigorúbb feltételeket fog tartalmazni.



# Köszönöm a figyelmet!





EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

# CACHE IN-MEMORY

<https://docs.microsoft.com/en-us/aspnet/core/performance/caching/memory?view=aspnetcore-6.0>

<https://nishanc.medium.com/in-memory-caching-on-net-6-0-a38f25698c93>

<https://www.blexin.com/en-US/Article/Blog/In-memory-caching-in-ASPNET-Core-45>

SZÉCHENYI 2020



MAGYARORSZÁG  
KORMÁNYA

Európai Unió  
Európai Strukturális  
és Beruházási Alapok



BEEFEKTETÉS A JÖVŐBE

# Introduction

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

- ASP.NET Core supports several different caches. The simplest cache is based on the `IMemoryCache`.
- `IMemoryCache` represents a cache stored in the memory of the web server.
- Apps running on a server farm (multiple servers) should ensure sessions are sticky when using the in-memory cache.
  - Sticky sessions ensure that subsequent requests from a client all go to the same server.
- The in-memory cache can store any object.
- The in-memory cache store cache items as key-value pairs.

# Cache guidelines

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

- Code should always have a fallback option to fetch data and not depend on a cached value being available.
- The cache uses a scarce resource, memory. Limit cache growth:
  - Do not use external input as cache keys.
  - Use expirations to limit cache growth.
  - Use SetSize, Size, and SizeLimit to limit cache size.
- The ASP.NET Core runtime does not limit cache size based on memory pressure. It's up to the developer to limit cache size.

# Enable in-memory caching

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- If you're creating a Web API you need to call `builder.Services.AddMemoryCache()` in `Program.cs` to register all dependencies.

```
var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
// Add memory cache dependencies
builder.Services.AddMemoryCache();
builder.Services.AddControllers();
```

# Use IMemoryCache

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

- In-memory caching is a service that's referenced from an app using Dependency Injection. Request the IMemoryCache instance in the constructor:

```
public class IndexModel : PageModel
{
    private readonly IMemoryCache _memoryCache;

    public IndexModel(IMemoryCache memoryCache) =>
        _memoryCache = memoryCache;
```

# Use IMemoryCache

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

- The following code uses TryGetValue to check if a time is in the cache. If a time isn't cached, a new entry is created and added to the cache with Set:

```
public void OnGet()
{
    CurrentDateTime = DateTime.Now;

    if (!_memoryCache.TryGetValue(CacheKeys.Entry, out DateTime cacheValue))
    {
        cacheValue = CurrentDateTime;

        var cacheEntryOptions = new MemoryCacheEntryOptions()
            .SetSlidingExpiration(TimeSpan.FromSeconds(3));

        _memoryCache.Set(CacheKeys.Entry, cacheValue, cacheEntryOptions);
    }

    CacheCurrentDateTime = cacheValue;
}
```

# Store and retrieve items

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- The following code uses the Set extension method to cache data for a relative time without MemoryCacheEntryOptions:

```
_memoryCache.Set(CacheKeys.Entry, DateTime.Now, TimeSpan.FromDays(1));
```

- The following codes uses GetOrCreate and GetOrCreateAsync to cache data.

```
var cachedValue = _memoryCache.GetOrCreate(  
    CacheKeys.Entry,  
    cacheEntry =>  
    {  
        cacheEntry SlidingExpiration = TimeSpan.FromSeconds(3);  
        return DateTime.Now;  
    });
```

# Store and retrieve items

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

```
var cachedValue = await _memoryCache.GetOrCreateAsync(  
    CacheKeys.Entry,  
    cacheEntry =>  
    {  
        cacheEntry SlidingExpiration = TimeSpan.FromSeconds(3);  
        return Task.FromResult(DateTime.Now);  
    });
```

# Use SetSize, Size, and SizeLimit

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

- A `MemoryCache` instance may optionally specify and enforce a size limit.
- The cache size limit doesn't have a defined unit of measure because the cache has no mechanism to measure the size of entries.
- If the cache size limit is set, all entries must specify size.
- The ASP.NET Core runtime doesn't limit cache size based on memory pressure. It's up to the developer to limit cache size.

# Use SetSize, Size, and SizeLimit

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

- The following code creates a unitless fixed size  
MemoryCache accessible by dependency injection:

```
public class MyMemoryCache
{
    public MemoryCache Cache { get; } = new MemoryCache(
        new MemoryCacheOptions
    {
        SizeLimit = 1024
    });
}
```

- The following code registers MyMemoryCache with  
the dependency injection container:

```
var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddSingleton<MyMemoryCache>();
```

# Use SetSize, Size, and SizeLimit

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

- The size of the cache entry can be set using the SetSize extension method or the Size property:

```
if (!_myMemoryCache.Cache.TryGetValue(CacheKeys.Entry, out DateTime cacheValue))  
{  
    var cacheEntryOptions = new MemoryCacheEntryOptions()  
        .SetSize(1);  
  
    // cacheEntryOptions.Size = 1;  
  
    _myMemoryCache.Cache.Set(CacheKeys.Entry, cacheValue, cacheEntryOptions);  
}
```

# Set expiration policies on cached data

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

- When we store objects with IMemoryCache, the class MemoryCacheEntryOptions provides us various techniques to manage the expiration of cached data.
- We can indicate a fixed time after which a certain key expires (absolute expiry), or it can expire if it is not accessed after a certain time (sliding expiry).
- Furthermore, there is the possibility to create dependencies between cached objects using the Expiration Token.

# Set expiration policies on cached data

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

- Here some examples.

```
//absolute expiration using TimeSpan
_cache.Set("key", item, TimeSpan.FromDays(1));

//absolute expiration using DateTime
_cache.Set("key", item, new DateTime(2020, 1, 1));

//sliding expiration (evict if not accessed for 7 days)
cache.Set("key", item, new MemoryCacheEntryOptions
{
    SlidingExpiration = TimeSpan.FromDays(7)
});

//use both absolute and sliding expiration
cache.Set("key", item, new MemoryCacheEntryOptions
{
    AbsoluteExpirationRelativeToNow = TimeSpan.FromDays(30),
    SlidingExpiration = TimeSpan.FromDays(7)
})
```

- Another interesting feature available with `MemoryCacheEntryOptions` class is the one that permits us to register callbacks, to be executed when an item is removed from the cache.

```
MemoryCacheEntryOptions cacheOption = new MemoryCacheEntryOptions()
{
    AbsoluteExpirationRelativeToNow = (DateTime.Now.AddMinutes(1) - DateTime.Now),
};
cacheOption.RegisterPostEvictionCallback(
    (key, value, reason, substate) =>
    {
        Console.WriteLine("Cache expired!");
});
```

# Conclusions

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

- Using caching in-memory allows you to store data in the server's memory and helps us to improve application performances by removing unnecessary requests to external data sources.
- As we have seen, it is very simple to use.
- I remind you that this approach cannot be used when your app is hosted on multiple servers or in a cloud hosting environment!
- For more information about the in-memory caching click the following link:
- <https://docs.microsoft.com/en-us/aspnet/core/performance/caching/memory?view=aspnetcore-6.0>



EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

# KÖSZÖNÖM A FIGYELMET!

SZÉCHENYI 2020



MAGYARORSZÁG  
KORMÁNYA

Európai Unió  
Európai Strukturális  
és Beruházási Alapok



BEEFEKTETÉS A JÖVŐBE



EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

# DISTRIBUTED CACHING

<https://docs.microsoft.com/en-us/aspnet/core/performance/caching/distributed?view=aspnetcore-3.1>

<https://www.blexin.com/en-US/Article/Blog/Distributed-cache-in-ASPNET-Core-53>

SZÉCHENYI 2020



MAGYARORSZÁG  
KORMÁNYA

Európai Unió  
Európai Strukturális  
és Beruházási Alapok



BEEFEKTETÉS A JÖVŐBE

# Introduction

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

- A distributed cache is a cache shared by multiple app servers, typically maintained as an external service to the app servers that access it.
- A distributed cache can improve the performance and scalability of an ASP.NET Core app, especially when the app is hosted by a cloud service or a server farm.
- A distributed cache has several advantages over other caching scenarios where cached data is stored on individual app servers.
- When cached data is distributed, the data:
  - Is coherent (consistent) across requests to multiple servers.
  - Survives server restarts and app deployments.
  - Doesn't use local memory.

# Introduction

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

- Non-sticky sessions in a web farm require a distributed cache to avoid cache consistency problems.
- For some apps, a distributed cache can support higher scale-out than an in-memory cache. Using a distributed cache offloads the cache memory to an external process.
- The distributed cache interface is limited to byte[].
- The distributed cache store cache items as key-value pairs.
- The app interacts with the cache using the IDistributedCache interface.

# Prerequisites

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

- To use a SQL Server distributed cache, add a package reference to the `Microsoft.Extensions.Caching.SqlServer` package.
- To use a Redis distributed cache, add a package reference to the `Microsoft.Extensions.Caching.StackExchangeRedis` package.
- To use NCache distributed cache, add a package reference to the `NCache.Microsoft.Extensions.Caching.OpenSource` package.

# IDistributedCache interface

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

- The IDistributedCache interface provides the following methods to manipulate items in the distributed cache implementation:
  - Get, GetAsync: Accepts a string key and retrieves a cached item as a byte[] array if found in the cache.
  - Set, SetAsync: Adds an item (as byte[] array) to the cache using a string key.
  - Refresh, RefreshAsync: Refreshes an item in the cache based on its key, resetting its sliding expiration timeout (if any).
  - Remove, RemoveAsync: Removes a cache item based on its string key.

# IDistributedCache interface

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

```
public interface IDistributedCache
{
    byte[] Get(string key);
    Task<byte[]> GetAsync(string key);

    void Set(string key, byte[] value, DistributedCacheEntryOptions options);
    Task SetAsync(string key, byte[] value, DistributedCacheEntryOptions options);

    void Refresh(string key);
    Task RefreshAsync(string key);

    void Remove(string key);
    Task RemoveAsync(string key);
}
```

# Distributed Memory Cache

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

- The Distributed Memory Cache (`AddDistributedMemoryCache`) is a framework-provided implementation of `IDistributedCache` that stores items in memory.
- The Distributed Memory Cache isn't an actual distributed cache. Cached items are stored by the app instance on the server where the app is running.
- The Distributed Memory Cache is a useful implementation:
  - In development and testing scenarios.
  - When a single server is used in production and memory consumption isn't an issue.
- Implementing the Distributed Memory Cache abstracts cached data storage. It allows for implementing a true distributed caching solution in the future if multiple nodes or fault tolerance become necessary.

# Distributed Memory Cache

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

- Use of the Distributed Memory Cache when the app is run in the Development environment in Program.cs:

```
builder.Services.AddDistributedMemoryCache();
```

# Distributed SQL Server Cache

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

- The Distributed SQL Server Cache implementation (`AddDistributedSqlServerCache`) allows the distributed cache to use a SQL Server database as its backing store.
- To create a SQL Server cached item table in a SQL Server instance, you can use the `sql-cache` tool. The tool creates a table with the name and schema that you specify.

# Distributed SQL Server Cache

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

- Create a table in SQL Server by running the sql-cache create command. Provide the SQL Server instance (Data Source), database (Initial Catalog), schema (for example, dbo), and table name (for example, TestCache):
  - `dotnet sql-cache create "Data Source=(local);Initial Catalog=DistCache;Integrated Security=True;" dbo TestCache`
- The table created by the sql-cache tool has the following schema:

Column Name	Data Type	Allow Nulls
<b>Id</b>	nvarchar(900)	<input type="checkbox"/>
Value	varbinary(MAX)	<input type="checkbox"/>
ExpiresAtTime	datetimeoffset(7)	<input type="checkbox"/>
SlidingExpirationInSecon...	bigint	<input checked="" type="checkbox"/>
AbsoluteExpiration	datetimeoffset(7)	<input checked="" type="checkbox"/>

# Distributed SQL Server Cache

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

- The sample app implements SqlServerCache in a non-Development environment in Program.cs:

```
builder.Services.AddDistributedSqlServerCache(options =>
{
    options.ConnectionString = builder.Configuration.GetConnectionString(
        "DistCache_ConnectionString");
    options.SchemaName = "dbo";
    options.TableName = "TestCache";
});
```

# Distributed Redis Cache

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- Redis is an open source in-memory data store, which is often used as a distributed cache.
- You can use Redis locally, and you can configure an Azure Redis Cache for an Azure-hosted ASP.NET Core app.
- An app configures the cache implementation using a RedisCache instance (AddStackExchangeRedisCache) in a non-Development environment in Program.cs:

```
builder.Services.AddStackExchangeRedisCache(options =>
{
    options.Configuration = builder.Configuration.GetConnectionString("MyRedisConStr");
    options.InstanceName = "SampleInstance";
});
```

# Distributed Redis Cache

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

- To install Redis on your local machine:
  - Install the Chocolatey Redis package.
    - <https://chocolatey.org/packages/redis-64/>
  - Run redis-server from a command prompt.

# Distributed NCache

## Cache

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

- NCache is an open source in-memory distributed cache developed natively in .NET and .NET Core.
- NCache works both locally and configured as a distributed cache cluster for an ASP.NET Core app running in Azure or on other hosting platforms.
- To configure NCache:
  - Install NCache open source NuGet.
  - Configure the cache cluster in [client.ncconf](#).
  - Add the following code to Program.cs:

```
builder.Services.AddNCacheDistributedCache(configuration =>
{
    configuration.CacheName = "democache";
    configuration.EnableLogs = true;
    configuration.ExceptionsEnabled = true;
});
```

# Use the distributed cache

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- To use the `IDistributedCache` interface, request an instance of `IDistributedCache` from any constructor in the app. The instance is provided by dependency injection (DI).
- When the sample app starts, `IDistributedCache` is injected into `Program.cs`. The current time is cached using `IHostApplicationLifetime`:

```
app.Lifetime.ApplicationStarted.Register(() =>
{
    var currentTimeUTC = DateTime.UtcNow.ToString();
    byte[] encodedcurrentTimeUTC = System.Text.Encoding.UTF8.GetBytes(currentTimeUTC);
    var options = new DistributedCacheEntryOptions()
        .SetSlidingExpiration(TimeSpan.FromSeconds(20));
    app.Services.GetService<IDistributedCache>()
        .Set("cachedTimeUTC", encodedcurrentTimeUTC, options);
});
```

# Use the distributed cache

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

- In the sample code each time the Index page is loaded, the cache is checked for the cached time in OnGetAsync.
  - If the cached time hasn't expired, the time is displayed.
  - If 20 seconds have elapsed since the last time the cached time was accessed (the last time this page was loaded), the page displays Cached Time Expired.
- Immediately update the cached time to the current time by selecting the Reset Cached Time button. The button triggers the OnPostResetCachedTime handler method.

# Use the distributed cache

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

```
public class IndexModel : PageModel
{
    private readonly IDistributedCache _cache;
    public string CachedTimeUTC { get; set; }

    public IndexModel(IDistributedCache cache)
    {
        _cache = cache;
    }

    public async Task OnGetAsync()
    {
        CachedTimeUTC = "Cached Time Expired";
        var encodedCachedTimeUTC = await _cache.GetAsync("cachedTimeUTC");
        if (encodedCachedTimeUTC != null)
        {
            CachedTimeUTC = Encoding.UTF8.GetString(encodedCachedTimeUTC);
        }
    }

    public async Task<IActionResult> OnPostResetCachedTime()
    {
        var currentTimeUTC = DateTime.UtcNow.ToString();
        byte[] encodedcurrentTimeUTC = Encoding.UTF8.GetBytes(currentTimeUTC);
        var options = new DistributedCacheEntryOptions()
            .SetSlidingExpiration(TimeSpan.FromSeconds(20));
        await _cache.SetAsync("cachedTimeUTC", encodedcurrentTimeUTC, options);
        return RedirectToPage();
    }
}
```

# Recommendations

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

- When deciding which implementation of IDistributedCache is best for your app, consider the following:
  - Existing infrastructure
  - Performance requirements
  - Cost
  - Team experience
- Caching solutions usually rely on in-memory storage to provide fast retrieval of cached data, but memory is a limited resource and costly to expand. Only store commonly used data in a cache.
- Generally, a Redis cache provides higher throughput and lower latency than a SQL Server cache. However, benchmarking is usually required to determine the performance characteristics of caching strategies.
- When SQL Server is used as a distributed cache backing store, use of the same database for the cache and the app's ordinary data storage and retrieval can negatively impact the performance of both.
- Microsoft recommend using a dedicated SQL Server instance for the distributed cache backing store.

# More information

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

- For more information about the Distributed caching click the following links:
- <https://docs.microsoft.com/en-us/aspnet/core/performance/caching/distributed?view=aspnetcore-6.0>
- <https://www.blexin.com/en-US/Article/Blog/Distributed-caching-in-ASPNET-Core-53>
- <https://dev.to/raidzen10/distributed-caching-in-asp-net-core-4fk4>
- <https://medium.com/net-core/in-memory-distributed-redis-caching-in-asp-net-core-62fb33925818>
- <https://jakeydocs.readthedocs.io/en/latest/performance/caching/distributed.html>



EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

# KÖSZÖNÖM A FIGYELMET!

SZÉCHENYI 2020



MAGYARORSZÁG  
KORMÁNYA

Európai Unió  
Európai Strukturális  
és Beruházási Alapok



BEFETKTETÉS A JÖVŐBE