Spring Üzleti Alkalmazások Fejlesztése

Tóth Zoltán

Szegedi Tudományegyetem
Informatikai Intézet
Szoftverfejlesztés Tanszék

2020

Tartalom

Agenda

- Bemutatkozás
- Követelmények
- Osztályzat
- Háttér
- Bevezetés

Bemutatkozás

- Tóth Zoltán

Szoba:	Szoftverfejlesztés Tanszék (Dugonics tér 13.) 138. szoba
Telefon:	+36 62 54-4521
E-mail:	zizo@inf.u-szeged.hu
Honlap:	http://www.inf.u-szeged.hu/~zizo/
Fogadó Óra	Hétfő 15-16

Előfeltételek, jelentkezés

- Előfeltételek
  - Alkalmazásfejlesztés I.
  - Adatbázisok
- Kurzusra Neptunban lehet jelentkezni
- Kísérleti jelleg $\rightarrow$ Nincs mód növelni a létszámot

Követelmények

- Előadás
  - Sikeres gyakorlat után CooSpace vizsga
- Gyakorlat
  - A gyakorlat látogatása kötelező
  - Kötelező program készítése
    - Ez adja a gyakorlat jegyét
    - Legalább 50%-ot kell elérni
    - Mivel a kötelező programot folyamatosan kell elkészíteni, így annak **javítása nem lehetséges**
  - Kvíz
    - Minden gyakorlat elején egy Coospace-es kvíz
    - Első két héten nincs
    - Kvíz nem pótolható
    - 50%-ot el kell érni a kvízekből (akkumulatív módon)
    - Gyakorlati jegybe nem számít bele

Előadás jegye

- CooSpace vizsga (100 pont)
- Az elért pontszám alapján a kollokvium jegye:
  - 0 - 49: elégtelen (1)
  - 50 - 62: elégséges (2)
  - 63 - 75: közepes (3)
  - 76 - 88: jó (4)
  - 89 - 100: jeles (5)

Tematika

- Spring keretrendszer madártávlatból
- Spring Inversion of Control (IoC) és Dependency Injection (DI)
- Spring konfigurációs lehetőségek és a Spring Boot
- Spring Data
- Spring MVC, Spring Web
- Spring Security
- Spring REST API
- Spring Microservices
- Spring Testing
- Application Monitoring

Ajánlott irodalom

- Iuliana Cosmina, Rob Harrop, Chris Schaefer, Clarence Ho - **Pro Spring 5 - An In-Depth Guide to the Spring Framework and Its Tools**. Fifth Edition, 2017 Apress, DOI 10.1007/978-1-4842-2808-1
- Craig Walls - **Spring in Action**. Fifth Edition, 2018, ISBN: 9781617294945, Publisher: Manning Publications
- Dinesh Rajput - **Spring 5 Design Patterns**. 2017, ISBN: 9781788299459, Publisher: Packt Publishing

Spring Üzleti Alkalmazások Fejlesztése

Tóth Zoltán

Szegedi Tudományegyetem
Informatikai Intézet
Szoftverfejlesztés Tanszék

2020

Tartalom

Mi az a Spring?

- Üzleti alkalmazások fejlesztéséhez használható keretrendszer (JEE helyett, illetve mellett)
- Groovy és Kotlin támogatás
- Rugalmas architektúra az igényeknek megfelelően
- Első release 2002-ben
- 5.1-es verzió után legalább JDK 8 (11-es JDK támogatás)
- Nyílt forráskódú
- Projektek/modulok gyűjteménye. Pl.:
  - Core Container, Config, DI
  - Security
  - AOP
  - Microservices
- Lightweight keretrendszer
  - minimális konfig (lightweight)
  - Bármilyen Java-s alkalmazás fejlesztéséhez (Asztali, web)

A Spring lelke - IoC, DI

- Inversion of Control (IoC)
  - Komponensek közötti függőségek kiszervezése
  - Komponensek automaikus létrehozása és menedzselése
  - Példa: Foo példánya függ Bar példányától $\rightarrow$ létre kell hozni manuálisan
  - IoC: futás közben kívülről kapjuk meg automatikusan $\Leftrightarrow$ dependency injection (DI)
  - Bean: minden Spring által menedzselt osztály
  - Interfészek: flexibilitás biztosítása (kód többlet szükséges)
  - DI használatával csökkenteni tudjuk a plusz kód mennyiségét, sőt nulla közelire redukálhatjuk
  - Egyfajta container-ként funkcionál (létrehozás a függőségekkel együtt, azok menedzselése)
  - Abszolút nem tolakodó módon végezhető (minimális extra kód)
  - DI for Java (2009): JSR-330

Dependency Injection előnyei

- Sokkal kevesebb "glue code"
- egyszerűbb rendszer konfigurációk (pl.: DB csere)
- Egyszerűbb kezelhetőség a közös függőségek kezelése tekintetében (pl.: data source connection, transaction, remote services)
- tesztelhetőség javítása (pl.: DAO műveletek lecserélése mock adatokra)
- Jó tervezési megoldások támogatása (általában interface-ekre kell majd terveznünk), melyet ha követünk, akkor egy rakás Spring-es feature-t kapunk cserébe (automatikus összedrótozások)

Hello World - Reloaded

1. Készítsünk el egy Hello World alkalmazást!
   - Nem túl rugalmas: Üzenet cseréje? Kiírás módja (stderr vagy HTML)?
   - Újradordítás, újratesztelés
2. Üzenetrész kiszervezése és futás közbeni beolvasása (command line arg)
   - renderer-t nehéz módosítani
3. Refactoring: interface-ek használata (MessageProvider, MessageRenderer)
   - Problem: ha más implementációt választanék, akkor megint újra kell fordítanom a kódot
4. Factory class: property fájl használata az implementáció megadáshoz
5. Spring-es implementáció

Application Context, konfiguráció

- Application Context
  - interface a környezeti információk tárolására
  - rajta keresztül kérhetünk le bármilyen Spring által menedzselt bean példányt (`getBean()`)
  - `ClassPathXmlApplicationContext`: az alkalmazás konfigurációja egy a XML dokumentumból töltődik be
- Annotáció alapú konfiguráció
  - Spring 3.0-tól
  - Olyan osztályok, melyek `@Configuration` annotációval vannak ellátva és benne Bean definíciók szerepelnek (`@Bean`)

Annotáció alapú konfiguráció

- `ClassPathXmlApplicationContext` helyett `AnnotationConfigApplicationContext` használata

```java
@Configuration
public class HelloWorldConfiguration {

    @Bean
    public MessageProvider provider() {
        return new HelloWorldMessageProvider();
    }

    @Bean
    public MessageRenderer renderer(){
        MessageRenderer renderer = new StandardOutMessageRenderer();
        renderer.setMessageProvider(provider());
        return renderer;
    }
}
```

```java
ApplicationContext ctx = new AnnotationConfigApplicationContext
        (HelloWorldConfiguration.class);
```

Spring Üzleti Alkalmazások Fejlesztése

Tóth Zoltán

Szegedi Tudományegyetem
Informatikai Intézet
Szoftverfejlesztés Tanszék

2020

Tartalom

IoC

- Az az elv, amely alapján az objektumok vezérlését (vagy a program bizonyos részeit) egy konténerre bízzuk
- Általában OO-ban használt
- A framework-nek megengedjük, hogy ő irányítsa a program futását, hívásokat intézzen a mi kódunkhoz (absztrakció szükséges hozzá)
- egyedi viselkedéskor ki kell terjesztenünk a framework osztályait, vagy plugin-szerűen kell hozzáadni további osztályokat
- Előnyök:
  - feladat végrehajtása és implementációja szétválasztva
  - különböző implementációk között egyszerű a váltás
  - magasabb modularitás biztosítása
  - tesztelés megkönnyítése
- Altípusok:
  - dependency lookup (tradicionális): a függő objektum szerez valahonnan referenciát a függőségre
  - **dependency injection** (a függőséget a konténer oldja fel)

Dependency Lookup vs. Dependency Injection

- Dependency Lookup
  - Dependency pull: Pl.: JDNI-vel, vagy előző órán (`ctx.getBean(...)`)
  - Contextualized Dependency Lookup (CDL): konténer menedzseli és nem központi registry-ből szedünk adatot. Általában kell egy interfészt implementálni hozzá (Pl.: `ManagedComponent`)
- Dependency Injection
  - Constructor alapú: argumentumok a függőségek
  - Setter alapú: függőség beállítása setter-rel
  - Field alapú: Közvetlenül a field-be injektálunk reflection-nel
- Springben mindig Dependency Injection-t fogunk használni, mivel az a dependency lookup-al ellentétben egyáltalán nem befolyásolja az általunk írt kódot, így az **független marad a konténertől**

Dependency Injection

- Az IoC egy fajtája
- Lényege a függőségek megadása, melyet később a framework felhasznál
- A függőségek megadását (példányosítását, azaz injektálását) nem maguk az objektumok végzik, hanem egy külön "assembler" komponens (a **IoC konténer**)
- Tradicionális kód:

```
1    public class Store {
2        private Item item;
3
4        public Store() {
5            item = new ItemImpl1();
6        }
7    }
```

- `Item` interface kiválasztott megvalósításának példányosítása a konstruktorban
- Rossz megoldás, mert bele van égetve a választott implementáció példányosítása

Dependency Injection folyt.

- DI használatával ezt elkerülhetjük

```
1     public class Store {
2         private Item item;
3         public Store(Item item) { // bármilyen implementáció jöhet
4             this.item = item;
5         }
6     }
```

- A kiválasztott implementációt ettől függetlenül valahol meg kell, hogy adjuk
- Ezt metaadat formájában tudjuk majd megtenni (pl: XML), de sokszor a keretrendszer ki tudja találni (pl.: csak egy implementáció létezik)

Spring IoC konténer

- IoC konténer: Általánosan használt, ahol IoC támogatás van
- Spring-ben maga az IoC konténer is egy interface mögé van rejtve, amellyel már találkoztunk is: `ApplicationContext` (`extends BeanFactory`)
- Felelőssége: objektumok példányosítása, konfigurálása, életciklusaiknak menedzselése
- Az összes Spring által menedzselt objektumot *bean*-nek nevezzük
- Spring által biztosított `ApplicationContext` megvalósítások:
  - `ClassPathXmlApplicationContext`: standalone apps
  - `FileSystemXmlApplicationContext`: standalone apps
  - `WebApplicationContext`: web apps
  - `AnnotationConfigApplicationContext`: web apps
- A konfigurációs metaadat (bean deafiníciók) alapvetően XML, property vagy annotáció alapú
- Példa:

```
1    ApplicationContext context
2    = new ClassPathXmlApplicationContext("applicationContext.xml");
```

BeanFactory és ApplicationContext

- a DI magját adja, ő menedzseli a bean-eket (konténer által menedzselt komponens, objektum)
- BeanFactory használata (konkrét implementációval: `DefaultListableBeanFactory`):
  - XML (`XmlBeanDefinitionReader`)
  - Property file (`PropertiesBeanDefinitionReader`)
- Ajánlás: `ApplicationContext`-et használjuk, mert az több támogatást nyújt (pl.: annotáció alapú konfig, i18n, stb.)
- ApplicationContext konfigurálása: XML vs. annotáció
  - XML, property $\rightarrow$ kiszervezi a konfigot a kódból
  - Annotáció: DI beállítások definiálása és áttekintése kódból
  - Ajánlás: XML-be, property-be infrastrukturális konfigok kerüljenek (pl: datasource, tranzakció kezelő megadása, JMS beállítások, stb), a DI konfigokat végezzük el annotációkkal

DI típusok

- A Spring által biztosított Dependency Injection fajták
  - Konstruktor alapú
  - Setter alapú
  - Field alapú

Konstruktor alapú DI

- A konténer a bean konstruktorát meghívja az argumentumokkal, ahol az összes argumentum egy-egy függőséget jelent
- Az argumentumok feloldása elsődlegesen típus (aztán név, és index) alapján történik
- Lássunk egy annotáció alapú bean konfigurációt a függőségeivel együtt

```
1    @Configuration
2    @ComponentScan("hu.suaf")
3    public class Config {
4
5        @Bean
6        public Engine engine() {
7            return new Engine("v8");
8        }
9
10   }
```

- `@Configuration`: az osztályban bean definíciókat adunk meg (több ilyen osztály is lehet)
- `@ComponentScan`: további bean-ek automatikus felderítése az adott packgage-ben
- `@Bean`: konkrét bean definíció. Névmegadás nincs → metódus neve

Konstruktor alapú DI folyt.

- `Autowired` használata konstruktoron

```
1     @Component
2     public class Car {
3
4         @Autowired
5         public Car(Engine engine) {
6             this.engine = engine;
7         }
8     }
```

- `@Component` bean megadása stereotype annotációval
- A `Config` osztály `engine` metódusát hívja a rendszer amikor egy új `Car` objektumot hoz létre.
- **Implicit Ctr Injection**: Spring 4.3-tól elhagyható az `@Autowired` a konstruktor elől, ha:
  - az osztálynak csak egy konstruktora van

Konstruktor alapú DI folyt.

- XML alapú megadás

```
1     <bean id="toyota" class="hu.suaf.domain.Car">
2         <constructor-arg index="0" ref="engine"/>
3     </bean>
4
5     <bean id="engine" class="hu.suaf.Engine">
6         <constructor-arg index="0" value="v4"/>
7     </bean>
```

- opcionálisan *index* és *type* megadható a bean-re

Setter alapú DI

- a konténer a setter-ek meghívásával állítja be a függőségeket, miután példányosította az objektumot a default konstruktorral (vagy argumentum nélküli statikus factory metódust) → menet közben is cserélhetőek a függőségek
- Példa:

```
1     @Bean
2     public Store store() {
3         Store store = new Store();
4         store.setItem(item1());
5         return store;
6     }
```

- XML konfigurációval

```
1     <bean id="store" class="org.foo.Store">
2         <property name="item" ref="item1" />
3     </bean>
```

- A konstruktor és a setter alapú injection kombinálható is egy-egy bean-re
- Ajánlás: kötelező függőségekhez konstruktor alapú, opcionális függőség: setter alapú

Field alapú DI

- Field alapú DI esetében az injektálás az `@Autowired` annotációval

```
1    public class Store {
2        @Autowired
3        private Item item;
4    }
```

- Ha nincs konstruktor vagy setter alapú DI az `Item` injektálásához, akkor a konténer reflection-t használ
- megadható XML-ben is
- egyszerűbbnek és tisztábbnak tűnő megoldás, ugyanakkor vannak hátrányai:
  - Reflection-t használ, ami költségesebb
  - Könnyebben megsértjük a Single Responsibility Principle-t
- Az `@Autowired` annotáció az ún. **Wiring** kategóriába tartozik (mellette létezik `@Resource`, `@Inject` annotáció is)

Bean annotációk

- Bean-ek létrehozhatóak XML-ben és a @Bean annotációval
- Bean-ek definiálására használhatóak: `org.springframework.stereotype`, a többit pedig az automatikus komponens szkennelésre hagyjuk
- `ComponentScan`: annotációval megadott beanek automatikus szkennelése megadott package-ben
  - `basePackages`
  - `basePackageClasses`
  - megadott argumentum nélkül *rightarrow* aktuális package (amin a ComponentScan van) és gyerekei

```
1 @Configuration
2 @ComponentScan(basePackages = "hu.suaf")
3 class VehicleFactoryConfig {}
```

  - "repeating annotation": többször is rá lehet rakni az osztályra, de alapból array-t is elfogad
  - @ComponentScans: több `ComponentScan` konfig megadása
  - XML: `<context:component-scan base-package="hu.suaf" />`

Bean annotációk - @Component

- osztály szintű annotáció
- Spring az összes @Component-el annotált osztályt regisztrálja bean-ként

```
1    @Component
2    class CarUtility {
3        // ...
4    }
```

- Alapértelmezés: bean neve = osztály neve kis kezdőbetűvel
- névadás: *value* attribútummal
- Ez a névadás érvényes az összes többi stereotype annotációra, mivel azok a @Component meta-annotációi (maguk is el vannak látva a Component annotációval)

Bean annotációk

- `@Repository`
  - DAO és Repository (adatelérésért felelős) külön réteg
  - Ezen osztályokat @Repository annotációval adjuk meg
  - Előnye: automatikus kivételtovábbítás: az alkalmazott perzisztencia keretrendszer (pl Hibernate) által dobott natív kivételek Spring-es `DataAccessExeption` kivételekké alakulnak.
- `@Service`
  - üzleti logikai réteg bean-jei
- `@Controller`
  - Spring MVC controller réteg beanjei
- `@Configuration`
  - Bean definíciós osztály: `@Bean`-el annotált metódusokat tartalmaz

Autowired

- függőségek automatikus injektálása
- összes függőség megadása konfigurációs fájlban → automatikus drótozás
- Annotáció alapú injektáláshoz engedélyezéséhez:

```
1 @Configuration
2 @ComponentScan("hu.suaf.example")
3 public class Config {}
```

- vagy XML-ben (`<context:annotation-config>`)
- később Spring Boot esetén: `@SpringBootApplication`
- A `ComponentScan` regisztrálja a bean-eket az `ApplicationContext`-ben, így lehetőség nyílik az `@Autowired` használatára
- Az `@Autowired` annotációt használhatjuk **property-ken, konstruktorokon, settereken** is

Autowired folyt.

- Field
  - Láttuk már a Field alapú DI-nál, de nézzünk még egy példát, `@Component` használatával
- Setter

```
1 public class FooService {
2     private FooFormatter fooFormatter;
3     @Autowired
4     public void setFooFormatter(FooFormatter fooFormatter) {
5         this.fooFormatter = fooFormatter;
6     }
7 }
```

- Konstruktor

```
1 public class FooService {
2     private FooFormatter fooFormatter;
3     @Autowired
4     public FooService(FooFormatter fooFormatter) {
5         this.fooFormatter = fooFormatter;
6     }
7 }
```

Autowired folyt.

- Autowired esetén a dependency-knek rendelkezésre kell állniuk az adott bean létrehozásakor, különben `NoSuchBeanDefinitionException`-t kapunk
- Amennyiben nem szükséges létrehozáskor rendelkezésre állnia a függőségnek, akkor `@Autowired(required = false)` használható
- A feloldás alapértelmezetten **típus** alapján működik
  - Több azonos típus esetén kivételt kapunk
  - Feloldás: Explicit bean megadás `@Qualifier` használatával

```
1 @Component("fooFormatter")
2 public class FooFormatter implements Formatter {
3     public String format() {
4         return "foo";
5     }
6 }
7 @Component("barFormatter")
8 public class BarFormatter implements Formatter {
9     public String format() {
10        return "bar";
11    }
12 }
13
14 public class FooService {
15    @Autowired
16    private Formatter formatter; //NoUniqueBeanDefinitionException
17 }
```

Autowired folyt.

- Megoldás:

```
1 public class FooService {
2     @Autowired
3     @Qualifier("fooFormatter") // @Component-ben megadott név
4     private Formatter formatter;
5 }
```

- A típus mellett a **név** is számít: ha az eredeti problémás esetben a field neve fooFormatter lett volna, akkor név alapján ki tudta volna találni a rendszer, hogy azt szerettük volna

Autowired vs. Inject vs. Resource

- Mind használható field-en és setteren is
- `javax.annotation.Resource` (JSR-250)
  - Precedencia: Name, Type, Qualifier
- `javax.inject.Inject` (JSR-330)
  - Precedencia: Type, Qualifier, Name (`@Named`)
- `org.springframework.beans.factory.annotation.Autowired`
  - Precedencia: Type, Qualifier, Name

@Primary

- Magasabb precedencia biztosítása egy bean-nek, amennyiben több ugyanolyan típusú bean is létezik
- Értelmezésben: default
- Use case: meglévő kód egy bean-nel, új azonos típusú bean primary-vel, így nem kell mindenhol átírni qualifier alapon
- Precedencia: Qualifier, Primary, name

```
1  @Configuration
2  public class Config {
3
4      @Bean
5      public Employee JohnEmployee() {
6          return new Employee("John");
7      }
8
9      @Bean
10     @Primary
11     public Employee TonyEmployee() {
12         return new Employee("Tony");
13     }
14 }
```

- csak akkor van értelme, ha van ComponentScan
- `@Order`-rel sorrend megadása

Értékek injektálása - @Value

- Spring által menedzselt bean-ek fieldjeibe érték injektálása
- String érték injektálás (elég haszontalan)

```
1 @Value("string value")
2 private String stringValue;
```

- Property fájlból - `@PropertySource`

```
1 @Value("${value.from.file}")
2 private String valueFromFile;
```

- System property (magasabb precedencia, mint prop file)

```
1 @Value("${systemValue}")
2 private String systemValue;
```

- Default érték megadás

```
1 @Value("${unknown.param:default}")
2 private String default;
```

- értékek listájára is működik
- SpEL (Spring Expression Language) használata

Értékek injektálása - SpEL

- Kifejezések dinamikus kiértékelése és azok használata az ApplicationContext-ben
- Bean-ek közötti érték átadás is
- aritmetikai, relációs, logikai, regex műveletek támogatása
- #{...} formájú (property-k ${...})
- SpEL kifejezés tartalmazhat property-t is

```
#{${someProperty} + 2}
```

- Példák:

```
@Value("#{36 / 2}") // 19
private double divide;
```

```
@Value("#{1 le 1}") // true
private boolean lessThanOrEqualAlphabetic;
```

- Ternary op: '?:' (Elvis operátor)

```
@Value("#{bean.property ?: 'default'}")
private String property;
```

További bean injektálási lehetőségek Spring-ben

- Kollekciók: List, Set, Map, Properties
- Generikus típusok

Bean Scope-ok

- Minden bean rendelkezik egy scope-al, mely meghatározza az életciklusát, a láthatóságát az adott context-en belül
- 6 típusú scope létezik:
  - singleton (default)
  - prototype
  - request (csak web)
  - session (csak web)
  - application (csak web)
  - websocket (csak web)
- `@Scope` használata a bean-en

Lusta inicializálás

- Alapértelmezés: A konténer legyártja és inicializálja az összes singleton bean-t, amikor magát a konténert inicializálja (induláskor)
- Ez a viselkedés megváltoztatható → *lazy-init*

```
1    <bean id="item1" class="org.foo.ItemImpl1" lazy-init="true" />
```

- vagy `@Lazy` annotációval
- Ilyenkor a bean csak akkor kerül inicializálásra, amikor először szükség van rá
- Előny: gyorsabb inicializáció
- Hátrány: Konfigurációs hibák csak később derülnek ki

Spring Üzleti Alkalmazások Fejlesztése

Tóth Zoltán

Szegedi Tudományegyetem
Informatikai Intézet
Szoftverfejlesztés Tanszék

2020

Tartalom

Inicializálási műveletek

- Indításkori műveletek a bean-re vonatkozólag
  - Konténer melózik → problémás lehet
  - Nem lehet simán konstruktorból intézni a dolgokat (dependency akkor még nem áll rendelkezésre, kivéve, ha konstruktor injekcióval dolgozunk)
  - `@PostConstruct`: bean init után fut le közvetlenül

```
1 @Component
2 public class PostConstructExample {
3
4     @Autowired private Environment environment;
5
6     @PostConstruct
7     public void init() {
8         System.out.println(Arrays.asList(environment.getDefaultProfiles()));
9     }
10 }
```

  - `InitializingBean` interface

```
1 @Component
2 public class InitializingBeanExample implements InitializingBean {
3
4     @Autowired private Environment environment;
5
6     @Override
7     public void afterPropertiesSet() throws Exception {
8         System.out.println(Arrays.asList(environment.getDefaultProfiles()));
9     }
```

ApplicationListener

- Miután az egész konténer inicializáció befejeződött

```java
@Component
public class AppListener implements ApplicationListener<ContextRefreshedEvent> {

    public static int counter;

    @Override
    public void onApplicationEvent(ContextRefreshedEvent event) {
        System.out.println("Increment counter");
        counter++;
    }
}
```

```java
@Component
public class EventListenerExampleBean {

    public static int counter;

    @EventListener
    public void onApplicationEvent(ContextRefreshedEvent event) {
        System.out.println("Increment counter");
        counter++;
    }
}
```

- ContextClosedEvent, ContextStartedEvent, ContextStoppedEvent

@Bean initMethod attribútum

- Szintén a bean inicializálás után fut le

```java
public class SampleBean {

    @Autowired
    private Environment environment;

    public void init() {
        System.out.println(Arrays.asList(environment.getDefaultProfiles()));
    }
}
```

```java
@Bean(initMethod="init")
public SampleBean exBean() {
    return new SampleBean();
}
```

CommandLineRunner

- Kontext init után fut le
- Több is megadható ugyanazon kontexten belül (@Order is használható)

```
1 @Component
2 public class CmdRunner implements CommandLineRunner {
3
4     public static int counter;
5
6     @Override
7     public void run(String...args) throws Exception {
8         System.out.println("Increment␣counter");
9         counter++;
10    }
11 }
```

- ApplicationRunner
  - String helyett ApplicationArguments paraméter

Init sorrend

1. Konstruktor
2. @PostConstruct
3. InitializingBean afterPropertiesSet()
4. megadott init metódus

Eltakarítás

- Bean példány megsemmisítése előtt közvetlenül
- Módszerek (lefutás sorrendjében)
  1. `@PreDestroy`

```
@Component
public class DestroyExample {

    @PreDestroy
    public void destroy() {
        System.out.println("PreDestroy called");
    }
}
```

  2. `DisposableBean`

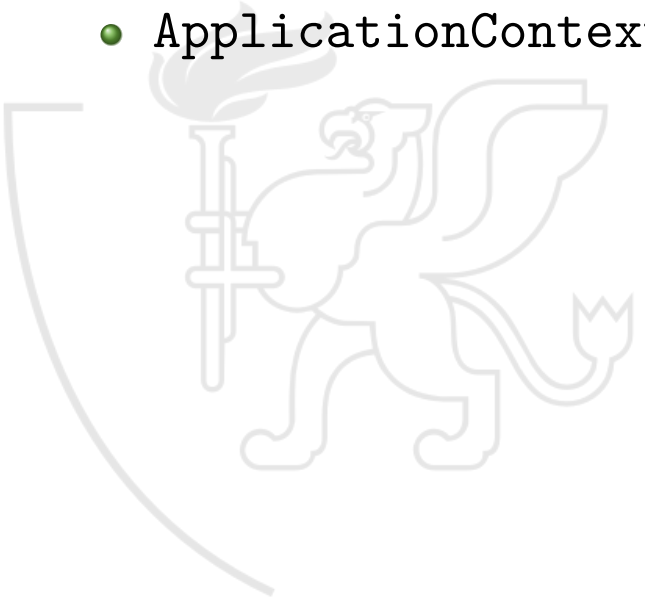
```
@Component
public class ExampleBean implements DisposableBean {

    @Override
    public void destroy() throws Exception {
        System.out.println("destroy called");
    }
}
```

  3. Destroy metódus

```
@Bean(destroyMethod = "destroy")
public ExampleBean initializeBean() {
    return new ExampleBean();
}
```

Spring Aware

- Spring IoC előny $\rightarrow$ nem kell tudnunk semmit a konténerről, az képes összedrótozni az alkalmazás elemeit
- Vannak esetek amikor pont a konténertől van szükségünk valamilyen információra (pl: bean, ami automatikusan konfigurálja a destroy hook-ot)
- Típusok:
  - `BeanNameAware`
  - `ApplicationContextAware`



BeanNameAware

- Olyan bean-ek implementálják, akik futás közben szeretnék lekérni a konténerben használt nevüket
- Spring hívás a setBeanName(String) metódusra init után, de lifecycle event-ek előtt

```
1 public class NameAwareEx implements BeanNameAware{
2     private String name;
3
4     @Override
5     public void setBeanName(String name) {
6         this.name = name;
7     }
8
9 }
```

ApplicationContextAware

- Referencia szerzése a bean-t létrehozó konténerre (`ApplicationContext`-re)
- Ne használjuk dependency létrehozásra, intézze a konténer DI-el
- `setApplicationContext(ApplicationContext)`
- Példa: Destroy hook automatikus regisztrációja

```
1 public class ShutdownHookEx implements ApplicationContextAware {
2     private ApplicationContext ctx;
3
4     public void setApplicationContext(ApplicationContext ctx) throws BeansException {
5         if (ctx instanceof GenericApplicationContext) {
6             ((GenericApplicationContext) ctx).registerShutdownHook();
7         }
8     }
9 }
```

FactoryBean

- Probléma: Hogyan injektálsz, olyan Bean-t, amit nem lehet a `new` meghívásával létrehozni (pl: csak static factory method hívható)?
- Megoldás: `FactoryBean`
- Maga is egy factory-ként működik a bean-ek számára
- ApplicationContext inicializálja a FactoryBean-t
- Dependency esetében a kontext nem a factory bean-t adja vissza, hanem meghívja a FactoryBean.getObject() metódust és annak eredményét adja vissza
- Pl: `MessageDigest`: kódolások használatához, maga az osztály absztrakt és `getInstance()-el` lehet konkrét implementációt kérni

FactoryBean - Példa

```
1  public class MsgDigestEx implements FactoryBean<MessageDigest>, InitializingBean {
2
3      private String algorithmName = "MD5";
4      private MessageDigest messageDigest = null;
5
6      public MessageDigest getObject() throws Exception {
7          return messageDigest;
8      }
9
10     public Class<MessageDigest> getObjectType() {
11         return MessageDigest.class;
12     }
13
14     public boolean isSingleton() {
15         return true;
16     }
17
18     public void afterPropertiesSet() throws Exception {
19         messageDigest = MessageDigest.getInstance(algorithmName);
20     }
21
22     public void setAlgorithmName(String algorithmName) {
23         this.algorithmName = algorithmName;
24     }
25 }
```

MessageSource (i18n)

- String erőforrások (*messages*) különböző nyelven
- megegyező kulcshalmaz (pl: color: brown, color: barna)
- ApplicationContext megvalósítja a MessageSource-ot
- Megvalósítások:
  - `StaticMessageSource` - kívülről nem konfigolható → prodba NE
  - `ResourceBundleMessageSource`
    - `ResourceBundle` alapon
  - `ReloadableResourceBundleMessageSource`
    - szintén `ResourceBundle` alapon
    - támogatja az ütemezett újratöltést
- `HierarchicalMessageSource`: MessageSource példányok egymásba ágyazása

MessageSource (i18n) - Példa

- Form validációs üzenetek

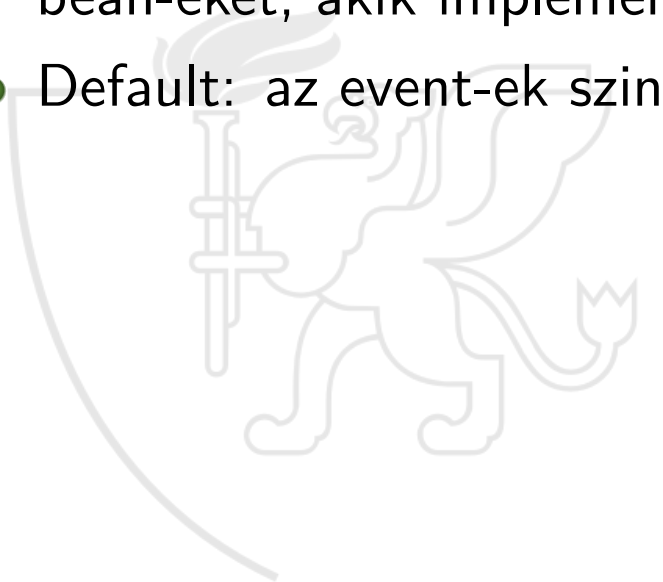
```
1 public class LoginForm {
2
3     @NotEmpty(message = "{email.notempty}")
4     @Email
5     private String email;
6
7     ...
8 }
```

```
1 @Bean
2 public MessageSource messageSource() {
3     ReloadableResourceBundleMessageSource messageSource
4     = new ReloadableResourceBundleMessageSource();
5
6     messageSource.setBasename("classpath:messages");
7     messageSource.setDefaultEncoding("UTF-8");
8     return messageSource;
9 }
10
11 @Bean
12 public LocalValidatorFactoryBean getValidator() {
13     LocalValidatorFactoryBean bean = new LocalValidatorFactoryBean();
14     bean.setValidationMessageSource(messageSource());
15     return bean;
16 }
```

```
1 # messages.properties
2 email.notempty=Please provide valid email id.
```

Application Events

- Egyedi események kezelése az ApplicationContext által
- Guidelines:
  - `MyEvent extends ApplicationEvent`
  - event kiváltója injektáljon egy `ApplicationEventPublisher` objektumot
  - `MyListener extends ApplicationListener<T>`
- ApplicationContext automatikus regisztrálja listenerekként, azokat a bean-eket, akik implementálják az interfészt
- Default: az event-ek szinkron-ok lesznek



Application Events - Példa

```
public class MyEvent extends ApplicationEvent {
    private String message;

    public MyEvent(Object source, String message) {
        super(source);
        this.message = message;
    }
    public String getMessage() {
        return message;
    }
}
```

```
@Component
public class MyEventPublisher {
    @Autowired
    private ApplicationEventPublisher applicationEventPublisher;

    public void publishMyEvent(final String message) {
        System.out.println("publishing␣myEvent");
        MyEvent myEvent = new My(this, message);
        applicationEventPublisher.publishEvent(myEvent);
    }
}
```

```
@Component
public class MyEventListener implements ApplicationListener<MyEvent> {
    @Override
    public void onApplicationEvent(MyEvent event) {
        System.out.println("Got␣the␣event␣-␣" + event.getMessage());
    }
}
```

Erőforrások kezelése

- Egységesített mechanizumus
- Protokol-független módszer (fájl, classpath, remote)
- `org.springframework.core.io.Resource` interface megvalósítások
  - `FileSystemResource`
  - `ClassPathResource`
  - `UrlResource`
- `ResourceLoader` interface (`DefaultResourceLoader` megvalósítás)
  - erőforrás lokalizáció és erőforrás létrehozása
- Nem használjuk direktben a `DefaultResourceLoader`-t, mivel van egy másik implementációja is. Igen, az **`ApplicationContext`**

```
1    ApplicationContext ctx = new ClassPathXmlApplicationContext();
2
3    File file = File.createTempFile("test", "txt");
4    Resource res1 = ctx.getResource("file://" + file.getPath());
5    Resource res2 = ctx.getResource("classpath:test.txt");
6    Resource res3 = ctx.getResource("http://www.google.com");
```

Erőforrások kezelése

- Egységesített mechanizumus
- Protokol-független módszer (fájl, classpath, remote)
- `org.springframework.core.io.Resource` interface megvalósítások
  - `FileSystemResource`
  - `ClassPathResource`
  - `UrlResource`
- `ResourceLoader` interface (`DefaultResourceLoader` megvalósítás)
  - erőforrás lokalizáció és erőforrás létrehozása
- Nem használjuk direktben a `DefaultResourceLoader`-t, mivel van egy másik implementációja is. Igen, az **`ApplicationContext`**

```
1    ApplicationContext ctx = new ClassPathXmlApplicationContext();
2
3    File file = File.createTempFile("test", "txt");
4    Resource res1 = ctx.getResource("file://" + file.getPath());
5    Resource res2 = ctx.getResource("classpath:test.txt");
6    Resource res3 = ctx.getResource("http://www.google.com");
```

- `classpath` protokoll: Spring specifikus

Profilok

- Különböző környezetek megadása
- Bean-ek feltételes regisztrációja a konténerben
- `@Profile`
  - egyszerre akár több is
  - negálás lehetősége
- Profil aktiválása
  - `WebApplicationInitializer (onStartup)`
  - `ConfigurableEnvironment (setActiveProfiles)`
  - web.xml: context param
  - JVM System Parameter: `-Dspring.profiles.active=dev`
  - Környezeti változó: `spring_profiles_active=dev`
  - Maven profile-okkal együttműködve
  - `@ActiveProfile`
  - Spring Boot: `spring.profiles.active` property
- Alapértelmezett profil: `spring.profiles.default`

Profilok - folytatás

- Aktív profil elérése kódból:
  - `Environtment (getActiveProfiles())`
  - Value injektálással (default-ot adjunk meg):

```
1 @Value("${spring.profiles.active}")
2 private String activeProfile;
```

- Profil-specifikus property fájlok
  - `application-profile.properties`

Environment

- Absztakciós szint
- Környezeti információk elérésére
- Összes property kezelése
- `PropertySource`: kulcs=érték alakú megadások

```
1 AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext();
2 ConfigurableEnvironment env = ctx.getEnvironment();
3 MutablePropertySources propertySources = env.getPropertySources();
4
5 Map<String,Object> appMap = new HashMap<>();
6 appMap.put("app.home", "application_home");
7 propertySources.addLast(new MapPropertySource("test_prop_source", appMap));
8
9 System.out.println("app.home: " + env.getProperty("app.home"));
10 ctx.close();
```

- Feloldási sorrend:
  - Rendszer property-k (aktuális JVM-re)
  - Környezeti változók
  - Alkalmazás által definiált property-k
- `@PropertySource, @PropertySources`

Spring Üzleti Alkalmazások Fejlesztése

Tóth Zoltán

Szegedi Tudományegyetem
Informatikai Intézet
Szoftverfejlesztés Tanszék

2020

Tartalom

Bevezetés

- Entities
- CRUD műveletek
- Tradicionális JDBC
  - standard mód DB elérésre
  - Mag: Vendor specifikus Driver
  - `java.sql.DriverManager`
    - Driverek menedzselése
    - Kapcsolat létrehozása DB felé (`getConnection()`)
  - `java.sql.Connection`
    - SQL utasítások futtatása a DB-n
  - Túlságosan komlpex $\rightarrow$ nehézkes vele a fejlesztés
    - DB kapcsolatok létrehozása, menedzselése (kapcsolat létesítése drága dolog)
    - egy kapcsolat = 1 szál
    - túl sok kapcsolat $\rightarrow$ lassulás

Tradicionális JDBC - Példa

```
static {
    try {
        Class.forName("com.mysql.cj.jdbc.Driver");
    } catch (ClassNotFoundException ex) {
        logger.error("Prblem␣loadng␣DB␣Diver!", ex);
    }
}

private Connection getConnection() throws SQLException {
    return DriverManager.getConnection(
    "jdbc:mysql://localhost:3306/testdb?useSSL=true",
    "admin", "admin");
}

private void closeConnection(Connection connection) {
    if (connection == null) {
        return;
    }
    try {
        connection.close();
    } catch (SQLException ex) {
        logger.error("Problem␣closing␣connection␣to␣the␣database!",ex);
    }
}
```

Tradicionális JDBC - Példa (folytatás)

```
1  public List<Contact> findAll() {
2      List<Contact> result = new ArrayList<>();
3      Connection connection = null;
4
5      try {
6          connection = getConnection();
7          PreparedStatement statement =
8              connection.prepareStatement("select␣*␣from␣contact");
9          ResultSet resultSet = statement.executeQuery();
10
11         while (resultSet.next()) {
12             Contact contact = new Contact();
13             contact.setId(resultSet.getLong("id"));
14             contact.setEmail(resultSet.getString("email"));
15             contact.setBirthDate(resultSet.getDate("birth_date"));
16             ...
17             result.add(contact);
18         }
19         statement.close();
20
21     } catch (SQLException ex) {
22         logger.error("Problem␣when␣executing␣SELECT!",ex);
23     } finally {
24         closeConnection(connection);
25     }
26         return result;
27     }
```

Spring JDBC infrastruktúra

- `org.springframework.jdbc.core`
  - JDBC alap osztályok
  - JdbcTemplate
  - SimpleJdbcInsert
  - SimpleJdbcCall
  - NamedParameterJdbcTemplate
- `org.springframework.jdbc.datasource`
  - `DataSource` implementációk
  - JDBC kód futtatása JEE konténeren kívül
  - embedded DB support
  - db inicializálás
  - datasource lookup
- `org.springframework.jdbc.object`
  - db adat konvertálása objektumokká és azok listájává
- `org.springframework.jdbc.support`
  - SQLException translation support

Spring JDBC konfiguráció

- Bean alapú konfiguráció
  - DataSource
- XML alapú konfig
- Embedded DB support
  - EmbeddedDatabaseBuilder

```java
@Configuration
@ComponentScan("hu.suaf.jdbc")
public class JdbcConfig {
    @Bean
    public DataSource mysqlDataSource() {
        DriverManagerDataSource dataSource = new DriverManagerDataSource();
        dataSource.setDriverClassName("com.mysql.jdbc.Driver");
        dataSource.setUrl("jdbc:mysql://localhost:3306/testdb");
        dataSource.setUsername("admin");
        dataSource.setPassword("admin");

        return dataSource;
    }
}
```

- Spring Boot Config
  - `spring-boot-starter-jdbc` és db-connector maven dependency
  - Automatikus konfigurálás

```
spring.datasource.url=jdbc:mysql://localhost:3306/testdb
spring.datasource.username=admin
spring.datasource.password=admin
```

JdbcTemplate

- Fő API
- Funkcionalitások:
  - DB kapcsolat létrehozás és lezárás
  - SQL utasítások futtatása
  - Tárolt eljárások hívása
  - Eredmények visszaadása (ResultSet)
- Példák:

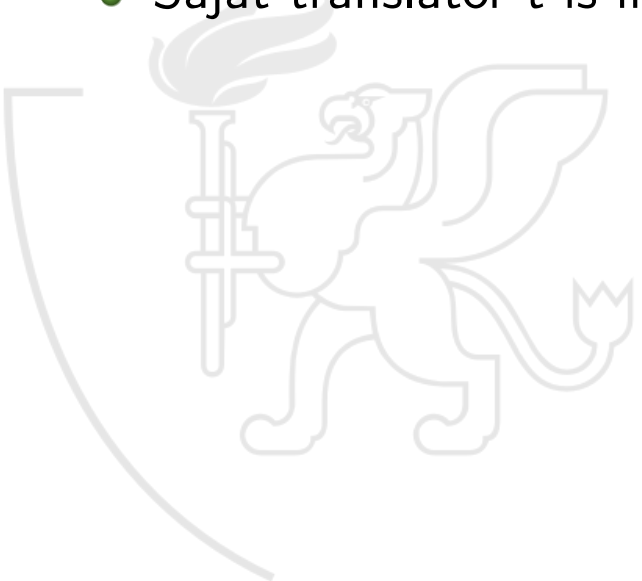
```
int result = jdbcTemplate.queryForObject(
    "SELECT␣COUNT(*)␣FROM␣EMPLOYEE", Integer.class);
```

```
public int addEmplyee(int id) {
    return jdbcTemplate.update(
        "INSERT␣INTO␣EMPLOYEE␣VALUES␣(?,␣?,␣?,␣?)", id, "Bill", "Gates", "USA");
}
```

- Vessük össze a tradicionális JDBC kóddal!
  - Sokkal letisztultabb

JdbcTemplate (folytatás)

- Nevesített paraméterek kezelése: `NamedParameterJdbcTemplate`
- Eredmény mappalése: `RowMapper`
  - átadható a queryForObject számára is
- Spring exceptions:
  - `DataAccessException` a hierarchia tetején
  - `DataAccessException`-be csomagolja a DB exception-t
  - Kivételkezelés független marad a használt DB-től
  - Saját translator-t is írhatunk



További építőkövek

- DB metaadat alapján
- SimpleJdbcInsert
- SimpleJdbcCall
- JdbcTemplate: batchUpdate, BatchSqlUpdate

Spring Üzleti Alkalmazások Fejlesztése

Tóth Zoltán

Szegedi Tudományegyetem
Informatikai Intézet
Szoftverfejlesztés Tanszék

2020

Tartalom

Hibernate - Bevezetés

- JDBC: sok kód
- ORM technika
  - objektumok és adatbázis közötti táblák leképezése
- JPA vs. Hibernate
- Közvetlenül Hibernate vs. JPA-n keresztül

Session, SessionFactory

- Session: kapcsolat a DB felé
- SessionFactory: Session-ök kezelése
  - Implementáció: LocalSessionFactoryBean
  - Beállítások
    - DataSource
    - Hibernate beállítások (DB dialektus, fetch_depth, fetch_size)
    - Entity-t tartalmazó csomagok megadása
- sessionFactory bean definiálása
- FactoryBean használatával
- Referencia a Session-re: sessionFactory.getCurrentSession()

ORM

- Kétféle módszer, megközelítés
  - Model first
    - Model alapján DB generálás
    - hibernate.hbm2ddl.auto=create
  - DB first
    - Táblák létrehozása
    - Majd ez alapján POJO-k
    - Nagyobb kontroll
    - hibernate.hbm2ddl.auto=none
- Foglalt táblanevek
  - User(s),
  - Group(s)
- Foglalt kulcsszavak
  - Lehet escape-elni, de inkább kerüljük

Egyszerű leképezések - Entitások

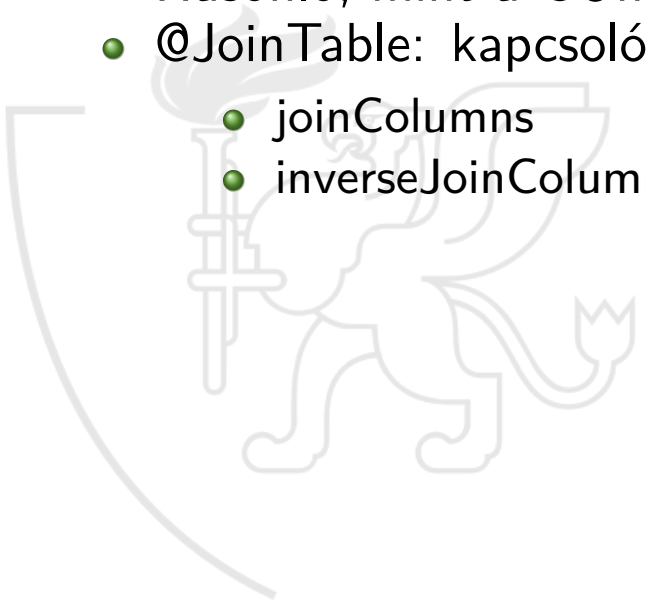
- @Entity
- @Table
- Az entitás osztályok implementálják a Serializable interfészt
- @Id
- @GeneratedValue
  - strategy
    - GenerationType.IDENTITY
    - GenerationType.SEQUENCE
    - GenerationType.AUTO
    - GenerationType.TABLE
- @Column
- @Temporal(TemporalType.xxx)
  - TemporalType.DATE
  - TemporalType.TIME
  - TemporalType.TIMESTAMP
- @Version

Assziciációk

- Típusok:
  - Egy a többhöz: @OneToMany
  - Több az egyhez: @ManyToOne
  - Több a többhöz: @ManyToMany
- Irányok:
  - Egyirányú (unidirectional)
  - Kétirányú (bidirectional)
- @OneToMany
  - mappedBy (asszociált entitásban a reprezentáló field, csak kétirányúnál)
  - cascade (ALL, MERGE, REMOVE, DETACH, REFRESH, PERSIST)
  - orphanRemoval
  - fetch (eager, lazy)
  - targetEntity (a target típusa)

Asszociációk - folytatás

- @ManyToOne
  - cascade (ALL, MERGE, REMOVE, DETACH, REFRESH, PERSIST)
  - fetch (eager, lazy)
  - targetEntity (a target típusa)
  - optional
  - @JoinColumn: külső kulcs megadása
- @ManyToMany
  - Hasonló, mint a @OneToMany
  - @JoinTable: kapcsolótábla megadása
    - joinColumns
    - inverseJoinColumns



HQL és alap műveletek

- Hibernate Query Language
- Domain specifikus nyelv
- Natív SQL-re fordítja a rendszer
- Objektumközeli
- Session.createQuery()

```
1 @Override
2 @Transactional(readOnly = true)
3 public List<Customer> findAll() {
4     return sessionFactory.getCurrentSession().createQuery("from␣Customer␣c").list();
5 }
```

- @NamedQuery (name, query), @NamedQueries
  - Entity-n lehet megadni
  - Használat: session.getNamedQuery([query_neve])
- nevesített paraméterek: setParameter(), setParameterList(), setParamters()
- join fetch
- Műveletek: save(), saveOrUpdate(), delete()

JPA

- Hibernate megvalósítással (OpenJPA, EclipseLink)
- standardizált interfész
- EntityManager
- EntityManagerFactory
- JPQL
- Aktuális verzió: 2.2
- 2.0-tól erősen típusos lekérdezések támogatása
- 2.2:
  - Stream API támgatás
  - Java 8 Time és Date támogatás

EntityManagerFactory konfiguráció

- LocalEntityManagerFactoryBean
  - Legegyszerűbb
  - nem támogatja a DataSource injektálást
  - Tranzakciókezelés nincs
- JEE kompatibilis konténerben való bootstrappalés (DD), majd JDNI lookup
- LocalContainerEntityManagerFactoryBean
  - Támogatja a Datasource injektálást
  - Tranzakciókezelés
- Bean-ek:
  - JpaVendorAdapter (HibernateJpaVendorAdapter)
  - EntityManagerFactory: hasonló beállítási lehetőségek, mint magában a Hibernate-ben
  - PlatformTransactionManager

JPA és ORM

- javax.persistence: JPA kompatibilis
- @PersistentContext: EntityManager injektálás
- Persistence Unit

```
1 em.createNamedQuery("Customer.findAll", Customer.class).getResultList();
```

- Query
- TypedQuery<T>
- View: típus nélküli result
- CRUD műveletek
  - persist
  - merge
  - remove

Natív lekérdezések

- Natív SQL lekérdezés
- Eredmény leképezhető marad az objektumokra
- EntityManager.createNativeQuery()
- @SqlResultSetMapping, @EntityResult(entityClass=Customer.class)

```
@SqlResultSetMapping(
    name="customerResult",
    entities=@EntityResult(entityClass=Customer.class)
)
```

```
@Transactional(readOnly=true)
@Override
public List<Customer> findAllByNativeQuery() {
    return em.createNativeQuery(ALL_CUSTOMER_NATIVE_QUERY, "customerResult").getResultL
}
```

Criteria API

- Keresések field-ek alapján
- Erősen típusos megadások (JPA 2.0-tól)
- Metamodell alapján
  - @StaticMetamodel
  - Attribute, SingularAttribute, SetAttribute, ...
  - Generálható: hibernate-jpamodelgen
- CriteriaBuilder
- CriteriaQuery<T> (select, where)
- Root<T> (CriteriaQuery.from(Class))
- Predicate

Repository

- Spring Data adja
- Repository<T,ID extends Serializable>
- CrudRepository
- JpaRepository
- @Query
- @Param

Entity audit

- Használhatóak
  - @CreatedBy
  - @CreatedDate
  - @LastModifiedBy
  - @LastModifiedDate
- @EntityListeners(AuditingEntityListener.class)
- @MappedSuperclass

Spring Üzleti Alkalmazások Fejlesztése

Tóth Zoltán

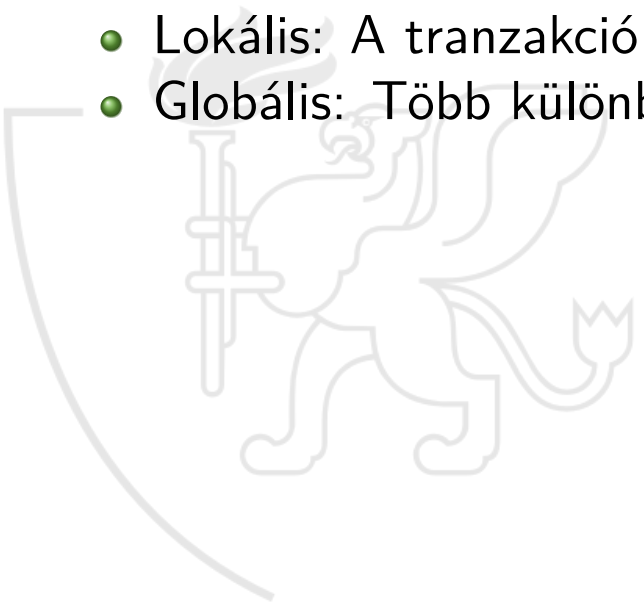
Szegedi Tudományegyetem
Informatikai Intézet
Szoftverfejlesztés Tanszék

2020

Tartalom

Tranzakciókezelés - Bevezetés

- Atomiként kezel utasítássorozat
- Tranzakció eredménye
  - Sikeres: minden tartalmazott utasítás sikeres
  - Sikertelen: egy vagy több utasítás sikertelen
    - Vissza kell állni a tranzakció megkezdése előtti állapotra
- Egy tranzakció több erőforrást is használhat
- Fajtái
  - Lokális: A tranzakció utasításai kizárólag egy erőforrásnak szólnak
  - Globális: Több különböző erőforrást is használ a tranzakció (pl.: 2 DB)



JDBC

- Támogatja a tranzakciókezelést
- Alapértelmezett viselkedés: auto-commit (minden utasítás külön tranzakció)
- Példa:

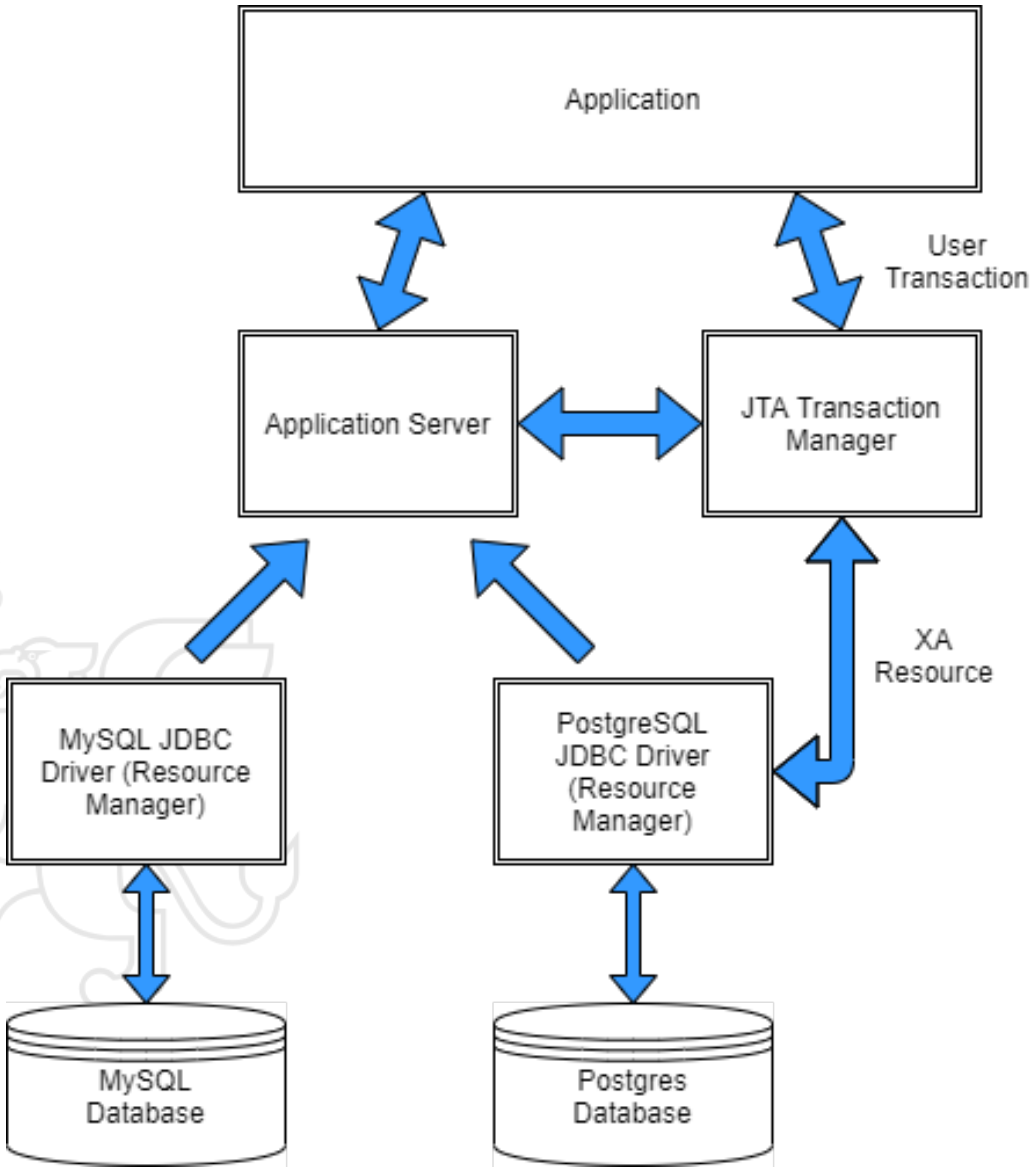
```
1  Connection connection = DriverManager.getConnection(CONNECTION_URL, USER, PASSWORD);
2  try {
3      connection.setAutoCommit(false);
4      PreparedStatement firstStatement = connection .prepareStatement("firstQuery");
5      firstStatement.executeUpdate();
6      PreparedStatement secondStatement = connection .prepareStatement("secondQuery");
7      secondStatement.executeUpdate();
8      connection.commit();
9  } catch (Exception e) {
10     connection.rollback();
11 }
```

JPA

- Egy persistence unit-hoz több EntityManager is tartozhat
- A persistence context:
  - Transaction-scoped: Egy darab tranzakcióhoz kötött, ez az alapértelmezett
  - Extended-scoped: Több tranzakcióhoz is kötődhet
- Példa:

```
1  EntityManagerFactory entityManagerFactory = Persistence
2      .createEntityManagerFactory("jpa-example");
3  EntityManager entityManager = entityManagerFactory.createEntityManager();
4
5  try {
6      entityManager.getTransaction().begin();
7      entityManager.persist(firstEntity);
8      entityManager.persist(secondEntity);
9      entityManager.getTransaction().commit();
10 } catch (Exceotion e) {
11     entityManager.getTransaction().rollback();
12 }
```

Globális tranzakciók



Globális tranzakciók - folytatás

- Elosztott tranzakciókezelés
- PlatformTransactionManager
  - DataSourceTransactionManager (JDBC)
  - JpaTransactionManager (JPA)
  - HibernateTransactionManager (Hibernate)
  - JmsTransactionManager (JMS - Java Messaging System)
- TransactionDefinition
- TransactionStatus

Tranzakciók tulajdonságai

- ACID tulajdonságok szem előtt tartása
  - **A**tomicity: művelet atomi végrehajtása
  - **C**onsistency: érvényes állapotból érvényesbe kerülünk
  - **I**solation: egy időben zajló tranzakciók ugyanazt eredményezik, mintha egymás után futtattuk volna őket
  - **D**urability: adatok tartós adattárolón való tárolása
- TransactionDefinition
  - Propagation Behavior: megadhatjuk, hogy mi történjék, akkor ha már van egy aktív tranzakció folyamatban
  - Isolation Level: konkurens tranzakciók mit láthatnak a másik tranzakció adataiból
  - Timeout: Az idő ami alatt a tranzakciónak meg kell valósulnia, különben sikertelen
  - Read Only: megadja, hogy a tranzakció csak olvasást végez-e
  - Name: minden tranzakció rendelkezik egy névvel

Izolációs szintek

- Fellépő problémák:
  - Dirty read: a még nem kommitált adat kiolvasása egy konkurens tranzakcióból (mely később még változhat)
  - Nonrepeatable read: Ismételt olvasás esetén új rekord eredmény (frissítés történt a kettő olvasás között)
  - Phantom read: Több rekord ismételt olvasása esetén eltérő eredményhalmaz
- Izolációs szintek:
  - ISOLATION_DEFAULT: Alapértelmezett
  - ISOLATION_READ_UNCOMMITTED: Legalacsonyabb szint, a fenti problémák felléphetnek. Olvashat nem kommitált adatot.
  - ISOLATION_READ_COMMITTED: Csak kommitált adatot olvashat. Más tranzakció módosíthatja az adatokat -> Nonrepeatable és Phantom read előfordulhat
  - ISOLATION_REPEATABLE_READ: Két konkurens tranzakció ugyanazt a rekordot nem használhatja egyszerre (Phantom read előfordulhat)
  - ISOLATION_SERIALIZABLE: Legköltségesebb. Olyan mintha a tranzakciók egymás után futnának

Propagálás szintjei

- REQUIRED
- SUPPORTS
- MANDATORY
- NEVER
- NOT_SUPPORTED
- REQUIRES_NEW
- NESTED

TransactionStatus

- New Transaction: új tranzakcióról van-e szó
- Savepoint: van-e beállítva mentési pont a tranzakcióra
- Rollback Only: rollback-elni kell-e a tranzakció alapján
- flush:
- Completed: befejeződött-e a tranzakció (akár rollback vagy commit lett a vége)

@Transactional vs. TransactionTemplate

- Deklaratív tranzakciómegadási mód: @Transactional
  - osztályra: tranzakció rendelkezésre állás
  - metódusra: tranzakciós műveletek személyre szabása
    - Izoláció: DEFAULT
    - Propagáció: REQUIRED
    - Timeout: DEFAULT
    - Mód: Olvasás-írás
- Programozhatóság: TransactionTemplate
  - Callback alapú API
  - execute(...)
    - TransactionCallback<T> callback
    - TransactionCallbackWithoutResult callback

Spring Üzleti Alkalmazások Fejlesztése

Tóth Zoltán

Szegedi Tudományegyetem
Informatikai Intézet
Szoftverfejlesztés Tanszék

2020

Tartalom

Bevezetés

- Biztonság többrétű probléma
- Célja
  - érzékeny adatok védelme
  - értékes erőforrások védelme
- Többrétegű alkalmazások $\rightarrow$ mindegyik megfelelő védelme
- DiD (Depth in Defense) - Védelem megfelelő mélységének meghatározása
  - Adminisztratív kontroll: policy, eljárásmódok, útmutatások
  - Fizikai kontroll: szerver szobák biztonsága, videókamerák
  - Technikai kontroll: Tűzfal, Antivirus, IPS (intrusion prevention systems)
    - Hálózati réteg: tűzfalak, SSL
    - Operációs rendszer réteg
    - Alkalmazás réteg: ez a fókuszunk

Alapfogalmak

- Autentikáció
- Autorizáció
- ACL (Access Control List)
- User
- Credentials
- Role
- Resource
- Permission
- Encryption
  - One-way
  - Symmetric
  - Public key
- Confidentiality
- Integrity

Leggyakoribb támadási formák

- SQL injection
- Denial of Service (DoS)
- Cross-site Scripting (XSS)

Java támogatás

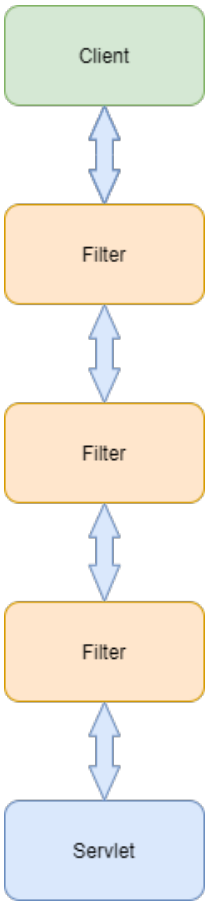
- Java Cryptography Architecture (JCA)
- Java Cryptographic Extensions (JCE)
- Java Certification Path API (CertPath)
- Java Secure Socket Extension (JSSE)
- Java Authentication and Authorization Service (JAAS)

Spring Security

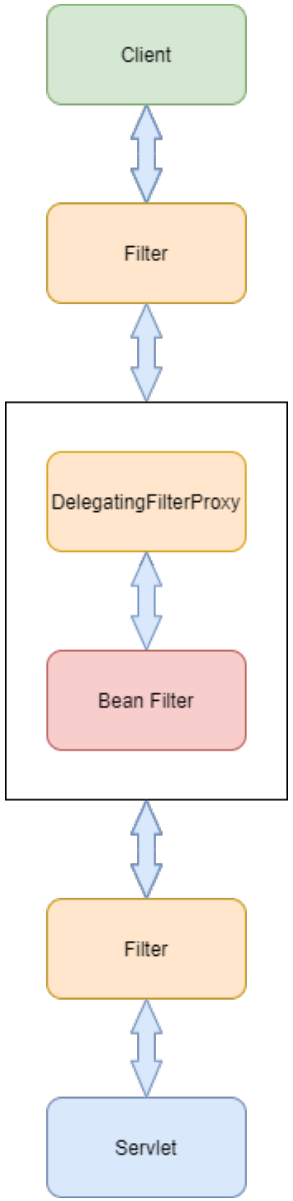
- Aspektus orientált programozás (AOP)
- Interceptorok alkalmazása
  - SecurityInterceptor (`AbstractSecurityInterceptor`)
    - `FilterSecurityInterceptor`
    - `MethodSecurityInterceptor`
  - Preprocesszálási fázis
    - Erőforrás védett-e
    - Igen → Auth objektum lekérése (SecurityContextHolder)
    - AuthenticationManager (ha kell autentikálni)
    - AccessDecisionManager
    - InterceptorStatusToken
  - Postprocesszálási fázis

Filterek

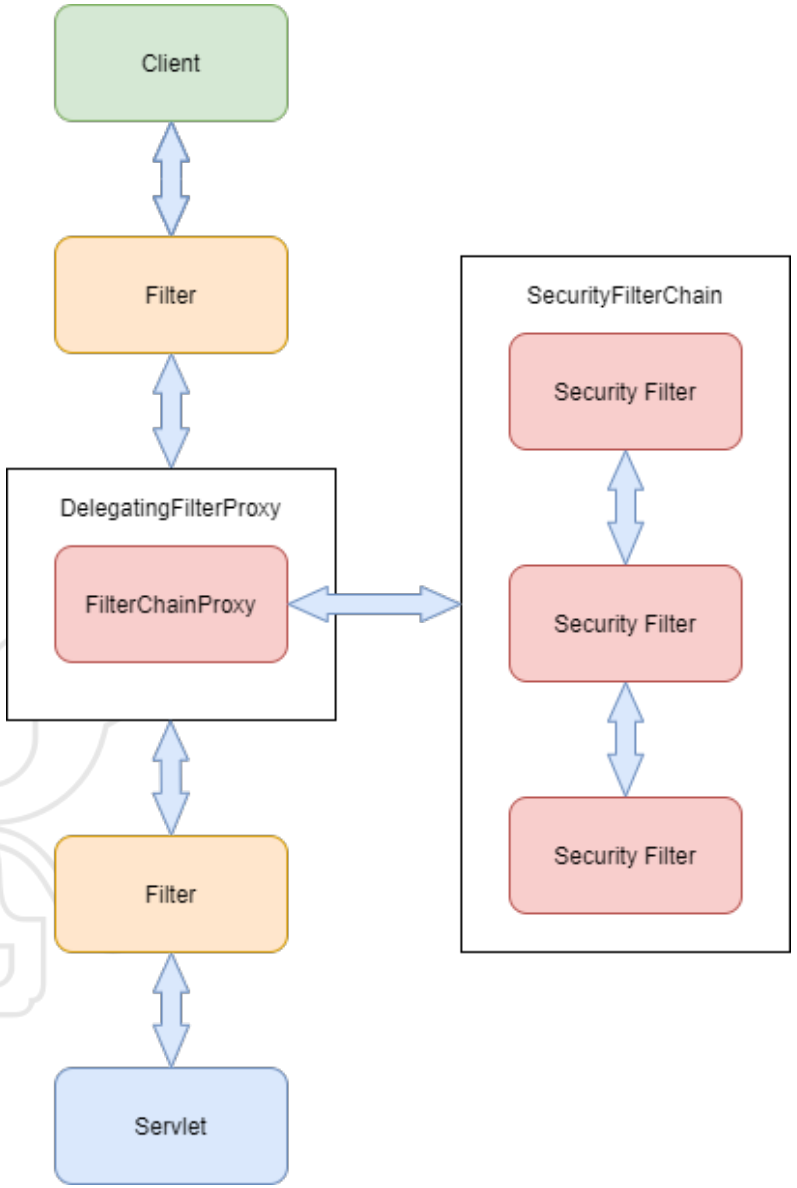
- Filter Chain model
- Standard servlet filter-re épülve
- Single Responsibility filter



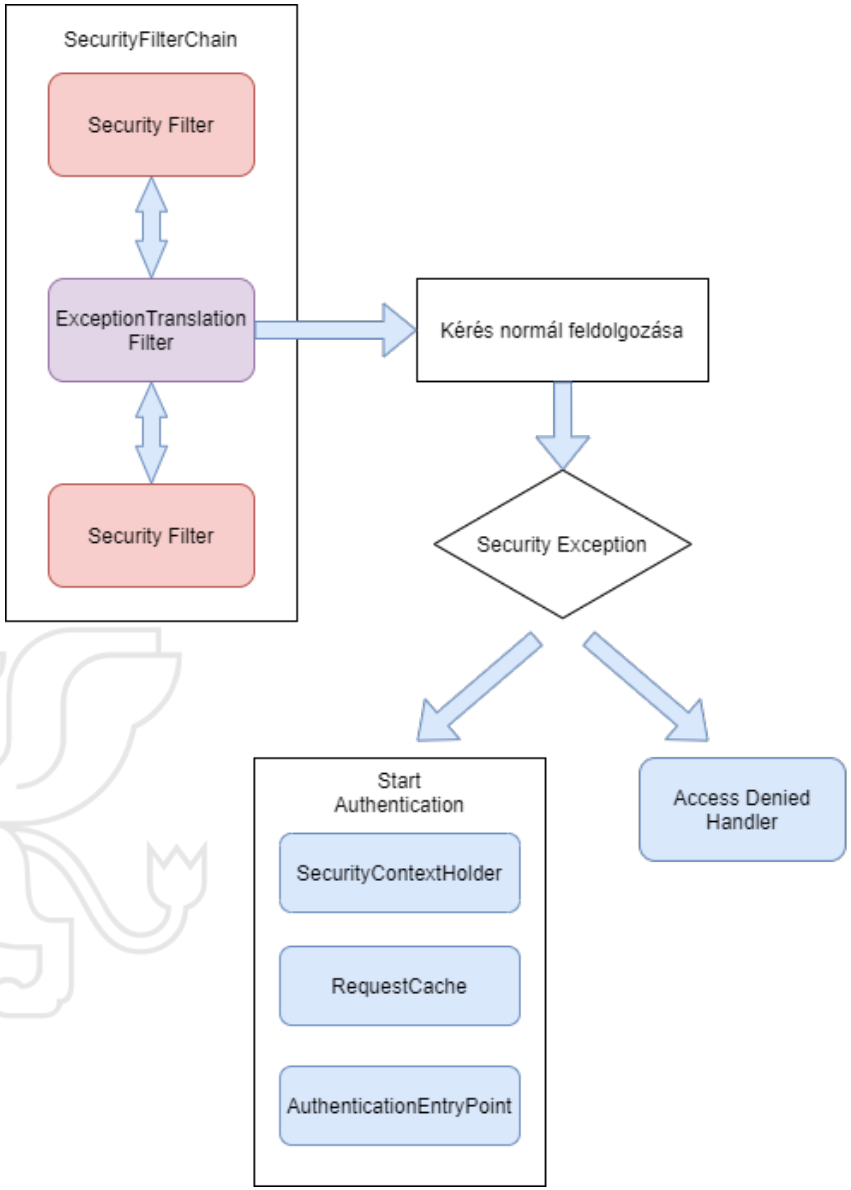
DelegatingFilterProxy



FilterChainProxy



Kivételek kezelése



Kivételek kezelése - folytatás

```
try {
    filterChain.doFilter(request, response);
} catch (AccessDeniedException | AuthenticationException ex) {
    if (!authenticated || ex instanceof AuthenticationException) {
        startAuthentication();
    } else {
        accessDenied();
    }
}
```

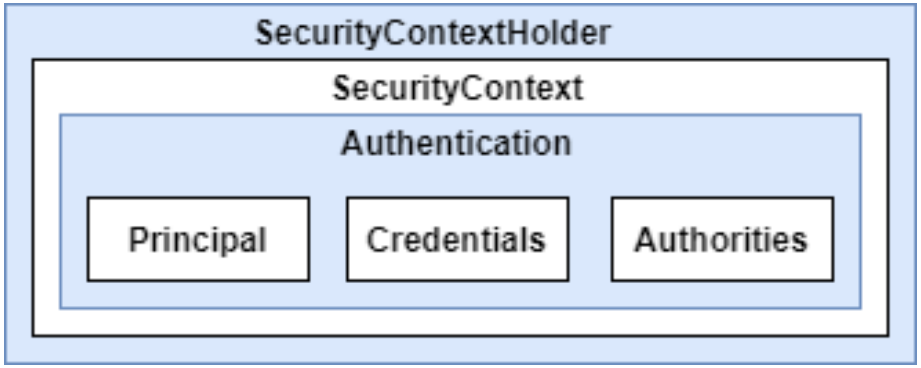
Autentikáció - Komponensek

- SecurityContextHolder
- SecurityContext
- Authentication
- GrantedAuthority
- AuthenticationManager
- ProviderManager
- AuthenticationProvider
- Request Credentials with AuthenticationEntryPoint
- AbstractAuthenticationProcessingFilter

Autentikációs mechanizmusok

- Username and Password
- OAuth 2.0 Login
- SAML 2.0 Login
- Central Authentication Server (CAS)
- Remember Me
- JAAS Authentication
- OpenID
- Pre-Authentication Scenarios
- X509 Authentication - X509 Authentication

SecurityContextHolder




```
1 SecurityContext context = SecurityContextHolder.createEmptyContext();
2 Authentication authentication =
3     new TestingAuthenticationToken("username", "password", "ROLE_USER");
4 context.setAuthentication(authentication);
5
6 SecurityContextHolder.setContext(context);
```

```
1 SecurityContext context = SecurityContextHolder.getContext();
2 Authentication authentication = context.getAuthentication();
3 String username = authentication.getName();
4 Object principal = authentication.getPrincipal();
5 Collection<? extends GrantedAuthority> authorities = authentication.getAuthorities();
```

Authentication objektum

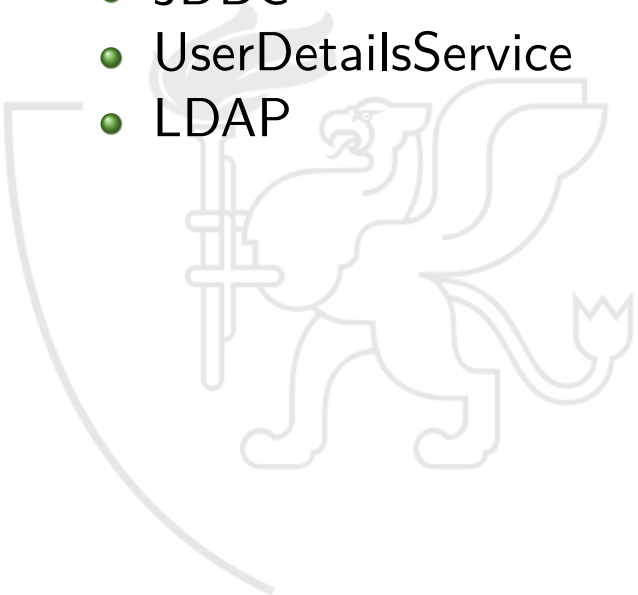
- Kettős szerep
- Principal
- Credentials
- Authorities

ProviderManager

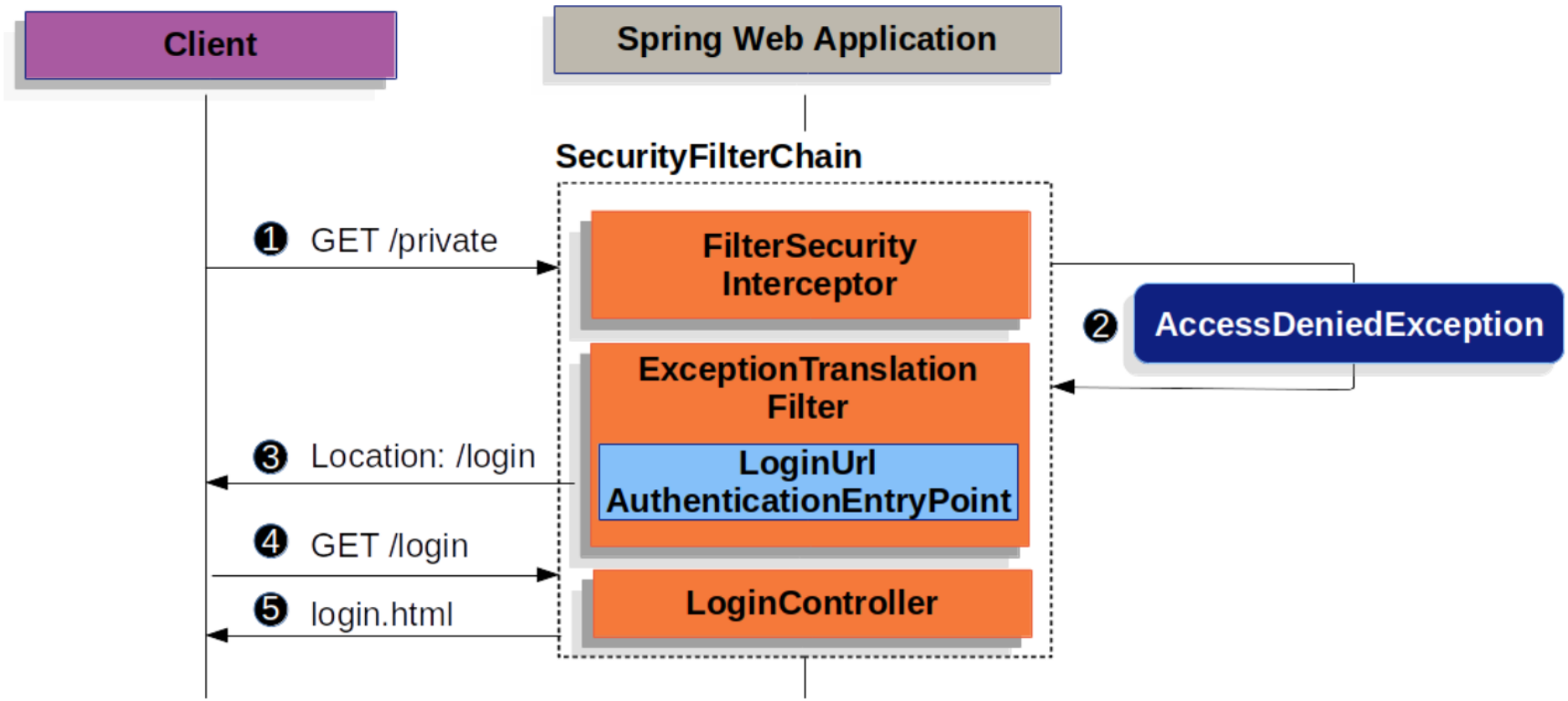
- Leggyakrabban használt AuthenticationManager implementáció
- DaoAuthenticationProvider
- JwtAuthenticationProvider
- parent

Username/Password Authentication

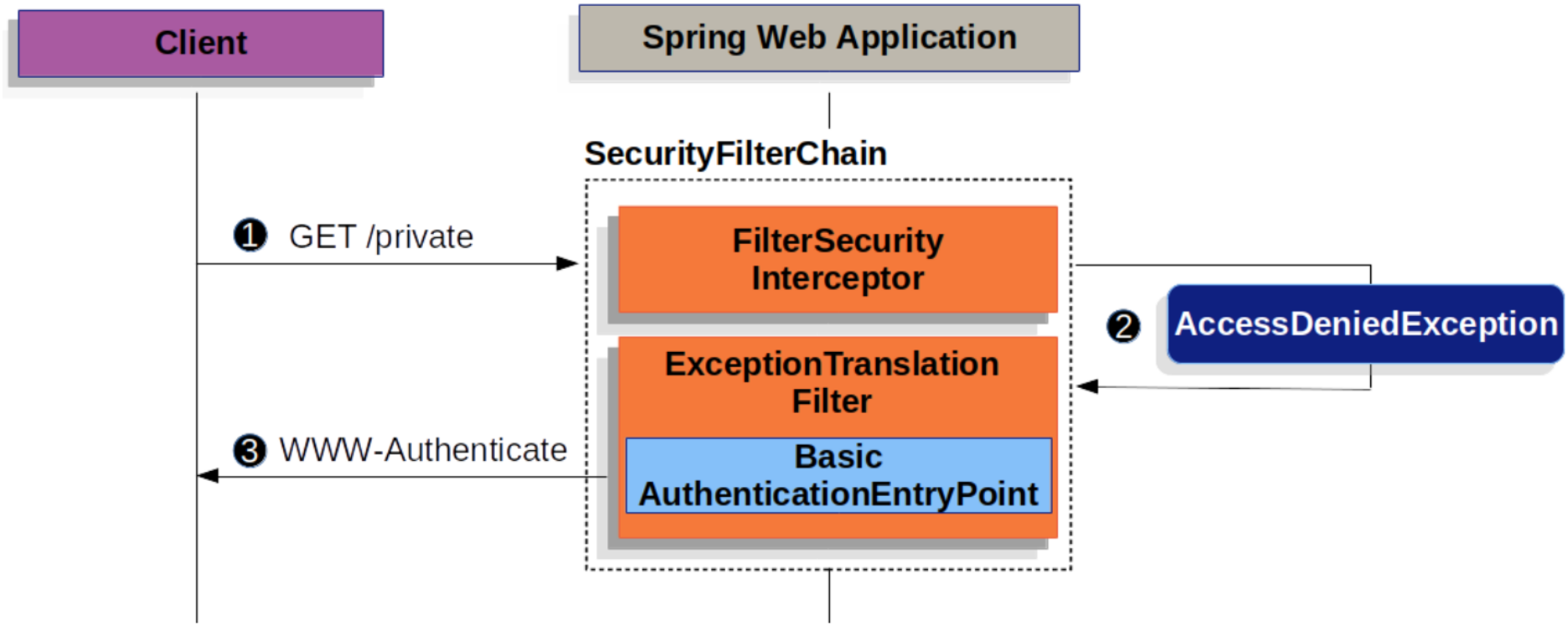
- HttpServletRequest
- Form Login
- Basic Authentication
- Digest Authentication
- Storage
  - In-Memory
  - JDBC
  - UserDetailsService
  - LDAP



Form Login


Client
Spring Web Application
SecurityFilterChain
1 GET /private
FilterSecurity Interceptor
2 AccessDeniedException
ExceptionTranslation Filter
LoginUrl AuthenticationEntryPoint
3 Location: /login
4 GET /login
5 login.html
LoginController

HTTP Basic Authentication



In-memory Authentication

```java
@Bean
public UserDetailsService users() {
    UserDetails user = User.builder()
        .username("user")
        .password("{bcrypt}$2a$10$GRLdNijSQMUvl/au9ofL.eDwmoohzzS7.rmNSJZ.0FxO/BTk76klW")
        .roles("USER")
        .build();
    UserDetails admin = User.builder()
        .username("admin")
        .password("{bcrypt}$2a$10$GRLdNijSQMUvl/au9ofL.eDwmoohzzS7.rmNSJZ.0FxO/BTk76klW")
        .roles("USER", "ADMIN")
        .build();
    return new InMemoryUserDetailsManager(user, admin);
}
```

Spring Üzleti Alkalmazások Fejlesztése

Tóth Zoltán

Szegedi Tudományegyetem
Informatikai Intézet
Szoftverfejlesztés Tanszék

2020

Tartalom

The only constant is change

- Hibák felderítése fejlesztési fázisban
- Szoftver minőség/megbízhatóság növelése
- Minél korábban találunk hibát, annál jobb (olcsóbb)
- Alapvető típusok:
  - White-box
    - Meglévő struktúra tesztelése
    - Forráskód alapján
    - Fejlesztők, illetve független (de cégen belüli) tesztelői csapat
    - Struktúra lehet: Kódsor, Elágazás, Metódus, Osztály, Modul, ...
  - Black-box
    - Specifikáció alapú tesztelés
    - Szoftver a specifikációnak megfelelően működik
    - Független (külsős) tesztelői csapat

Tesztelés szintjei

- Komponens teszt: Rendszer komponensének izolált tesztje.
  - Modulteszt: Nem funkcionális teszt. Pl.: Sebesség, memóriahasználat
  - Unit teszt: Metódusok tesztelése mellékhatások nélkül
- Integrációs teszt
  - Rendszer komponensei közti kölcsönhatások tesztelése
  - Különböző rendszerek közti kölcsönhatások tesztelése
- Rendszerteszt
  - Általában feketedobozos teszt
  - Specifikációnak megfelel-e a termék
- Átvételi teszt
  - Teljes rendszer teszt, de felhasználók tesztelik
  - Típusai:
    - Alfa teszt: Cégen belüli, de nem a fejlesztőcsapat végzi.
    - Béta teszt: Végfelhasználók csoportja végzi
    - Felhasználó átvételi teszt: A rendszer élesben történő használata, de még nem erre alapoznak
    - Üzemeltetői átvételi teszt: Ugyanaz mint az előző, de üzemeltetői szempontból (automatic backup, restoring data, stb)

Test Driven Development (TDD)

- Agilis fejlesztés eleme sok helyen
- Először bukó teszteket írunk, majd javítjuk őket
- Lépések:
  - Teszt az új funkcióhoz
  - Futtatás -> el kell, hogy hasaljon
  - Kód írása -> teszt át kell, hogy menjen
  - Ellenőrzés: a többi teszt továbbra is zöld?
  - Kód refaktorálás
  - Iterálás, ameddig az új funkció el nem készül
- Segíti a rendszer megértését
- Gyorsabb, kis ciklusok
- Célok, amiket segít elérni:
  - Regressziós hibák detektálása
  - Rendszer egyszerű marad (lazán csatoltság segítése)

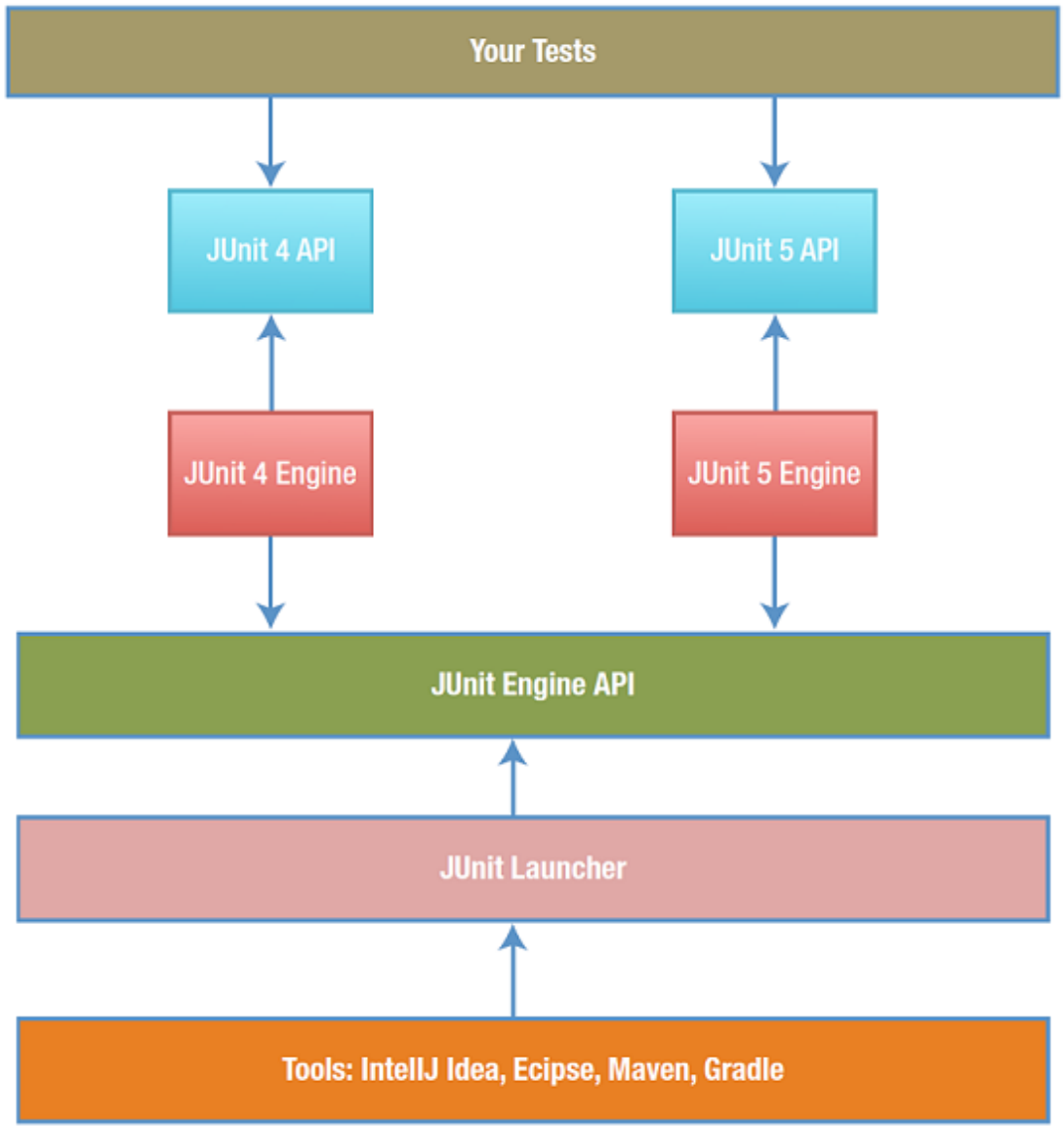
Unit tesztelés

- Amikre figyelni kell:
  - Minden publikus metódust teszteljünk
  - Triviális esetek tesztelése
  - Speciális esetek tesztelése
  - Pozitív és negatív tesztesetek egyaránt
- A jó Unit teszt:
  - Olvasható
  - Gyors
  - Független és izolált
  - Korrekt
  - Környezet agnosztikus
  - Megismételhető
- A legtöbb nyelvhez létezik Unit tesztrendszer
- Java-ban a leginkább használt a JUnit

JUnit 5

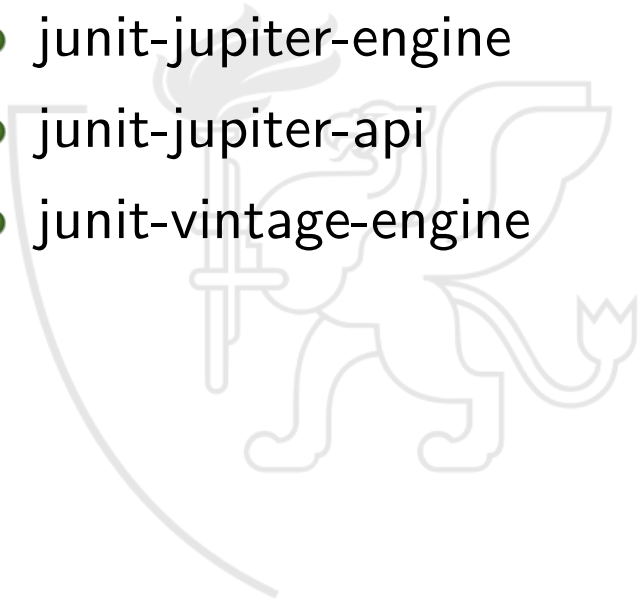
- Miért jó az 5-ös verzió?
  - Modularitás támogatása
  - Bővíthetőség
  - Java 8 támogatás
  - Visszafelé kompatibilis marad
  - A Junit 5 az elődjének az újraírása Java 8-ban
- Architektúra:
  - JUnit Platform
    - TestEngine API
    - ConsoleLauncher
  - JUnit Jupiter
    - TestEngine JUnit 5 implementációja
  - JUnit Vintage
    - TestEngine JUnit 3 és 4 implementációja

JUnit 5 Architektúra



JUnit 5 Modulok

- junit-platform-commons
- junit-platform-launcher
- junit-platform-engine
- junit-platform-console
- junit-platform-gradle-plugin
- junit-platform-surefire-provider
- junit-jupiter-engine
- junit-jupiter-api
- junit-vintage-engine



JUnit 5 Assertions

- Teszt osztály
- Teszt esetek - `@Test`
- Assertions
  - assertEquals(expected, result, message)
  - assertArrayEquals(expected, result, message)
  - assertIterableEquals(expected, result, message)
  - assertNotEquals(expected, result, message)
  - assertTrue(result, message)
  - assertThrows(expected, executable, message)
  - assertDoesNotThrow(expected, executable, message)
  - assertNull(result, message)
  - assertNotNull(result, message)
  - assertSame(expected, result, message)
  - assertNotSame(expected, result, message)
  - assertTimeout(expected, executable, message)
  - assertAll(executable)
  - fail(message)
- 3rd party libek: AssertJ, Hamcrest

JUnit 5 Életciklus

- Fázisok
  - Setup
    - Osztály szinten (@BeforeAll): Költséges inicializációk (DB kapcsolat)
    - Metódus szinten (@BeforeEach)
  - Teszt végrehajtás
  - Clean-up
    - Osztály szinten (@AfterAll)
    - Metódus szinten (@AfterEach)
- @TestListener(Lifecycle.PER_CLASS)
  - Alap viselkedés: Minden tesztesethez egy új teszt instance
  - Ennek következménye a BeforeAll és AfterAll statikussága
  - Fenti viselkedés viszont megosztja a teszt állapotát a tesztesetek között

Függőségek

- Nehézség
  - Komponensek közötti függőségek
  - Izoláció megtartása
  - Teszt inicializálás komplex lehet, vagy túl sok időt vihet el
- Megoldás: Mock-olás
  - Úgy teszünk mintha interakcióba kerülnénk a függő objektummal
  - Egy absztrakciós szintet húzunk a tesztelt komponens függőségei fölé
  - A tesztek egyszerűek és fókuszáltak maradnak
  - A tesztek olvashatóak és karbantarthatóak maradnak
  - Az egyik legnépszerűbb keretrendszer a Mockito
- Kerüljük:
  - Ne mockoljunk egyszerű bean-eket (általánosabban, olyan objektumokat, amiknek nincs viselkedése, pusztán adatot tárol)
  - Ne mockoljunk olyan típusokat, amik nem hozzánk tartoznak (3rd party libs) - ilyenkor igazából találgatnánk a 3rd party lib-ek viselkedését, illetve a viselkedésük változhat, de mivel mockolsz, így nem is tudsz a változásukról

Test Doubles

- A mockolt objektumokat hívják test doubles-nek
- Típusai
  - Manuális (Saját implementáció)
    - Dummy: pl.: interface üres implementációja
    - Fake: Igazi objektum, de egyszerűsített implementációval (pl.: in-memory DB)
    - Stub: hard-coded behaviour
  - Automatikus (Framework biztosítja)
    - Mocks: Objektum, amelyen megadhatjuk az elvárt viselkedést, majd ezt tesztelhetjük is (verify)
    - Spies: Proxy-k valódi objektumok felé, ahol a néhány metódus csak stub, néhány pedig valódi viselkedést biztosít

Mockito limitations

- A következőket nem lehet mockolni:
  - Statikus metódusokat
  - Konstruktorok
  - equals(), hashCode()
  - final metódusok/osztályok
  - private metódusok

Annotációk

- @Mock
- @InjectMocks (típus alapon)
  - Konstruktor alapú DI
  - Setter alapú DI
  - Field alapú DI

Spring Üzleti Alkalmazások Fejlesztése

Tóth Zoltán

Szegedi Tudományegyetem
Informatikai Intézet
Szoftverfejlesztés Tanszék

2020

Tartalom

Bevezetés

- Aszinkron kommunikációs forma
- Integrációs folyamatok szerves részét képezi
- Különböző rendszerek között a kommunikáció üzenetek alapján megy végbe
  - Küldő (producer, sender)
  - Fogadó (consumer, receiver)
  - Csatorna (channel): az üzenet továbbítását lehetővé tevő komponens (pl.: Web socket, HTTP, stb)
  - Message Broker
- Fő előnyök
  - Garantált kézbesítés (Darability)
  - Laza csatoltság
  - Skálázhatóság és magas rendelkezésre állás
  - Aszinkron
  - Interoperability

Üzenetküldési modellek

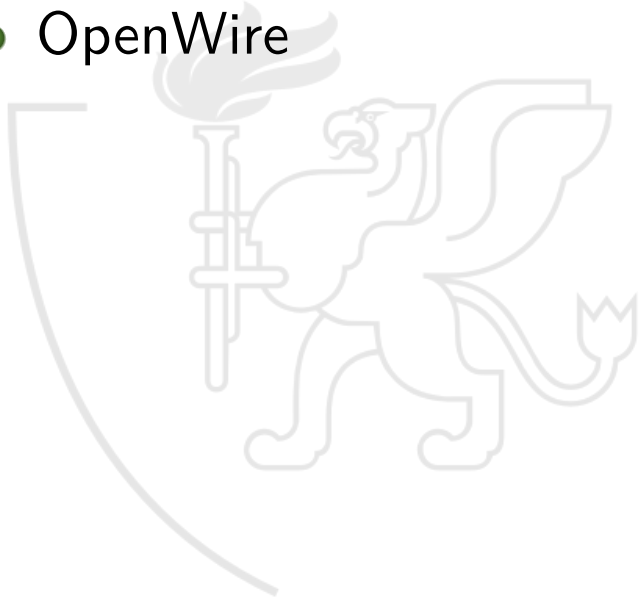
- Point-to-point
  - Queue (FIFO)
  - Egy fogadó
  - Garantált kézbesítés
  - Ack
  - Több küldő és több fogadó is lehet ugyanahhoz a queue-hoz (skálázhatóság)
- Publish-subscribe
  - Topic
  - Feliratkozók
  - Broadcast

Pattern-ek

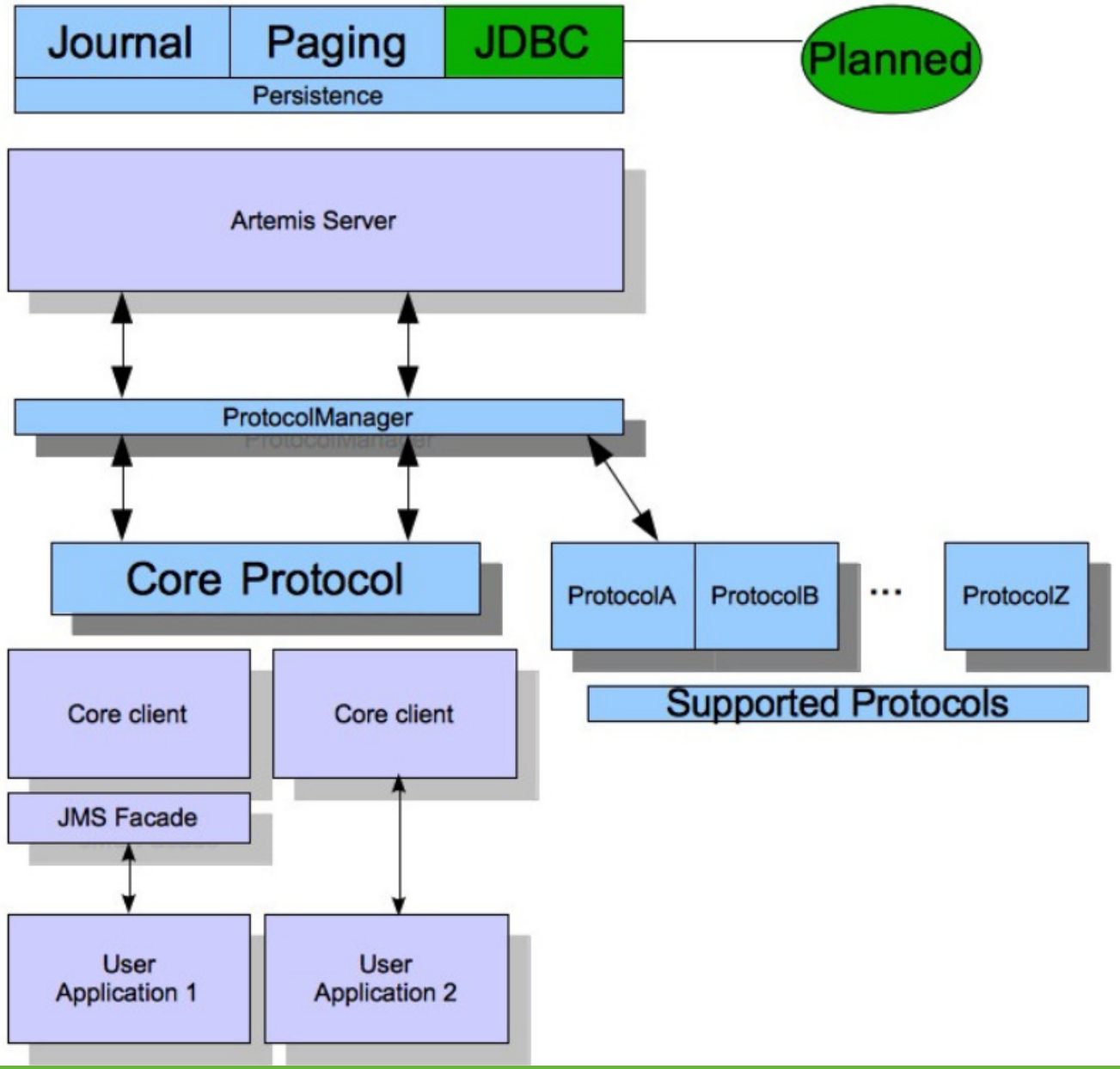
- Message Type patterns
- Message Channel Patterns
- Routing patterns
- Service Consumer Pattern
- Contract Pattern
- Message Construction Pattern
- Transformation Pattern

Messaging API-k

- JMS (Java Messaging Service)
- Rendszer specifikus API (Pl.: Apache ActiveMQ Artemis)
- Restful API-k
- AMQP (Advanced Message Queuing Protocol)
- MQTT
- STOMP
- OpenWire



JMS - Apache ActiveMQ (Artemis)

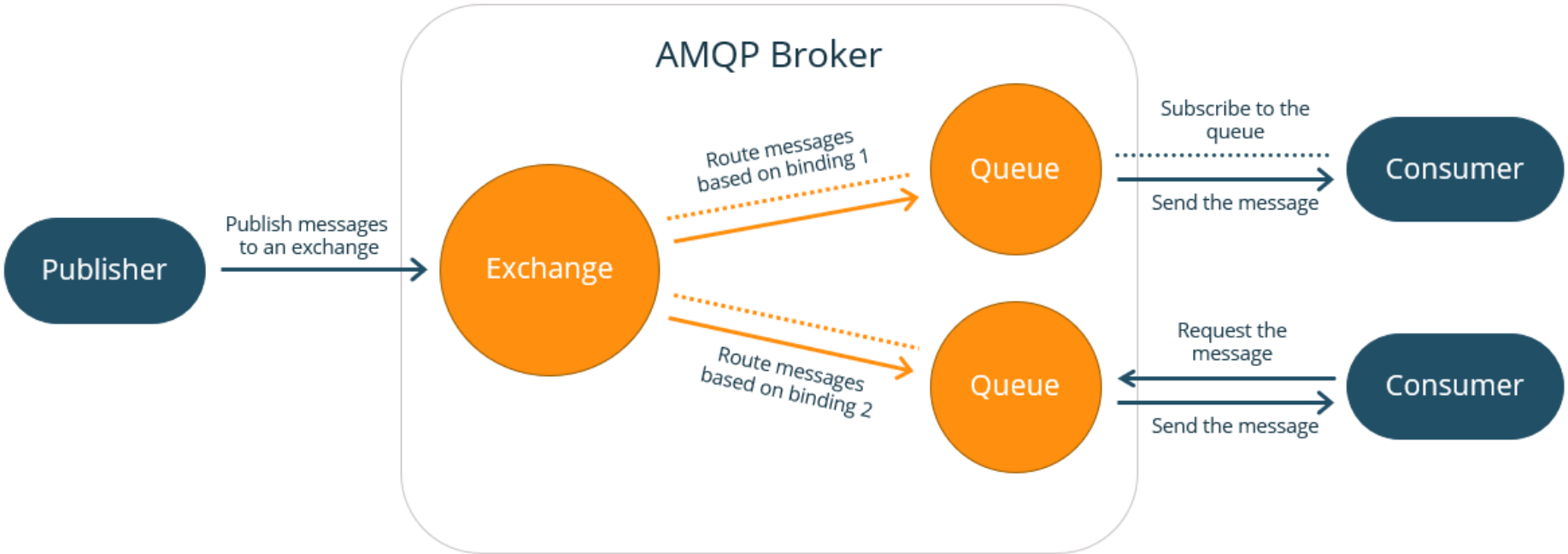


AMQP

- Nyílt szabvány
- Hálózati protokoll
- Átjárhatóság különböző vendorok (brokerek) között
- Wire level protokoll
- AMQP pre-1.0 (RabbitMQ)
- AMQP 1.0 és utána (Apache Qpid)

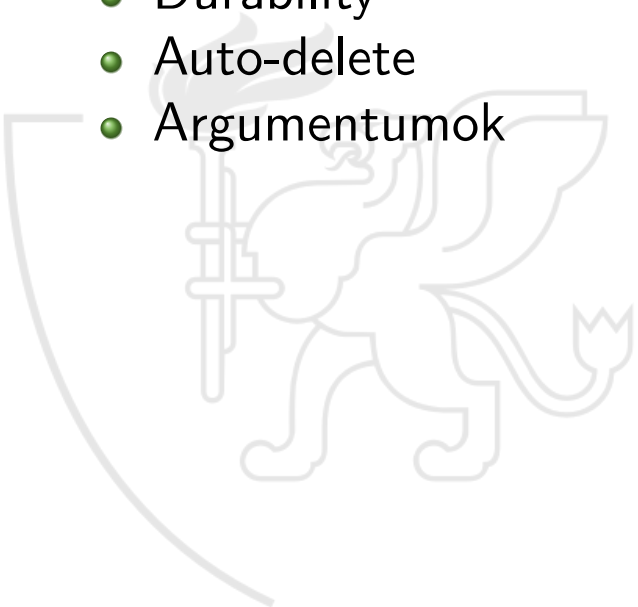
AMQP 0-9-1

- Exchange
- Queue
- Binding
- Push vs Pull alapú üzenettovábbítás
- Ack

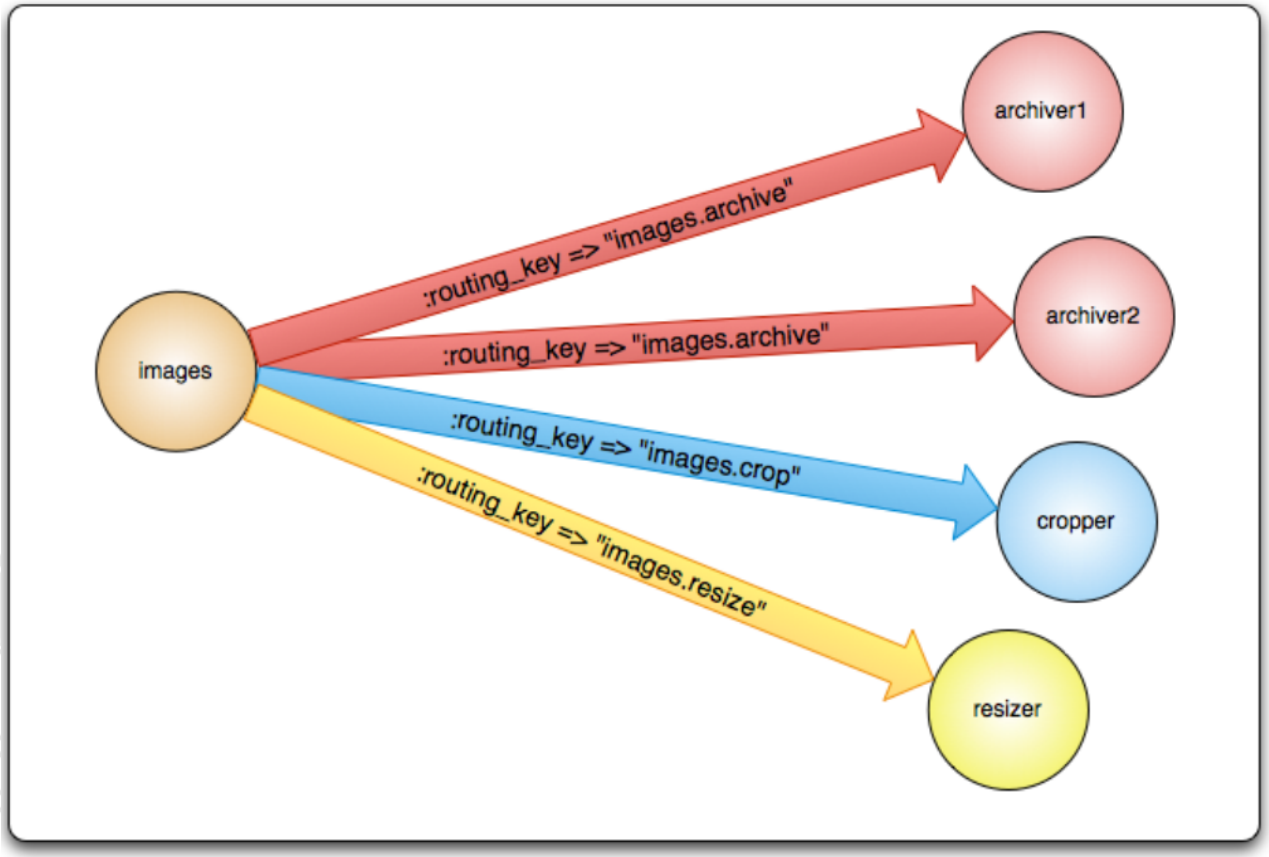


Exchange típusok és attribútumok

- Típusok
  - Direct
  - Fanout
  - Topic
  - Header
- Attribútumok
  - Name
  - Durability
  - Auto-delete
  - Argumentumok



Direct exchange

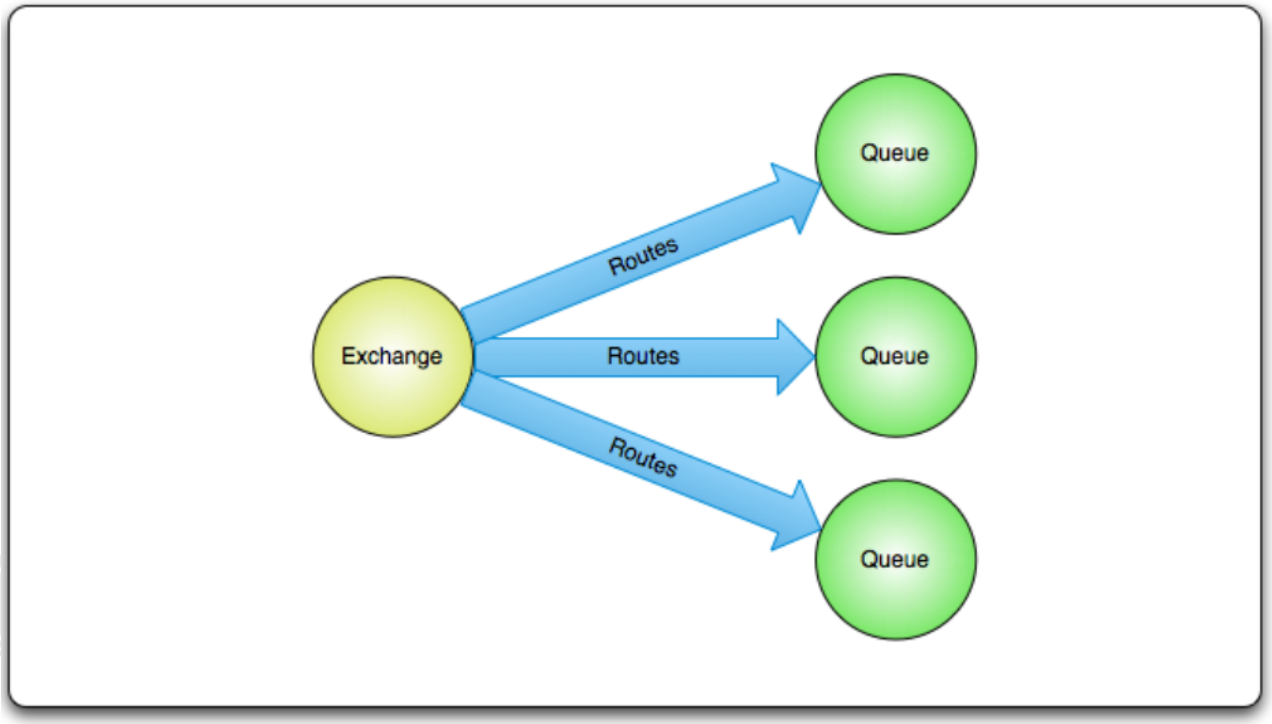


- Routing key alapú üzenettovábbítás
- Több worker között terhelés elosztás

Default exchange

- Direct exchange típusú
- Broker által létrehozott
- Nincs neve
- Minden queue hozzá van rendelve
- Routing key = queue neve
- Közvetlen queue-ba küldés szimulálása

Fanout exchange



- Routing key ignorálva
- Exchange összes route-ja megkapja az üzenetet

Topic és Header exchange

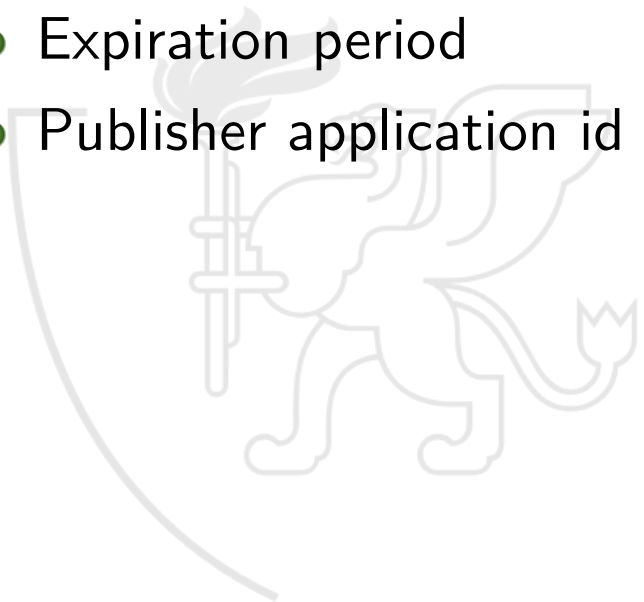
- Topic exchange
  - Routing key és pattern alapú továbbítás
  - Publish-subscribe megvalósítás
- Headers exchange
  - Header alapú üzenettovábbítás (nem routing key)
  - Több feltétel (egyszeri vagy együttes teljesülés)

Queue attribútumok

- Name
- Durable
- Exclusive
- Auto-delete
- Arguments

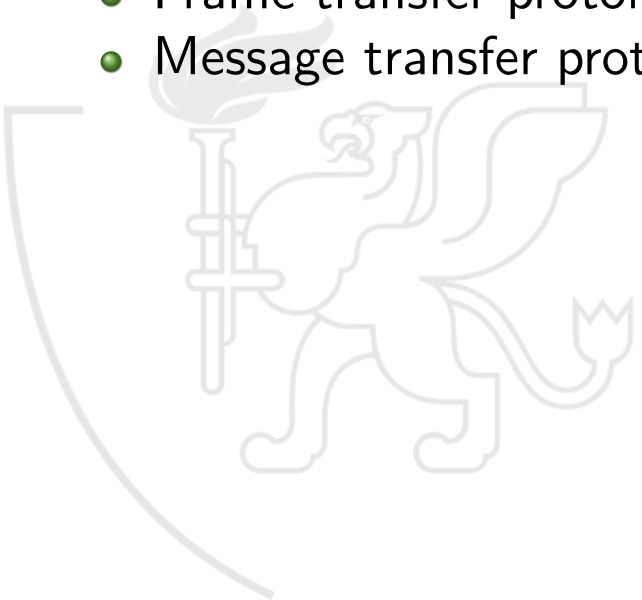
Message attribútumok

- Content type
- Content encoding
- Routing key
- Delivery mode (perzisztens vagy sem)
- Message priority
- Message publishing timestamp
- Expiration period
- Publisher application id



AMQP 1.0

- Nem kompatibilis az elődjével
- Broker belső felépítését teljesen ejti
- Nincs exchange, nincs queue
- ISO/IEC 19464:2014
- 3 rétegbe szerveződik
  - Transport and connection security protokoll
  - Frame transfer protokoll
  - Message transfer protokoll



Köszönöm a figyelmet!