CODESPRING
OUR CODE YOUR SUCCESS
20 years

Codespring mentorprogram
Enterprise platformok es keretrendszerek

Let's win the race together!

Enterprise alkalmazások Springgel

Dependency Injection, Spring Boot

Sulyok Csaba
sulyok.csaba@codespring.ro

1. rész

Inversion of Control

Könyvtár vs. keretrendszer: Inversion of Control

*Inversion of Control is a key part of what makes a **framework** different to a **library**. A library is essentially a set of functions that you can call, these days usually organized into classes. Each call does some work and returns control to the client.*

A framework embodies some abstract design, with more behavior built in. In order to use it you need to insert your behavior into various places in the framework either by subclassing or by plugging in your own classes. The framework's code then calls your code at these points.

Martin Fowler

`https://martinfowler.com/bliki/InversionOfControl.html`

Könyvtár vs. keretrendszer: Inversion of Control

- A keretrendszerek implementálják az *inversion of control* elvet: az alkalmazás kontrollját átadják a keretrendszernek, amely később meghívja az általunk megadott kódot.
- Az általunk írt kódot a keretrendszer konfigurációs állományokkal vagy annotációkkal találja meg.
- *Don't call us, we'll call you.*
- Pl. az `slf4j` (implementációja) egy **könyvtár**
  - Mi kérünk példányokat tőle factory tervezési mintával.
  - Mi hívjuk meg az általa definiált metódusokat.
  - Az alkalmazás kontrollja a mi kezünkben marad.
- Pl. a `Servlet` API és az őt implementáló webkonténer (pl. Tomcat) egy **keretrendszer**
  - **Nem** mi írjuk meg a HTTP hívások feldolgozását, az a keretrendszer dolga
  - Ő meghívja a mi kódunkat, melyet a `web.xml` konfigurációs állomány és a `@WebServlet` annotáció segítségével talál meg.

Spring

- Erőteljes, nyílt forráskódú, flexibilis Java **keretrendszer** enterprise webalkalmazások készítésére
- Egy *pehelysúlyú* alternatívaként indult a Java EE implementációira, de ez a különbség mára elmosódott
- Modulárisan használható - húzhatunk csak azon részeire függőséget, amelyre az alkalmazásunknak aktívan szüksége van
- Sok *low-level* tervezési mintát automatikusan implementál, általában reflection/annotáció-feldolgozás segítségével
  - (Abstract) factory különböző változatai
  - Többrétegű architektúrában elfoglalt pozició jelölése
  - Externalizáció/automatikus konfiguráció (properties, YAML)
  - Profilozás
  - Adatelérési réteg generálása (JPA/Hibernate segítségével)
  - Webalkalmazások (automatikus hibakezelés, szerializáció, validálás)

Spring Boot

- Mivel a Spring sok kicsi modulra bontja a logikáját, sok függőségre lehet szükség egy használható prototípus alkalmazás létrehozásához.
- Ezért a Spring nyújt előre definiált függőség-kombinációkat, amelyek segítenek különböző típusú alkalmazások felépítésében kevés függőséggel.
- Automatikus konfigurációkat nyújt, pl. tipikusan használt third-party függőségeket is behúz, pl.
  - Hibernate mint default JPA implementáció
  - *Embedded* Java webkonténer futtat (pl. Tomcat, Jetty) - nem szükséges kitelepíteni WAR állományt
- Gradle pluginnal is rendelkezik.
- Függőségek: `org.springframework.boot...` group

Spring Boot Starters

Elterjedtebb starter csomagok:

- `spring-boot-starter` - auto-konfigurációs támogatás, loggolás és YAML-formátumú konfigurációk olvasása
- `spring-boot-starter-data-jpa` - adatelérési réteg generálása JPA és Hibernate segítségével (az implementáció lecserélhető)
- `spring-boot-starter-web` - webalkalmazások készítésére: RESTful endpointok definiciója, embedded webszerver (alapértelmezetten Tomcat), sablonmotor használata (Web MVC)
- `spring-boot-starter-test` - alkalmazások tesztelésére alkalmas függőségek: JUnit, Hamcrest and Mockito
- `spring-boot-starter-security` - webalkalmazások biztonságáért felelős csomagok (pl. hitelesítés, engedélyezés)

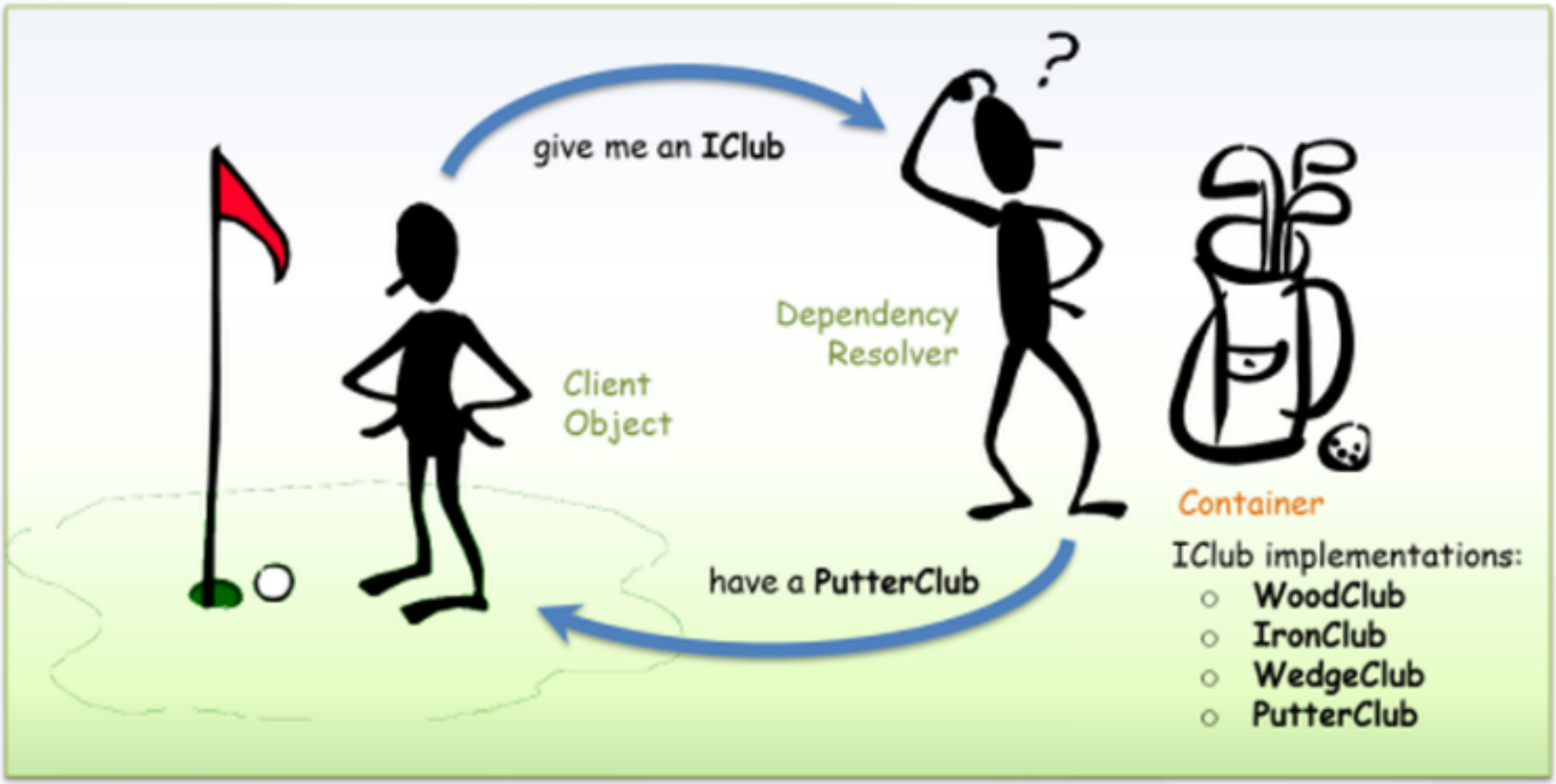
2. rész

Dependency Injection

Dependency injection

- A dependency injection (DI) egy *tervezési minta*, amely megvalósítja az *inversion of control* elvet.
- Ebben a kontextusban a *dependency* (függőség) fogalmat ne keverjük a Maven függőségekkel.
  - *Dependency* - egy osztálynak szüksége van egy másik osztály egy példányára, de nem szeretné ő karbantartani ennek a létrehozását, törlését, életciklusát, példányainak számát, konkrét implementációját, stb.
- A felhasználó csak jelzi egy függőség szükségét, s a DI-t implementáló keretrendszernek átadja a control flow-t
- *"Instead of an object looking up other objects that it needs to get its job done (dependencies), a DI container injects those dependent objects."*
- Legelterjedtebb Java DI keretrendszerek:
  - Spring
  - Weld - a *CDI (Contexts & Dependency Injection)* Java EE specifikáció referencia-implementációja
  - Google Guice

Dependency injection



forrás: `http://www.veloxcore.com/media/1241/dependency-injection-golf.png`

Spring Dependency Injection

- A Spring karbantart egy **bean container**t, amelyben pool formájában tárolja az injektálható objektumokat
- *Spring Bean* - Spring által menedzselt példány
- `BeanFactory`: a komponensek menedzsmentjéért felelős, egy `BeanFactory` interfészen keresztül kommunikálhatunk a Spring IoC konténerrel.
- `ApplicationContext`:
  - A `BeanFactory` interfész kiterjesztése. A DI szolgáltatásokon kívül további szolgáltatásokat biztosít, pl. tranzakció-kezelés, AOP, nemzetköziesítés, eseménykezelés stb.
  - Egy példánya erősen kötött egy Spring alkalmazás életciklusához
  - Konfigurálható kódból vagy konfigurációs állományból, de a Spring autokonfigurációs mechanizmusa automatikus példányt állít be.
  - Spring Boot autokonfigurációs mechanizmus: classpath scanninget alkalmaz a Spring Boot kontextusgyökerétől kiindulva, s keresi a megfelelően annotált osztályokat.

Spring Dependency Injection: `@Autowired`

- Egy függőség szükségességét az **`@Autowired`** annotációval jelezzük.
- Része a **`spring-beans`** csomagnak

```
public class Bean2 {
    @Autowired
    private Bean1 bean1;
}
```

- A Spring *futási időben* injektál egy megfelelő **`Bean1`** példányt a **`Bean2`** minden (Spring által menedzselt) példányába
- Az **`@Autowired`** Spring-specifikus, emellett támogatott a JSR-250-ben definiált **`@Resource`**, illetve a JSR-330-as **`@Inject`** is.
- Lehetséges injektálási pontok (ide helyezhetőek az **`@Autowired`** annotációk):
  - Adattag
  - Konstruktor
  - Setter metódus

Spring Beanek megjelölése

- Spring Beaneket deklarálhatunk a következőképpen:
  1. XML formátumú **`ApplicationContext`** konfiguráció esetén egy **`bean`** elemmel
  2. Egy classpath scanninget alkalmazó alkalmazásban egy szemantikus stereotype annotációt alkalmazunk az osztályon:
     - **`@Component`** - általános bean
     - **`@Repository`** (szemantikus) - adatelérési réteghez tartozó bean
     - **`@Service`** (szemantikus) - business logikát megvalósító bean
     - **`@Controller`** (szemantikus) - web erőforrások lekezeléséhez használatos bean
  3. Egy konfigurációs bean (**`@Configuration`**) tartalmazhat **`@Bean`**-nel annotált factory metódusokat.

Spring DI példa

ServiceInterface.java:

```
public interface ServiceInterface {
    void serviceMethod();
}
```

ServiceBean.java:

```
/**
 * Jelöljük, hogy ez az osztály injektálható.
 * Ha valamely szülő csomagban indul classpath scanning a @ComponentScan annotációval,
 * ez az osztály meg lesz találva.
 */
@Component
public class ServiceBean implements ServiceInterface {
    private static final Logger LOG = LoggerFactory.getLogger(ServiceBean.class);

    @Override
    public void serviceMethod() {
        LOG.info("I am performing a service");
    }
}
```

Spring DI példa

CODESPRING
OUR CODE YOUR SUCCESS
20 years

ClientBean.java:

```
@Component
public class ClientBean {
    private static final Logger LOG = LoggerFactory.getLogger(ClientBean.class);

    /**
     * Adattag-szinten megadott függőség.
     * Lehetne @Resource vagy @Inject is.
     * A konténer automatikusan megtalálja az egyetlen injektálható példányt.
     */
    @Autowired
    private ServiceInterface service;

    /**
     * Ez a metódus hívódik meg a DI konténer injektáló logikájának legvégén.
     * Minden függőségi bean garantálan be van állítva.
     */
    @PostConstruct
    public void postConstruct() {
        LOG.info("In @PostConstruct of ClientBean. Service is of type {}",
                service.getClass().getSimpleName());
        service.serviceMethod();
    }
}
```

Spring DI példa

- Konstruktorokban való logika alkalmazása nem ajánlott, mivel a függőségek még potenciálisan nincsenek beállítva.
- A Spring támogatja a JSR-250-ben definiált életciklus-kezelő annotációkat: `@PostConstruct`, `@PreDestroy`
- Az annotált metódusok lehetnek privátak is.
- A keretrendszer azután hívja meg a `@PostConstruct`-tal annotált metódust, amiután minden inicializálási logika lefutott a beanen és függőségein.
- Egy menedzselt bean tartalmazhat több metódust is az annotációval ellátva.

Spring Boot Application


```java
@SpringBootApplication
public class SpringDependencyInjectionDemo {
    public static void main(String[] args) {
        SpringApplication.run(SpringDependencyInjectionDemo.class, args);
    }
}
```

- `@SpringBootApplication`
  - Burkoló 3 másik Spring annotációra (convenience annotation):
    - `@Configuration`: factory-ként működik az adott osztály, ha `@Bean`-nel annotált metódusokat tartalmaz
    - `@ComponentScan`: a **jelen csomag**ban komponens szkennelést állít be, ezért mindig az alkalmazásunk legfelsőbb csomagjába tegyük
    - `@EnableAutoConfiguration`: JAR függőségek alapján különböző alapértelmezett beaneket és szolgáltatásokat tesz elérhetővé
- `SpringApplication`
  - Spring Boot által definiált inicializáló osztály. A következőket végzi el:
    - Létrehozza és karbantartja a megfelelő `ApplicationContext` példányt.
    - Ezen keresztül elvégzi a valid *singleton* beanek betöltését.
    - Parancssori konfigurációra ad lehetőséget.

Spring factory példa

ClientBean.java:

```
@Component
public class ClientBean {

    @Autowired
    Logger log;

    @Autowired
    ServiceBean serviceBean;

    @PostConstruct
    public void postConstruct() {
        log.info("In @PostConstruct of client bean");
        serviceBean.serviceMethod();
    }
}
```

Spring factory példa


```java
/**
 * Konfigurációs bean. Eszerint factory-ként dolgozik nem annotált bean típusoknak.
 */
@Configuration
public class ConfigurationBean {

    /**
     * Factory metódus. Visszatérítési típus alapján kezeli a Spring.
     * Minden injektálási pont külön példányt kap ezzel az implementációval.
     */
    @Bean
    public ServiceBean buildServiceBean() {
        return new ServiceBean();
    }

    /**
     * Külső (nem általunk definiált) objektum factory-ja.
     * Az opcionális injectionPoint paraméterből információt kapunk arról,
     * ahova az injektálás történik.
     */
    @Bean
    @Scope("prototype")
    public Logger buildLogger(InjectionPoint injectionPoint) {
        Class<?> clazz = injectionPoint.getField().getDeclaringClass();
        return LoggerFactory.getLogger(clazz);
    }
}
```

Spring Bean hatókörök

- Egy-egy Spring bean *hatóköre* befolyásolja, hogy a DI konténer hány példányt tart karban az illető osztályból.
- Meghatározza az életciklust és láthatóságot a kontextuson belül.
- A Spring 6 hatókört definiál:
  - `singleton` (*alapértelmezett*) - egy globális példány
  - `prototype` - új példány minden injekciós ponton
  - `request` - egy példány per HTTP kérés
  - `session` - egy példány per felhasználó
  - `application` - egy példány per `ServletContext` életciklus
  - `websocket` - egy példány per WebSocket session
- Az utolsó 4 csak *web-aware* Spring alkalmazások esetén definiált - amikor az autokonfigurációs mechanizmus az `ApplicationContext` egy megfelelő implementációt hozza létre.
- A bean definiciójánál fölülírható a `@Scope("value")` annotációval - `@RequestScope`, `@SessionScope` segéd-annotációk is alkalmazhatóak

Spring körkörös függőségek

- A körkörös függőségek megengedettek singleton beaneknek, mivel ezek referenciákat tárolnak egymásra.
- Prototype scoped beanek között nem lehet körkörös függőség.
- Példa:

```
@Component
@Scope("prototype")
class Bean1 {
    @Autowired Bean2 bean2;
}

@Component
@Scope("prototype")
class Bean2 {
    @Autowired Bean1 bean1;
}

@SpringBootApplication
public class CircularDep {
    @Autowired
    Bean2 bean2;

    // ... main ...
}
```

```
***************************
APPLICATION FAILED TO START
***************************

Description:

The dependencies of some of the beans
in the application context form a cycle:

   circularDep (field Bean2 CircularDep.bean2)
┌─────┐
|  bean2 (field Bean1 Bean2.bean1)
↑     ↓
|  bean1 (field Bean2 Bean1.bean2)
└─────┘
```

Spring DI: több implementáció

- Egy osztályt több bean definició is létrehozhat, mely esetben a DI konténernek választania kell a potenciális metódusok között.

```
@Configuration
public class ConfigurationBean {

    @Bean
    ServiceBean buildServiceBean1() {
        return new ServiceBean();
    }

    @Bean
    ServiceBean buildServiceBean2() {
        return new ServiceBean();
    }
}
```

```
***************************
APPLICATION FAILED TO START
***************************

Description:

Field serviceBean in ClientBean required a single bean,
but 2 were found:
    - buildServiceBean1: defined by method 'buildServiceBean1'
      in class path resource [ConfigurationBean.class]
    - buildServiceBean2: defined by method 'buildServiceBean2'
      in class path resource [ConfigurationBean.class]
```

Spring DI: több implementáció

Potenciális megoldások:

1. Valamelyik factory metódus primary-vé tétele:

```
@Configuration
public class ConfigurationBean {
    @Bean
    @Primary
    ServiceBean buildServiceBean1() { ... }
    // ...
```

2. Az egyik factory metódus **nevét** egyenlővé tenni az injekciós pontbéli adattaggal:

```
@Configuration
public class ConfigurationBean {
    @Bean
    ServiceBean myServiceBean() { ... }
    // ...
```

```
@Component
public class ClientBean {
    @Autowired
    ServiceBean myServiceBean;
    // ...
```

3. `@Qualifier`-ek használata - nevet adnak a készített beannek

```
@Configuration
public class ConfigurationBean {
    @Bean
    @Qualifier("serviceBean1")
    ServiceBean buildServiceBean1() { ... }
    // ...
```

```
@Component
public class ClientBean {
    @Autowired
    @Qualifier("serviceBean1")
    ServiceBean myServiceBean;
    // ...
```

4. **Profilonkénti szétválasztás** (következő példa)

Spring alkalmazásszintű konfiguráció

- A Spring számos globális beállítást támogat.
- Ezeknek beolvasási sorrendje (prioritási sorrend):
  1. Command line JVM opcióként (pl. `-Dlogging.level.root=DEBUG` a Java parancs hívásánál)
  2. Globális konfigurációs állomány - a Spring alapértelmezetten keresi a classpath gyökerében az `application.properties` vagy `application.yml` állományt ezen beállításokért
  3. Beépített alapértelmezett értékek - a *convention over configuration* elv megtartásáért.
- Lehetséges beállítások listája

Spring Properties példa

- Konfigurációs állományokat automatikusan beolvashatunk egy konfigurációs osztályba a `@PropertySource` annotáció megadásával.
- Ennek értéke: `classpath:útvonal_a_classpathen.properties`
- Az értékeket a következő formában érhetjük el:
  1. Injektálhatunk egy `PropertyResolver` vagy `Environment` típusú beant, amely hasonló a korábbi `PropertyProvider` megvalósításunkhoz (property-ket enged lekérni, s profilozást is támogat)
  2. Egy-egy property értékét injektálhatjuk a konfigurációba a `@Value` annotációval - ennek értéke expression language (`${kulcs}`) formájában tartalmazza a keresett kulcsot.

`src/main/resources/config.properties`:

```
# Automatikusan olvassuk ezt a file-t egy beanbe
key1=String value in props
key2=42
```

Spring Properties példa: `PropertyResolver`


```java
/**
 * Konfigurációs osztály, amely tartalmazhat kintről beolvasott értékeket.
 * A PropertyResolver injektálásával egy Properties-szerű konténert kapunk minden értékkel.
 */
@Configuration
@PropertySource("classpath:/config.properties")
public class PropertiesWithResolverBean {

    @Autowired
    private PropertyResolver resolver;

    public String getProperty(String key) {
        return resolver.getProperty(key);
    }
    public String getProperty(String key, String defaultValue) {
        return resolver.getProperty(key, defaultValue);
    }
    public <T> T getProperty(String key, Class<T> targetClass) {
        return resolver.getProperty(key, targetClass);
    }
    public <T> T getProperty(String key, Class<T> targetClass, T defaultValue) {
        return resolver.getProperty(key, targetClass, defaultValue);
    }
}
```

Spring Properties példa: `@Value`


```java
/**
 * Konfigurációs osztály, amely tartalmazhat kintről beolvasott értékeket.
 * A @Value annotációkkal konkrét kulcsokat keres az állományban.
 */
@Configuration
@PropertySource("classpath:/config.properties")
public class PropertiesWithValuesBean {
    // properties-ből való betöltés
    @Value("${key1}")
    private String key1;

    // működik különböző típusokra is
    @Value("${key2}")
    private Integer key2;

    // nem létező kulcs esetén alapértelmezett érték megadása
    @Value("${key3:defaultValue}")
    private String key3;

    // ... getterek ...
}
```

Spring Properties példa

CODESPRING
OUR CODE YOUR SUCCESS
20 years

```
@Component
public class ClientBean {
    private static final Logger LOG = LoggerFactory.getLogger(ClientBean.class);

    @Autowired
    private PropertiesWithValuesBean propertiesWithValues;

    @Autowired
    private PropertiesWithResolverBean propertiesWithResolver;

    @PostConstruct
    public void postConstruct() {
        LOG.info("In @PostConstruct of ClientBean");
        LOG.info("* properties via values:");
        LOG.info(" - key1 = {}", propertiesWithValues.getKey1());
        LOG.info(" - key2 = {}", propertiesWithValues.getKey2());
        LOG.info(" - key3 = {}", propertiesWithValues.getKey3());
        LOG.info("* properties via environment:");
        LOG.info(" - key1 = {}", propertiesWithResolver.getProperty("key1"));
        LOG.info(" - key2 = {}", propertiesWithResolver.getProperty("key2", Integer.class));
        LOG.info(" - key3 = {}", propertiesWithResolver.getProperty("key3", "defaultValue"));
    }
}
```

Spring profilok

- A Spring natívan támogatja a profilozást. A profilok aktívvá tétele a **`spring.profiles.active`** kulcs beállításán keresztül történik.
- Több profil is lehet aktív - a fenti beállítás tartalmazhat több profilnevet tömbben.
- Spring beanek és konfigurációk annotálhatóak a **`@Profile`** annotációval, mely befolyásolja érvényességüket.
- A kiolvasott globális konfigurációs állomány is kiterjesztődik a beállított profil alapján, pl. ha a **`dev`** profil van beállítva, mind az **`application.properties`**-t, mind az **`application-dev.properties`**-t beolvassa és alkalmazza a rendszer.
- A fő konfig állományban is állíthatjuk a profilt.

`application.properties`:

```
# Globális beállításai az alkalmazásnak
# Aktív profil beállítása
# - ezáltal az application-dev.properties
#   is be lesz olvasva
spring.profiles.active=dev
```

`application-dev.properties`:

```
# Csak dev profilon belüli beállítások
logging.level.root=DEBUG
```

Spring profilok példa


```java
/**
 * Csak dev profilban érvényes ez a bean. @Configuration -ekre is alkalmazható.
 */
@Component
@Profile("dev")
public class DevServiceBean implements ServiceInterface {
    private static final Logger LOG = LoggerFactory.getLogger(DevServiceBean.class);
    @Override
    public void serviceMethod() {
        LOG.info("I am in development mode service bean");
    }
}

/**
 * Ez a bean érvényes ha NEM dev profilban vagyunk.
 */
@Component
@Profile("!dev")
public class ProdServiceBean implements ServiceInterface {
    private static final Logger LOG = LoggerFactory.getLogger(ProdServiceBean.class);
    @Override
    public void serviceMethod() {
        LOG.info("I am in production mode service bean");
    }
}
```

Spring profilok példa


```
@Component
public class ClientBean {
    private static final Logger LOG = LoggerFactory.getLogger(ClientBean.class);

    // profil szerint más példányt kapunk
    @Autowired
    private ServiceInterface service;

    @PostConstruct
    public void postConstruct() {
        LOG.debug("In @PostConstruct of ClientBean");
        service.serviceMethod();
    }
}
```

- A fenti kliens bean nem tartozik egy bizonyos profilhoz, de viselkedése változik, mivel különböző példányt fog kapni a beanből profil szerint.
- A **`@Profile`**-t alkalmazhatjuk konfigurációkra is, pl. különböző DAO factory beant tehetünk aktívvá profil szerint.

Gyakorlatok

1. A Spring keretrendszer segítségével vezessük be a dependency injection tervezési mintát a alprojektünkbe.
   - Vezessük be a Spring Boot Starter függőséget és Gradle plugint.
   - Készítsünk megfelelő indító osztályt (`@SpringBootApplication`).
   - Az osztályok közötti függőségeket **injektáljuk**, pl. a `Servlet`be és `AbstractTableModel`be injektáljuk a DAO-t.
2. Alakítsuk a DAO factory implementációinkat `@Configuration`-ekké, amelyek injektálhatóvá teszik a DAO-kat. Mivel legalább 2 lesz belőlük, alkalmazzunk *profilonkénti megkülönböztetést* a factory-k között. Így az adatelérési módszer az `application.properties` `spring.profiles.active` kulcsa alapján dől el.
3. **Otthoni:** A DAO-k függőségeit (`EntityManager`, `ConnectionPool`) is dependency injectionnel töltsük be.
4. **Otthoni:** A JDBC kapcsolatok paramétereit továbbra is töltsük properties állományból, de alkalmazzuk a `@PropertySource` segítette automatikus betöltést.