Bevezetés

Üdvözöllek a *Microservice alkalmazás felépítése Spring Boot keretrendszerrel Docker környezetben* képzésen!

A képzés elméleti és gyakorlati videókból, valamint gyakorlati feladatokból áll. Először nézd meg az egy leckéhez tartozó elméleti és gyakorlati videót. Ha úgy találod hasznosnak, a gyakolati videón látottakat magad is megcsinálhatod.

Majd végezd el a gyakorlati feladatokat, melyek megtalálhatóak a GitHubon, a https://github.com/Training360/javax-mcr-public címen. Ugyanezen a címen elérhetitek a videón szereplő gyakorlati feladatok forrását, valamint a szükséges alkalmazások forráskódját, melyhez integrálódni lehet.

Spring Boot bevezetés

Ebben a fejezetben megismerheted, milyen okok és körülmények vezettek oda, hogy a Spring Framework létrejöjjön, és ezután mi volt az oka annak, hogy az erre épülő Spring Bootot is létrehozták. Szót ejtünk arról, hogyan hozhatsz létre egy Springes alkalmazást, és részletesen megismerheted annak belső működését és alapvető építőköveit, a Spring Beaneket. Megnézzük, hogy ezeknek mi az alapértelmezett konfigurációja, illetve hogyan konfigurálhatod őket te magad. Kitérünk arra, hogyan build-elheted le az alkalmazásodat Maven vagy Gradle használata esetén. Szóba kerül, hogyan írhatsz unit és integrációs teszteket, hogyan használhatod a fejlesztési folyamatot megkönnyítő Developer Tools-t, hogyan indíthatod el a Springes alkalmazásod Docker konténerben és végül hogyan töltheted azt fel a GitHubra. Külön lecke szól arról, milyen ajánlásokat tartalmaz a felhőbe telepíthető és üzemeltethető alkalmazások fejlesztéséhez a Twelve-Factor App.

Bevezetés a Spring Framework és Spring Boot használatába

A **Spring Framework** egy keretrendszer nagyvállalati alkalmazások fejlesztésére. Keretrendszer azért, mert az általunk írt komponenseket ő maga hozza létre, azaz példányosítja az általunk írt osztályokat. Ezt úgy is szokás mondani, hogy keretrendszer az, amely az általunk írt komponensek életciklusát vezérli.

Nagyvállalati alkalmazásnak azokat az alkalmazásokat nevezzük a Java világában, amelyek számára nem elegendőek a *Java Standard Edition*-be beépített tulajdonságok, hanem valamilyen további tulajdonságokra van szükség. A Spring Framework bizonyos problémákra saját megoldást biztosít, de ez a ritkább, mert a legtöbb esetre már létező third party library-ket integrál egy egységes modellbe.

Felmerülő igények, problémák egy nagyvállalati alkalmazás fejlesztése esetén, amelyek a Java SE-ben nem, vagy csak túl nagy munka árán megvalósíthatóak:

- A komponenseket nem a programozónak kell példányosítania és figyelni az életciklusukat, hanem a keretrendszer ezt elvégzi helyette.
- Fontos a távoli elérés másik számítógépről, interneten keresztül, sokszor böngészőből.
- Elvárás, hogy a keretrendszer maga biztosítsa a többszálúságot, és ne a programozónak kelljen azzal foglalkoznia, hogy hogyan tudja az alkalmazást egyszerre több felhasználó is párhuzamosan használni. Vagyis a programozó programozhasson úgy, mintha az alkalmazásnak csak egy felhasználója lenne, a többit pedig a keretrendszer oldja meg helyette.
- A perzisztencia problémakörének kezeléséhez a Java SE-ben csak a JDBC található, amely már elavult technológia. Egyrészt nagyon sok kódot kell írni, hogy adatbázist tudjunk kezelni vele ("bőbeszédű"), másrészt gyakorlatilag minden egyes metódushívásnál foglalkozni kell a kötelezően kezelendő `SQLException` kivételel (*checked exception*).
- Fontos a tranzakciókezelés, hogy tudjunk atomi műveleteket definiálni. Fontos továbbá az elosztott tranzakciókezelés, ahol vannak olyan műveletek, amelyek egyszerre két erőforrással is dolgoznak. Ilyenek például a két adatbázison átívelő tranzakciók, amikor az egyik műveletet az egyik, másik műveletet a második adatbázisban kell elvégezni.
- Elvárás az aszinkron üzenetküldés, tipikusan valamilyen *message oriented middleware* használata, azaz hogy egy alkalmazás üzeneteket tudjon küldeni egy másik alkalmazás számára. Azért aszinkron, mert nem várja meg a választ, hanem *"send-and-forget"* modellben működik.
- Fontos az ütemezés (*scheduling*), hogy bizonyos műveletek valamilyen időközönként fussanak le automatikusan, és ne kelljen ehhez felhasználói beavatkozás.
- Egy nagyvállalati alkalmazás tipikusan nem önmagában létezik, hanem más alkalmazásokhoz kell integrálódnia. Egy keretrendszernek kell tudnia támogatni az integráció bizonyos módjait, azaz az alkalmazások közötti kommunikációt.
- Elvárás az auditálhatóság, hogy vissza lehessen nézni, hogy ki, mikor, milyen műveletet hajtott végre az alkalmazáson belül.
- Fontos a konfigurálhatóság, azaz hogy az alkalmazás több különböző környezetben is tudjon futni (tesztkörnyezet, éles környezet, stb.), és ezekhez különböző beállításokat lehessen megadni.
- Az alkalmazás üzemeltethető legyen, vagyis igény az, hogy az alkalmazás naplózható, monitorozható legyen, valamint hogy az üzemeltetők be is tudjanak avatkozni, ha erre van szükség.
- Fontos a biztonság, az autentikáció és az autorizáció, vagyis hogy valamilyen módon be lehessen jelentkezni, illetve hogy a felhasználók csak a nekik rendelt szerepkörök alapján tudjanak műveleteket végezni.
- Az alkalmazást le lehessen fedni automatizált tesztekkel, ezáltal csökkentve a hibák számát. A tesztelhetőség által az alkalmazás architektúrája is könnyebben megérthetővé, áttekinthetővé tud válni.

A Spring Framework ahhoz, hogy ezeket a nagyvállalati igényeket kielégítse, a következő tulajdonságokkal rendelkezik:

- A fejlesztőnek a Spring Frameworkben egyszerű Java objektumokat kell implementálnia (*POJO – Plain Old Java Object*), ezek a komponensek. A komponensek a konténer felügyelete alatt állnak, az vezérli az életciklusukat, az példányosítja le őket, és az is engedi el a referenciát akkor, amikor már nincs rá szükség. A konténer neve a Spring Frameworkben **application context**.
- A komponensek nem önmagukban léteznek, hanem közöttük kapcsolatok vannak, ezeket is a konténer felügyeli. A Spring Framework valójában egy *Dependency Injection* vagy *Inversion of Control* keretrendszer.
- A konténer a komponensek és kapcsolataik leírására, a konfiguráció létrehozására három különböző módot használ: XML, annotáció és Java kód.
- A Spring Framework támogatja az aspektusorientált programozást, azaz bizonyos műveleteket egy helyre össze lehet gyűjteni és aztán a konfigurációval lehet bekapcsolni őket, ilyenek a biztonság, a naplózás, a tranzakciókezelés, stb.
- A Spring Framework nem mindent önmaga implementál, hanem a leggyakrabban használt, legelterjedtebb third party library-ket integrálja egy egységes modellbe úgy, hogy azután ezeket nagyon egyszerű legyen használni és konfigurálni. A Springet ezért *glue kód*-nak is szokták nevezni.
- A Spring Framework egyik fő jellemzője az ún. *boilerplate kódok* eliminálása abból a célból, hogy a fejlesztőnek ne a technológiai problémák megoldásával kelljen töltenie az időt, hanem csak az alkalmazás üzleti logikájára koncentrálhasson. Ezek olyan kódrészletek, amelyeknek önmagukban nincs túl nagy hasznuk, és a fejlesztő csak azért írja őket, mert a keretrendszer, 3rd party library megköveteli.
- A konténert (application context) úgy kell elképzelni, hogy vannak benne különböző komponensek, amelyek között fennállnak bizonyos kapcsolatok. A fejlesztő implementálja, a konténer pedig példányosítja őket és beállítja a közöttük lévő kapcsolatokat.
- A Spring tipikusan háromrétegű webes alkalmazások fejlesztésére való, erre használják a legszélesebb körben. A három réteg:
  - A legalsó réteg a perzisztens réteg, amely az adatbázissal tartja a kapcsolatot, ezt a Spring **Repository** rétegnek hívja.
  - A középső réteg az üzleti logika réteg, ahol az üzleti entitások és az üzleti folyamatok vannak implementálva, ezt a Spring **Service** rétegnek hívja.
  - A legfelső réteg a felhasználói felület réteg, a *user interface*, azaz prezentációs réteg, amely a felhasználóval tartja a kapcsolatot, ezt a Spring **Controller** rétegnek hívja.
- A Spring Framework része a **Spring MVC**, amely egy Spring gondolatiságához illeszkedő webes keretrendszer.
- A Spring nem kezeli a HTTP-t, ennek kezelését egy webkonténerre bízza (pl. Tomcat, Jetty).

Bevezetés a Spring Framework és Spring Boot használatába

A feladat nem megy a 10-es és 11-es Jetty-vel. Használható verzió:

```
<plugin>
  <groupId>org.eclipse.jetty</groupId>
  <artifactId>jetty-maven-plugin</artifactId>
  <version>9.4.39.v20210325</version>
</plugin>
```

Bevezetés a Spring Boot használatába

A Spring Framework egy viszonylag régebb óta létező keretrendszer, a Spring Boot sokkal fiatalabb. Azért hozták létre, mert észrevették, hogy a Spring Frameworkben egy alkalmazás konfigurálása, a beanek testreszabása és összekapcsolása, illetve a különböző third party library-k integrálása elég sok plusz munkával jár. A **Spring Boot**-ban ezeket már előre megírták helyettünk. Kitalálták az ún. *autoconfiguration*-t, ami azt jelenti, hogy a Spring Boot induláskor megvizsgálja, hogy milyen third party library-k vannak a classpath-on, és automatikusan konfigurálja ezeket, méghozzá ún. intelligens alapértékekkel, vagyis a leggyakrabban, legáltalánosabban használt konfigurációkkal. Ettől természetesen el is lehet térni, ha ez valamiért szükséges vagy fontos. A Spring Boot nem egy új keretrendszer, hanem a Spring Frameworkre épül rá és annak a használatát könnyíti meg.

A Spring Boot képes az alkalmazást *self-contained*-ként létrehozni. Ez azt jelenti, hogy maga az alkalmazás tartalmazza a futáshoz szükséges konténert is.

A Spring Boot további előnye a nagyvállalati üzemeltethetőség, vagyis hogy egy üzemeltető információkat tud kinyerni az alkalmazásból (monitorozás) és akár be is tud avatkozni.

Létrehoztak egy https://start.spring.io/ weboldalt, ahol le lehet generálni egy üres alkalmazást, meg lehet adni a projekt Maven koordinátáit, majd ezekkel legenerálódik számunkra egy ZIP állomány, amelyet kicsomagolva és a fejlesztőeszközben megnyitva máris elkezdődhet a fejlesztés.

A Spring Boot moduláris, nekünk csak a számunkra szükséges modulokat kell felvennünk külső függőségként.

A Spring Boot különösen alkalmas kis alkalmazások, *microservice*-ek fejlesztésére, hiszen önmagában is futtatható állományt kapunk, amely mindent tartalmaz az alkalmazás működéséhez, ráadásul könnyen telepíthető és üzemeltethető is.

Ha a feljebb említett weboldalon egy Maven projektet generáltatunk, akkor a letöltött ZIP állományban egy klasszikus Maven projektstruktúra található. Az `src/main/resources` könyvtáron belül található egy kiemelt konfigurációs állomány, az `application.properties`. A Spring Boottal kapcsolatban mindent ebben az állományban kell konfigurálni, ez egyébként a projekt generálásakor üres. Két könyvtárat érdemes még megemlíteni: a `static`, ebbe kerülnek webes alkalmazás esetén a statikus állományok, amelyek nem módosulnak, vagyis a HTML-, CSS- és JavaScript-fájlok, illetve a `templates`,

ebbe pedig template-ek kerülnek, amennyiben szerver oldali HTML-generálást akarunk. Két előre legenerált osztály az `Application.java` végű, amellyel el lehet indítani az alkalmazást, és a `Tests.java` végű, amely egy nagyon egyszerű üres tesztesetet tartalmaz, amit elindítva azt lehet megvizsgálni, hogy a Spring jól van-e bekonfigurálva, az application context egyáltalán elindul-e. (A projekt generálásakor az alkalmazás nevét tartalmazó prefix kerül az osztályok neve elé, pl. ha az alkalmazás neve `employees`, akkor `EmployeesApplication.java` és `EmployeesTests.java`.)

A `pom.xml`-be a következő komponensek kerülnek be:

- A `pom.xml` egy másik projektnek lesz a leszármazottja, ezért egy *parent projekt* lesz benne definiálva: `org.springframework.boot:spring-boot-starter-parent`. Ide vannak felvéve és innen öröklődnek a függőségek, a verziószámokkal együtt, ezért nem kell őket megadnunk.
- Ha kiválasztjuk, hogy egy webes alkalmazást szeretnénk generálni, akkor eleve felveszi nekünk függőségként a *Tomcat*-et, ami a default konténer lesz, felveszi a *Jackson*-t, ami a Java objektumok és a JSON dokumentumok közötti megfeleltetésért felelős.
  > A *Hibernate Validator* újabb verziókba már nem kerül bele.
- Teszthez szükség van a Spring Bootnak az `org.springframework.boot:spring-boot-starter-test` moduljára, amely nagyon sok függőséget felvesz: *JUnit 5*, *Mockito*, *AssertJ*, *Hamcrest*, *XMLUnit*, *JSONassert*, *JsonPath*.

Spring Beanek

A Spring alkalmazás alapvető építőkövei a **Spring Bean**-ek, amelyek egyszerű Java osztályok. Ezeket a fejlesztő hozza létre. A Spring keretrendszer tud róluk és ő példányosítja le ezeket, nem a fejlesztőnek kell ezt megtenni. A Spring Beanek között természetesen lehetnek függőségek és ezeknek a függőségeknek a beállítása is a keretrendszer feladata. Mindezeket a Springnek valamilyen konfigurációval lehet megadni. A Spring Beanek **POJO**-k, azaz **Plain Old Java Object**-ek, tehát semmiféle különös elnevezési konvenciónak nem kell megfelelniük, semmilyen interfészt nem kell implementálniuk. Az application contexten belül helyezkednek el, a konténer vezérli az életciklusukat. Ez azt jelenti, hogy a konténer példányosítja őket, valamint engedi el a referenciát akkor, ha már nincs rájuk szükség, hogy a *garbage collector* be tudja őket gyűjteni.

A Spring Beanek nem ömlesztve vannak, hanem tipikusan rétegekbe rendezve helyezzük el őket. A legalsó, perzisztens rétegben vannak a repository beanek, a középső, üzleti logika rétegben a service-ek és a legfelső, felhasználói rétegben a controllerek.

A Springben alapértelmezetten a singleton tervezési mintát használják, ezért minden bean egyszer és csak egyszer jön létre. Amikor egy webes alkalmazás esetén bejön egy HTTP-kérés, akkor ugyanez a példány kerül megszólításra. Ez abban az esetben lehet probléma, ha a bean tárol valamilyen állapotot, ezért az a legegyszerűbb, ha nem definiálunk benne állapotot tároló attribútumot, csak pl. más beanekre történő hivatkozásokat.

Ha az alkalmazás jar típusú, akkor a belépési pontja a `main()` metódus. Amikor a `https://start.spring.io` oldalon legeneráltatunk egy alkalmazást, akkor létrejön egy `Application` osztály, rajta van a `@SpringBootApplication` annotáció, és van egy `main()` metódusa, ezzel lehet elindítani az alkalmazást.

```
@SpringBootApplication
public class EmployeesApplication {

  public static void main(String[] args) {
    SpringApplication.run(EmployeesApplication.class,
      args);
  }

}
```

A `@SpringBootApplication` maga egy ún. metaannotáció, azaz rajta is egy csomó annotáció van:

- Ezek közül az egyik az `@EnableAutoConfiguration`, amely azt mondja, hogyha a classpath-on talál valamilyen library-t, amihez van konfigurációja, akkor azt automatikusan konfigurálja föl, automatikusan tegye elérhetővé a fejlesztő számára.
- A `@SpringBootConfiguration` annotáción rajta van a `@Configuration` annotáció, ami azt jelenti, hogy ebben az osztályban is tudunk konfigurációt Java kóddal megadni. Tehát ha nekünk valamiért nem jók az intelligens alapértékek, vagy saját magunk szeretnénk valamit beállítani, akkor ezt ebben az osztályban megtehetjük.
- A `@ComponentScan` annotáció azt jelenti, hogy automatikusan felolvassa azokat az osztályokat, amelyeken rajta van a következő annotációk valamelyike: `@Component` jelzi az általános objektumokat, `@Repository` a perzisztens rétegbe tartozó objektumokat, `@Service` az üzleti logika rétegbe tartozó objektumokat és a `@Controller` pedig a user interface rétegbe tartozó objektumokat. Ez utóbbi mellett megjelent a `@RestController` annotáció, mely a `@Controller` annotációt egészíti ki további funkcionalitással.

A *Controller* komponensek egyszerű Java osztályok (POJO-k), a Spring Framework részeként megvalósított Spring MVC részei. Arra valók, hogy a felhasználótól a bejövő kéréseket értelmezzék, és azokra válaszoljanak. Rajtuk van a `@Controller` annotáció. Ezen osztályok forráskódjában is elég gyakran használatosak a különböző annotációk, ugyanis ezekkel tudunk konfigurálni különböző dolgokat, például hogy milyen URL-en lehessen ezeket meghívni, milyen kéréseket vár, milyen válaszokat ad vissza, kell-e ezeket validálni, stb. Nem feltétlenül van Servlet API függősége (bár a Spring MVC nagyon erősen támaszkodik a Servlet API-ra). A legjobb gyakorlat az, hogyha kerüljük a Servlet API használatát és legfeljebb a Springes osztályokra függünk, vagy lehetőleg még azokat se használjuk, hanem kizárólag a Java Standard Editionben meglévő osztályokat és a Springes annotációkat. Az ezen osztályokban található metódusok elnevezésére nincs különösebb konvenció és ezek paraméterezése is flexibilis lehet. A Controllereket kétféleképpen szoktuk használni. Abban az esetben, hogyha olyan alkalmazást írunk, amelyek a HTML tartalmat a szerveroldalon állítják elő, akkor a metódusok egy templatenek (sablonnak) a

nevét adják vissza, amelyben az értékeket később a Spring cseréli ki. RESTful webszolgáltatások esetén viszont a visszaadott objektumot megpróbálják leképezni valahogy a HTTP response-ba, tipikusan JSON vagy XML formátumban.

Milyen annotációkat szokás gyakran használni ezekben az osztályokban?

- `@Controller`: ezt kell rátenni ezekre az osztályokra azért, hogy a component scan megtalálja és felolvassa őket, vagyis hogy a Spring MVC egyáltalán tudjon róluk.
- `@RequestMapping`: ezzel tudjuk megmondani, hogy milyen URL-en figyeljen az adott metódus. Ezt meg lehet adni osztályszinten is, ekkor az összes metódusra vonatkozik. Ha mindkét helyen szerepel (osztály- és metódusszinten), akkor konkatenálódik, tehát összeadódik. Az URL-t meg lehet adni *wildcard* karakterek használatával is (például *), illetve a HTTP metódus megadható a `method` paraméterrel is.
- `@ResponseBody`: ha egy metódusból visszatérünk, azt egy template-névként próbálja a Spring értelmezni. Akkor kell a metódusra rátenni ezt az annotációt, ha meg akarjuk mondani a Springnek, hogy a visszatérési értéket alakítsa át JSON formátumúvá.

```
@Controller
public class EmployeesController {

    @RequestMapping("/")
    @ResponseBody
    public String helloWorld() {
        return "Hello World!";
    }
}
```

A *Service* komponensekre rá kell tenni a `@Service` annotációt.

```
@Service
public class EmployeesService {

    public String helloWorld() {
        return "Hello World at " + LocalDateTime.now();
    }
}
```

Nézzük meg, hogyan lehet hivatkozni az egyik komponensből a másik komponensbe. Kötelező betartani azt, hogy csak a magasabb szinten elhelyezkedő osztály láthat rá az eggyel alacsonyabb szinten elhelyezkedő osztályra. Azaz a Controller csak a Service réteget látja, a Service pedig csak a Repository réteget. A Controller nem hívhatja a Repository-t, és az előzőekben leírt kapcsolatok fordítva sem történhetnek meg (pl. a Repository nem láthatja a Service-t). Az egy szinten lévő beanek természetesen látják egymást.

Dependency injection-nel lehet hozzáférni a függőségekhez. A Spring háromfajta dependency injectiont ismer a beanek közötti függőségek megadásakor: az attribútum, a konstruktor és a setter metódus alapján lehet dependency injectiont megadni. Ez a gyakorlatban azt jelenti, hogy deklarálunk egy attribútumot, és ennek értéket vagy közvetlenül adunk, vagy konstruktoron keresztül, vagy setter-en keresztül. A legjobb gyakorlat, amit a Spring mond, az az, hogy a bean működéséhez kötelezően szükséges

függőségeket konstruktorban adjuk át, az opcionális függőségeket pedig, amelyek nélkül tud működni a bean, setter injectionnel.

Ha az osztályban csak egy konstruktor van, akkor automatikusan konstruktor injection van.

```
@Controller
public class EmployeesController {

    public final EmployeesService employeesService;

    public EmployeesController(EmployeesService employeesService) {
        this.employeesService = employeesService;
    }

    @RequestMapping("/")
    @ResponseBody
    public String helloWorld() {
        return employeesService.helloWorld();
    }
}
```

Ha viszont több konstruktor van, akkor nekünk kell kiválasztanunk, hogy melyik kerüljön meghívásra, ilyenkor kell rátenni az `@Autowired` annotációt.

Konfiguráció Javaban

A `@SpringBootApplication` annotáció egy metaannotáció, azaz rajta is vannak további annotációk. Ezekből az egyik a `@ComponentScan` annotáció, ami azt jelenti, hogy az adott csomagban és az alatta lévő csomagokban található összes Spring Beant megtalálja, vagyis azokat az osztályokat, amelyeken rajta van a következő annotációk valamelyike: `@Component`, `@Repository`, `@Service` vagy `@Controller` vagy `@RestController`.

Van azonban lehetőség arra is, ha mi nem a `@ComponentScan`-t akarjuk használni, hanem explicit módon, Javaban szeretnénk megadni a beanjeinket. Ezt Java konfigurációnak nevezik. Ezt akkor szoktuk használni, amikor a Service osztályainkból nem akarunk Springes dolgokra hivatkozni, nem akarjuk őket módosítani, nem akarjuk rájuk tenni a `@Service` annotációt. Ebben az esetben Javaban kell megadnunk, hogy a Spring hogyan példányosítsa az osztályunkat, és ekkor a Spring Frameworköt *non-invasive* módon használjuk, ami azt jelenti, hogy nem kell a saját osztályunkból a Springre hivatkozni. Ezt a konfigurációt egy külön osztályban kell megadnunk, amelyet a `@Configuration` annotációval látunk el. Tipikusan rétegenként szoktuk külön állományokba kitenni a konfigurációs utasításokat. A legjobb gyakorlat egyébként az, hogy a third party library-ket Java konfigurációval adjuk meg, a saját beanjeinket pedig component scannel, ugyanis ezeket tipikusan csak példányosítani kell, azt pedig a Spring is automatikusan el tudja végezni helyettünk az annotáció alapján.

Ha egy Service osztályon nincs rajta a `@ComponentScan` annotáció, a component scan nem fogja felolvasni. Ahhoz, hogy mégis Spring Bean legyen belőle, a következőt kell tennünk: a

konfigurációs osztályon (ami nem a Service, hanem egy másik osztály) szerepelnie kell a `@Configuration` annotációnak, valamint létre kell hoznunk egy metódust ebben az osztályban, amelynek visszatérési értéke ennek az osztálynak egy általunk példányosított példánya, és erre a metódusra rá kell tennünk a `@Bean` annotációt. A Spring meg fogja hívni ezt a metódust és amit ez a metódus visszaad, azt el fogja helyezni az application contextben, vagyis pontosan ugyanaz történik, mintha az osztályra rátettük volna a `@Service` annotációt.

```
@Bean
public EmployeesService employeesService() {
  return new EmployeesService();
}
```

```
public class EmployeesService {

    public String helloWorld() {
        return "Hello Dev Tools at " + LocalDateTime.now();
    }
}
```

Látható, hogy ebben az esetben mi magunk, explicit módon példányosítottuk a Service osztályunkat. Ez akkor lehet különösen hasznos, ha valamiért nekünk nem elég egy egyszerű példányosítás, hanem olyan példányra van szükségünk, amelyen előtte meghívunk még más metódusokat, akár különböző paraméterekkel.

Fontos megjegyezni, hogy vagy a Service osztályon kell szerepelnie a `@Service` annotációnak, vagy pedig valahol az alkalmazáson belül kell lennie egy `@Bean` annotációval ellátott metódusnak, amely létrehozza a Service osztály egy példányát, mert ha ezek egyike sincs, akkor a Service osztály a Spring számára láthatatlan marad, és a Spring nem fogja azt megtalálni.

Build és futtatás Mavennel

Ha a `https://start.spring.io/` oldalt használjuk arra, hogy a projektünk vázát legeneráltassuk, akkor ott ki lehet választani, hogy a Maven legyen a *build tool* vagy Gradle. Maven projekt választása esetén a klasszikus Maven projektstruktúrát kapjuk. Ha le szeretnénk buildelni az alkalmazásunkat, akkor az `mvn clean package` parancsot kell kiadni, amely a `clean` parancs hatására letörli a `target` könyvtárat, a `package` hatására pedig több különböző lépés után létrejön maga az alkalmazásunk, egy JAR állomány. Ezek a lépések a következők: az osztályok lefordítása, az erőforrás állományok megfelelő helyre másolása, illetve a tesztesetek futtatása.

A `spring-boot-maven-plugin` szerepel a legenerált `pom.xml` állományban, ugyanis ennek a feladata az, hogy miután létrejön a JAR állomány, amiben a lefordított osztályok és az erőforrás állományok vannak, abba még magát a webkonténert, a Tomcatet is belecsomagolja. Ezt úgy teszi meg, hogy az eredeti, "kis" JAR állományt (pl. `employees-`

`0.0.1-SNAPSHOT.jar`) átnevezi `employees-0.0.1-SNAPSHOT.jar.original` névre, és létrehoz egy új, "nagy" JAR állományt, amelyben már benne van a Tomcat konténer is.

Ha szeretnénk elindítani az alkalmazásunkat, azt Maven használatával is megtehetjük. Ehhez az `mvn spring-boot:run` parancsot kell kiadnunk. Ennek hatására először elindul Tomcat és figyelni fog az alapértelmezett 8080-as porton, valamint a Tomcaten belül maga az alkalmazás. Így az alkalmazásunkat már böngészőből is meg tudjuk majd nyitni, le tudunk kérni belőle egy oldalt. Ha az adott gépre nincs telepítve a Maven, csak a JDK, akkor a következő paranccsal indítható az alkalmazás: `java -jar` és ezután következik a JAR állomány teljes neve, tehát például: `java -jar employees-0.0.1-SNAPSHOT.jar`. Mivel ebben a JAR fájlban bele van csomagolva a Tomcat, ezért automatikusan az fog elindulni, és rajta pedig a Springes alkalmazás.

A `https://start.spring.io/` oldalon generált alkalmazás tartalmaz egy ún. Maven wrappert is, ami arra való, hogy akkor futtassuk, ha nincs telepítve az adott számítógépre a Maven. Ez letölti az internetről a Mavent, kicsomagolja és úgy indítja el a build folyamatot. Ennek az az előnye, hogy nem kell előre külön telepíteni a Gradle-t, mert maga az wrapper megteszi ezt. Használható pl. az `mvnw package` parancs kiadásával.

Build és futtatás Gradle használatával

Amennyiben a `https://start.spring.io/` használatával generáltatjuk le az alkalmazásunk vázát, és ott a Gradle projektet választjuk ki, akkor egy olyan projekt jön létre, amelyet Gradle-lel tudunk majd buildelni.

A Gradle-höz két plugint is írtak a Spring Boot fejlesztői. Az egyik az `io.spring.dependency-management` plugin, amely arra való, hogy Maven-szerű függőségkezelést biztosítson, vagyis maga a Spring definiálja a függőségek verziószámait, és amikor mi megadjuk a függőséget, nem kell a verziószámot külön megadnunk, hanem automatikusan a Spring fejlesztői által előre megadott verziószámokat fogja használni. A másik az `org.springframework.boot`, amely képes az előző plugin figyelembe vételével a JAR állományból egy másik, olyan JAR állomány létrehozására, amelybe belecsomagolja a Tomcat webkonténert is.

A `https://start.spring.io/` oldalon generált alkalmazás tartalmaz egy ún. Gradle wrappert is, ami arra való, hogy akkor futtassuk, ha nincs telepítve az adott számítógépre a Gradle. Ez letölti az internetről a Gradle-t, kicsomagolja és úgy indítja el a build folyamatot. Ennek az az előnye, hogy nem kell előre külön telepíteni a Gradle-t, mert maga az wrapper megteszi ezt.

Használható pl. az `gradlew build` parancs kiadásával.

Ezenkívül a Gradle konfigurációs állománya tartalmaz függőséget a JUnit 5-re is.

Nézzük meg, milyen *task*-okat lehet használni:

- Ha le akarjuk buildelni a teljes alkalmazást, elő akarjuk állítani a JAR állományt, akkor a `gradle build` parancsot adjuk ki. Ez nem túl sok mindent ír ki a konzolra, ezért ha

jobban szeretnénk látni, hogy mi történik, akkor használjuk a `-i` kapcsolót, amely `INFO` szintű naplózást ad vissza (`gradle -i build`).

- Amennyiben az alkalmazást futtatni is szeretnénk, tehát először elindítani a beépített Tomcat konténert, majd pedig azon az alkalmazást, akkor a `gradle bootRun` parancsot kell kiadni. Ha az adott gépre csak a JVM van feltelepítve, akkor a következő paranccsal indítható az alkalmazás: `java -jar` és ezután következik a JAR állomány teljes neve, tehát például: `java -jar employees-0.0.1-SNAPSHOT.jar` Mivel ebbe a JAR fájlba bele van csomagolva a Tomcat, ezért automatikusan az fog elindulni, és rajta pedig a Springes alkalmazás.

Unit és integrációs tesztek

A Spring Framework fel van készítve a tesztesetek implementálására. Akár Maven, akár Gradle projektet választunk ki a generálás során, a JUnit 5, a Hamcrest és az AssertJ (meg még néhány, teszteléshez használható függőség) alapból a classpath-ra kerül, vagyis ezen eszközöket automatikusan tudjuk használni a tesztesetek megírásához.

Amikor unit teszteket írunk, akkor nem indítjuk be a Springes dolgokat, az application contextet, mert ott már valamiféle integráció történik, az már nem tisztán unit teszt. Unit tesztek azok, amikor konkrétan egy Java osztályt akarunk letesztelni. Ezt csak akkor tudjuk megtenni, ha mi magunk le tudjuk példányosítani. A Springet tudjuk *non-invasive* módon használni, tehát alapértelmezetten adott az, hogy tudjunk egyes komponenseket unit tesztelni.

```
@Test
void testSayHello() {
  EmployeesService employeesService = new EmployeesService();
  assertThat(employeesService.sayHello())
    .startsWith("Hello");
}
```

Külső hivatkozással rendelkező osztályokat pedig úgy tudunk tesztelni a Spring bevonása nélkül, hogy a külső hivatkozásként tartalmazott objektumot kicseréljük egy *test double*-re, például egy *mock* objektumra. Ez úgy viselkedik, mintha az eredeti objektum lenne, ugyanolyan metódusai vannak, de igazi működés nincs ezek mögött.

```
@ExtendWith(MockitoExtension.class)
public class EmployeesControllerTest {

  @Mock
  EmployeesService employeesService;

  @InjectMocks
  EmployeesController employeesController;

  @Test
  void testSayHello() {
```

```
    when(employeesService.sayHello())
      .thenReturn("Hello Mock");
    assertThat(employeesController.helloWorld())
      .startsWith("Hello Mock");
  }
}
```

Integrációs teszt írásakor már elindul a Spring application contextje, ő maga fogja létrehozni a beaneket, ő végzi el a beanek közötti kapcsolatok kialakítását, vagyis a dependency injectiont. A tesztosztályt a `@SpringBootTest` annotációval kell ellátni. Ez egy metaannotáció, szerepel rajta egy `@ExtendWith(SpringExtension.class)` annotáció. Ez azt mondja meg a JUnit 5-nek, hogy ezt a tesztosztályt az `SpringExtension`-nel kell lefuttatni. A JUnit 5-ben az *Extension*-ök valók arra, hogy a tesztesetek előtt és után bizonyos kódrészletek lefussanak, jelen esetben beindításra kerüljön az application context. Valójában a Spring tesztfuttatója a különböző tesztesetek között nem indítja be mindig újra az application contextet, hanem cache-eli azt, annak érdekében, hogy megfelelő gyorsasággal fussanak a tesztek. Az application context így az első teszteset előtt fog beindulni, és az utolsó után áll le, és minden teszteset ugyanabban az application contextben fog lefutni. Ezért vigyázni kell, hogy az application contextnek ne legyen állapota, ne módosítsuk az állapotát a tesztesetekkel. A beanek injektálhatóak a tesztbe az `@Autowired` annotációval (dependency injection).

```
@SpringBootTest
public class EmployeesControllerIT {

  @Autowired
  EmployeesController employeesController;

  @Test
  void testSayHello() {
    String message = employeesController
      .helloWorld();
    assertThat(message).startsWith("Hello");
  }
}
```

Az integrációs teszteket tartalmazó tesztosztály neve elnevezési konvenció szerint kapja az `IT` végződést (pl. `EmployeesControllerIT`), amely az *integrációs teszt* kifejezés rövidítése.

A Mavenben különválasztható, hogy melyik fázisában melyik tesztek fussanak le, mert a unit teszteket a surefire plugin indítja el, az integrációs teszteket pedig a failsafe plugin. A surefire plugin alapértelmezetten a következő nevű tesztosztályokat indítja el:

- `**/Test*.java`
- `**/*Test.java`
- `**/*Tests.java`
- `**/*TestCase.java`

A failsafe plugin alapértelmezetten a következő osztályokat indítja el:

- `**/IT*.java`
- `**/*IT.java`
- `**/*ITCase.java`

Developer Tools

A Spring Boothoz jár egy **Developer Tools** is, ami a fejlesztési folyamatot könnyíti meg.

Nézzük meg, milyen lehetőségeket ad nekünk a Developer Tools:

- Felülír bizonyos property-ket, például kikapcsolja a template cache-t. A template cache azt jelenti, hogy a Spring Boot alapértelmezetten cache-eli a template-eket, vagyis ha már egyszer betöltötte őket, akkor onnantól memóriából szolgálja ki. Ez megnehezíti a fejlesztést, hiszen ha hozzányúlunk a template-ekhez, akkor minden esetben újra kell indítani az alkalmazást. Developer Tools használata esetén tehát nincs template cache.
- Automatikus újraindítás, ha változik az alkalmazás egy része, akkor automatikusan újraindítja azt, nem kell nekünk megtennünk.
- Globális beállítási lehetőségek a saját számítógépen belül, tehát nem kell projektenként mindig beállítani bizonyos tulajdonságokat, valamint attól sem kell félnünk, hogy véletlenül a verziókövető rendszerbe commitoljuk azokat.
- Lehetőség van távoli eléréssel az alkalmazás módosítására, frissítésére, majd az automatikusan újra is indul.

Ahhoz, hogy a Developer Tools-t használjuk, fel kell venni a `pom.xml`-be a `spring-boot-devtools` függőséget.

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-devtools</artifactId>
</dependency>
```

Ha változik valami a classpath-on, akkor a Developer Tools újraindítja az alkalmazást. IDE-függő, hogy ez mikor történik meg. Eclipse esetén mindig, ha elmentünk egy állományt. IDEA-ban viszont a *Build/Rebuild Project* menüpont használatával (Ctrl + F9 billentyűkombináció).

Úgy oldották meg, hogy két osztálybetöltő van, az egyik a saját kódot tölti be, a másik pedig a függőségeket, tehát újraindításkor nem cseréli ki a sok függőséget, a third party library-ket. Ha viszont a függőségeken változtatunk, tehát a `pom.xml`-hez nyúlunk hozzá, akkor nekünk kell manuálisan újraindítani az alkalmazást. Ilyenkor a háttérben a webkonténer, a Tomcat is újraindul.

Az IDEA-val való fejlesztéskor történhet olyan, hogy az IDEA letörli a `.class` állományt, majd újra lefordítja. Ha pont a kettő között történik meg az újraindítás, akkor nem fogja megtalálni a `.class` állományt. Ehhez a következőt kell beállítani az `application.properties` állományban:

```
spring.devtools.restart.poll-interval=2s
spring.devtools.restart.quiet-period=1s
```

A LiveReload egy ehhez kapcsolódó lehetőség: ha szerveroldalon módosítunk valamit, akkor ez a böngészőben azonnal látszik. Ehhez a böngészőbe fel kell tenni egy LiveReload plugint. A Developer Tools a szerveren elindít egy LiveReload szervert, ekkor a szerver és a böngésző plugin között kialakul egy WebSocket kapcsolat. Amikor pedig a szerveroldalon újraindítás történik, akkor erről értesül a böngésző plugin, és automatikusan frissíti az oldalt.

Statikus állományok változtatása esetén a böngészőre átváltva azonnal látszik a változtatott erőforrás. Ez is IDE-függő, hogy mi számít változtatásnak a Developer Tools számára.

Global Settings esetén a saját számítógépemen globálisan felül tudom írni az alkalmazás konfigurációját. Ebben az esetben a *home*-könyvtáramon belül létre kell hozni egy `.config/spring-boot` alkönyvtárat, és ebben el kell helyezni a saját properties állományunkat, amelyben meg kell adni azokat az értékeket, amiket felül akarok írni.

Remote Applications használatakor a `spring-boot-maven-plugin`-t tudjuk felkonfigurálni. Az alkalmazás lokális fejlesztésekor a *Remote Client Application*-t kell elindítani, ez hozzácsatlakozik a távoli számítógépen futó alkalmazásunkhoz, és ha változik egy állomány, akkor azt a hálózaton átviszi (a `.class` fájlt), ezt a szerveroldalon lévő alkalmazás észreveszi és újra fog indulni. Ez nagyon jól használható fejlesztő- és tesztkörnyezetben, de azért éles környezetben ne használjuk, mert megvannak a maga veszélyei.

Twelve-Factor App

A **Twelve-Factor App** egy olyan ajánlásgyűjtemény, melyet a *Heroku* platform fejlesztői dolgoztak ki arra, hogy hogyan lehet olyan alkalmazást fejleszteni, amely a felhőbe telepíthető és üzemeltethető. A Twelve-Factor App tehát nem más, mint szabályok és javaslatok gyűjteménye. Ha ezeket betartjuk, akkor az alkalmazásunkat egyszerű lesz felhőbe telepíteni. Nem csak felhőben futó alkalmazás esetén érdemes ezeket betartani, hanem klasszikus alkalmazások esetén is, hiszen ezzel az üzemeltetési költségeket biztosan csökkenthetjük, valamint a hibázási lehetőségek is minimalizálhatóak.

A kidolgozásakor a *cloud*-ot, a *Platform as a Service (PaaS)*-t és a *continuous deployment*-et tartották szem előtt, hiszen a Heroku nem más, mint egy PaaS-szolgáltató.

A *Platform as a Service* azt jelenti, hogy a szolgáltató számunkra nem egy virtuális gépet ad, tehát nem nekünk kell rá az operációs rendszert, az adatbázisokat, az alkalmazásszervereket telepíteni, hanem a szolgáltató ezt elvégzi helyettünk. Nagyon sok ilyen szolgáltató létezik (pl. Google App Engine, Redhat Open Shift, Pivotal Cloud Foundry, Heroku, AppHarbor, Amazon AWS, stb.)

A Spring fejlesztése mögött a Pivotal nevű cég áll. Ők dolgozták ki a *cloud native* fogalmat. Ez egy jelző, azokat a cégeket, szervezeteket jellemzi, amelyek gyorsan képesek az automatizálás előnyeit kihasználva olyan alkalmazásokat fejleszteni, amelyek megbízhatóak és skálázhatóak. A fogalom megalkotásakor a *continuous delivery*-t, a *DevOps*-ot és a

microservice-eket tartották szem előtt. Egy *cloud native* jelzővel rendelkező cég által kifejlesztett alkalmazás jellemzői:

- Tipikusan valahol a felhőben fut (PaaS-en, amely lehet publikus vagy privát is).
- Rendelkezik az *elastic* tulajdonsággal, vagyis ha maga az infrastuktúra észreveszi, hogy hirtelen megnövekedett a felhasználók száma, ezárt automatikusan skálázódik, hogy az alkalmazás ki tudja szolgálni a megnövekedett igényeket.

A Twelve-Factor App a következő 12 szabályt tartalmazza:

1. Verziókezelés: az adott alkalmazáshoz tartozó összes forráskódnak és erőforrás állománynak egy repository-ban kell lenniük, ugyanabban a verziókezelő rendszer repository-ban.
2. Függőségkezelés: az alkalmazás külső függőségeit explicit módon deklarálni kell és külön kell tárolni ezeket az alkalmazás forráskódjától.
3. Konfiguráció: az adott környezetekre jellemző konfigurációk nem az alkalmazás részét képezik, hanem az adott környezet részének kell lenniük.
4. Háttérszolgáltatások: el kell gondolkozni azon, hogy milyen külső háttérszolgáltatásokat akarunk igénybe venni, és ezeket külön kell kezelnünk. Ezek ne legyenek az alkalmazásunk részei, hanem ezek külön jelenjenek meg, és az alkalmazásunk ehhez kapcsolódjon. Ilyen például az adatbázis. Itt fontos megjegyezni, hogy a többi alkalmazás is, amihez a mi alkalmazásunk kapcsolódik, háttérszolgáltatásnak minősül.
5. Vegyük külön a build, a release és a futtatás lépéseit! Ezek ne keveredjenek, hanem legyenek jól szeparálva!
6. Folyamatok: Az alkalmazás állapotmentes folyamatokból álljon. Az állapotmentesség nagyon fontos, hiszen az alkalmazásunk csak így lehet tetszőlegesen skálázható.
7. Port hozzárendelés: fontos, hogy meg tudjuk mondani, hogy az alkalmazásunk melyik porton induljon el. Ez is a skálázhatóság miatt fontos, hogy akár több alkalmazáspéldányt is el lehessen indítani párhuzamosan, ha arra van szükség.
8. Párhuzamosság: fontos arra figyelni, hogy az alkalmazást a felhasználók párhuzamosan is teljes értékűen igénybe vehessék, ne legyenek a párhuzamosságból eredő problémák.
9. Mivel felhőkörnyezetben dolgozunk, fontos, hogy az alkalmazás gyorsan tudjon elindulni és gyorsan tudjon leállni is.
10. Próbáljunk arra törekedni, hogy az éles és a fejlesztői környezet a lehető legjobban hasonlítson egymásra. Így minimalizálható a hibák elkövetésének lehetősége.
11. Naplózás: fontos, hogy az alkalmazás naplózzon. A Twelve-Factor App azt mondja, hogy lehetőleg a konzolra naplózzon, mert az alkalmazásnak nem feladata a naplózás céljának meghatározása, hanem az üzemeltető és az infrastruktúra dolga az, hogy ezt a naplót oda továbbítsa/mentse, ahova szükséges. Tehát a naplózás céljáról ne az alkalmazás döntsön, hanem ezt kívülről lehessen megadni standard, operációs rendszer szintű eszközök használatával.

12. Felügyeleti folyamatok: az alkalmazás mellett tároljuk és ugyanúgy verziókezeljük azokat a felügyeleti folyamatokat, amelyeket ad-hoc módon kell lefuttatni. Ez azért fontos, hogy ne lehessen ilyeneket kívülről, egyszeri esetekben lefuttatni.

Ezenkívül van egy könyv is, amelyet szintén a Pivotalnál írtak. A címe: *Beyond the Twelve-Factor App*. Ez főleg példákat hoz arra, hogy hogyan lehet a fenti 12 szabályt megvalósítani. Ezek közül néhányat érdemes kiemelni:

- A könyv azt mondja, hogy először alakítsuk ki az API-t és utána fejlesszük le az alkalmazást. (Erről megoszlanak a vélemények, sokan vannak, akik ezt pontosan fordítva látják jónak.)
- A konfigurációnál kiemeli a jelszavak és tanúsítványok használatának fontosságát.
- Azt írja, hogy különös figyelmet kell fordítani az autentikációra és az autorizációra, vagyis arra, hogy hogyan történik az alkalmazásba való bejelentkezés és a hozzáférések szabályozása. Ha állapotmentes alkalmazást írunk, akkor nem tárolhatjuk el magában az alkalmazásban a felhasználók adatait, hanem valamilyen más módon kell ezt megoldani.
- Telemetria: ezeket az alkalmazásokat monitorozni kell tudni, azaz az alkalmazás állapotát lehessen lekérdezni, és az így kapott adatokat visszamenőlegesen tárolni és akár grafikusan megjeleníteni is lehessen.

Bevezetés a Docker használatába

A Spring Boot beépítetten tartalmaz Docker-támogatást. Nézzük meg, mi is az a Docker, hol könnyíti meg a munkánkat!

A **Docker** egy operációs rendszer szintű virtualizáció. Nagyon hasonlít a klasszikus virtualizációra, csak éppen kevésbé erőforrás-igényes, ugyanis ebben az esetben nem kell a teljes operációs rendszert telepítenünk és futtatnunk. A virtualizáció használatával különálló, jól elkülönített környezeteket tudunk létrehozni, Docker esetén saját fájlrendszerrel, saját telepített szoftverekkel, függőségekkel együtt. Ezek a létrehozott környezetek viszont csak egy jól meghatározott módon tudnak egymással kommunikálni. A fejlesztéskor a saját számítógépünkkel dolgozunk, de az éles környezet, ahol az alkalmazás futni fog, eltérhet ettől, tehát az alkalmazásunk akár máshogy is működhet. Ha viszont mindkét helyen Dockerben futtatjuk, akkor a két környezet nem tér el egymástól, ezért bízhatunk benne, hogy az éles környezetben nem fognak előjönni olyan hibák, amelyeket csak ott lehet reprodukálni. Ha egyszerre több alkalmazás fejlesztésében veszünk részt, akkor nem kell azzal küzdeni, hogy az adott számítógépre a különböző fejlesztőkörnyezetek különböző verzióit hogyan telepítsük föl, hanem meg tudjuk azt csinálni, hogy létrehozunk egy Docker-konténert, amelyben az adott szoftver egyik verziója, és egy másikat, amiben egy másik verziója van feltelepítve. Azt is egyszerűen meg tudjuk tenni, hogy ilyen módon egy több számítógépből álló hálózatot reprodukálunk. A microservice-ek esetén az is fontos lehet, hogy a Docker konténerrel azonnal egy kész környezetet kapunk, és nem kell egy másik alkalmazást feltelepítenünk, annak minden komponensével együtt.

A Docker architektúrája úgy néz ki, hogy az infrastruktúrára (az adott számítógépre) feltelepítünk egy operációs rendszert, arra a Dockert, és azon lehet futtatni a különböző környezeteket. A Docker sok komponensből, modulból áll. Ezek közül a következőket érdemes megemlíteni:

- Docker Hub: ez egy publikus szolgáltatás Docker image-ek tárolására és megosztására.
- Docker Compose: egyszerre több Docker konténert lehet vele egyidejűleg kezelni és elindítani.
- Docker Swarm: olyan eszköz, amellyel teljes hálózatot lehet felépíteni, natív cluster támogatást tartalmaz.
- Docker Machine: távoli Docker környezetek üzemeltetéséhez használható.

Két fontos fogalom:

A **Docker image** valójában egy (több) fizikai állomány, ez tartalmazza az adott környezetet, annak fájlrendszerét, a feltelepített szoftvereket. Ez az *Image Registry*-ben helyezhető el, tehát például a Docker Hub-on, publikusan. A **Docker konténer**: amikor az image-et letöltjük és elindítjuk, akkor jön létre a saját gépünkön egy konkrét futó példány.

A következő parancsokat lehetnek fontosak a Docker használatakor:

- Az image-eket a `docker images` paranccsal lehet lekérdezni.
- A `docker run`-nal lehet elindítani egy adott image-et (pl. `docker run hello-world` a `hello-world` nevű image-et indítja el). Ha a kívánt image nem található a számítógépünkön, akkor a Docker automatikusan letölti annak a legfrissebb verzióját a Docker Hub-ról, majd el is indítja azt.
- A futó konténereket a `docker ps` paranccsal lehet kilistázni.
- Az összes konténert (a leállítottakat is) pedig a `docker ps -a` paranccsal.
- Egy korábban leállított konténert a `docker start` paranccsal lehet újraindítani.
- A `docker version` paranccsal lehet lekérdezni a Docker verziószámát.
- A `docker run -p 8080:80` parancs azt jelenti, hogy az elindított konténerben elindul egy webszerver, amely a konténer 80-as portján kezdi meg a kiszolgálást. Ez a port hozzákapcsolódik a saját host-számítógépünk 8080-as portjához, és mi azon keresztül kaphatjuk vissza a webszerver által szolgáltatott dolgokat.
- A `docker run -d -p 8080:80` parancs esetén a `-d` kapcsoló hatására azonnal visszakapjuk a promptot, miközben elindul a konténer a háttérben.
- A `docker stop` paranccsal együtt egy számot megadva lehet leállítani egy adott konténert. Amikor elindul egy konténer, akkor egy hash-t kap, amelyre a leállításhoz hivatkozni kell. (Például: `docker stop 517e15770697`.) Ha ennek az első három száma egyedi, akkor elegendő azt is beírni.
- A `--name` paraméter használatával nevet is adhatunk az egyes konténereknek, hogy ne a hash-ekkel kelljen dolgoznunk, ezt követően természetesen hivatkozhatunk rájuk a nevükkel is.
- A `docker logs -f` hatására hozzá tudunk csatlakozni az adott konténerhez, és meg tudjuk nézni, hogy az mit ír ki a konzolra. A `-f`-fel folyamatosan látjuk a kiírt üzeneteket, ahogy azok kiírásra kerülnek.

- Egy konténer törléséhez ki kell adni először a `docker stop` parancsot, és ha ilyen módon leállítottuk, akkor a `docker rm` paranccsal törölni lehet.
- A `docker images` paranccsal az összes letöltött vagy létrehozott image-et ki lehet listázni.
- A `docker rmi` paranccsal ki tudjuk törölni az adott image-et.

Fontos megkülönböztetni az előbbiekben említett `run` és `start` parancsot: a `run` egy image-ből készít új konténert és elindítja, a `start` pedig egy már korábban létrehozott és leállított konténert indít újra.

Java alkalmazások Dockerrel

Ha az alkalmazásunkat Docker konténerben szeretnénk futtatni, ahhoz először létre kell hozni a Docker image-et.

> A videóhoz képest itt kicsit más parancsokat használunk, de alapvetően nagyon hasonló.

Standard Dockeres eszközökkel ez a következőképpen néz ki:

```
FROM adoptopenjdk:14-jre-hotspot
WORKDIR /opt/app
COPY target/*.jar employees.jar
CMD ["java", "-jar", "employees.jar"]
```

Először létre kell hozni az alkalmazás gyökerében egy `Dockerfile` nevű fájlt, amelyben leírhatjuk, hogyan álljon össze az image. Ennek az első sora a `FROM` után tartalmazza azt, hogy milyen, már létező image-ből indulunk ki. Nem kell ugyanis minden esetben elölről kezdeni, hanem támaszkodhatunk már meglévő image-ekre. Ezután a `WORKDIR` paranccsal hozunk létre egy `/opt/app` könyvtárat az image-en belül. Az ezt követő parancsok ebben a könyvtárban fognak dolgozni. A `COPY` paranccsal a `target` könyvtárban szereplő JAR állományunkat másoljuk át az aktuális (`/opt/app`) könyvtárba, így ez már a konténeren belül szerepel. A `CMD` után pedig kiadunk egy parancsot, amivel elindítjuk a JVM-et, és paraméterül pedig átadjuk a JAR állomány elérési útját. (Ez a parancsot a konténer indításakor adja ki, és nem az image buildelésekor.)

A `docker build -t employees .` paranccsal a Docker-t megkérjük arra, hogy az adott könyvtárból (erre utal a parancs végén lévő pont (`.`) karakter) olvassa fel a `Dockerfile`-t, illetve az összes többi fájlt is megkapja a Docker, hogy azokat be tudja másolni az image-be, tehát a `COPY` parancs valójában ezért fog működni. A `-t employees` kapcsolóval mondjuk azt meg, hogy az image neve (*tag*) `employees` legyen. Ezzel tehát létrejön egy image, aminek a neve `employees`, és benne van egy JAR állományunk. Ahhoz, hogy ebből az image-ből létrehozzunk egy konténert, a következő parancsot kell kiadnunk: `docker run -d -p 8080:8080 employees` Így elindul az image alapján a konténer és azon belül elindul az alkalmazásunk is, ami a 8080-as porton várja a kéréseket, valamint a saját számítógépünk 8080-as portja automatikusan továbbításra kerül a konténer 8080-as portjára.

Lehetőség van a Docker vezérlésére Mavenből is, ehhez egy plugin-t kell felvenni a `pom.xml`-be függőségként, és ennek meg kell adni egy konfigurációt is. Többféle ilyen plugin is van, a videóban a következő szerepel:

```
<plugin>
    <groupId>io.fabric8</groupId>
    <artifactId>docker-maven-plugin</artifactId>
    <version>0.32.0</version>
    <configuration>
      <verbose>true</verbose>
      <images>
          <image>
              <name>employees</name>
              <build>

<dockerFileDir>${project.basedir}/src/main/docker/</dockerFileDir>
                  <assembly>
                      <descriptorRef>artifact</descriptorRef>
                  </assembly>
                  <tags>
                      <tag>latest</tag>
                      <tag>${project.version}</tag>
                  </tags>
              </build>
              <run>
                  <ports>8080:8080</ports>
              </run>
          </image>
      </images>
  </configuration>
</plugin>
```

A `Dockerfile`, amit az előbb említett könyvtárban el kell helyezni, ebben az esetben így néz ki:

```
FROM adoptopenjdk:14-jre-hotspot
RUN mkdir /opt/app
ADD maven/${project.artifactId}-${project.version}.jar \
  /opt/app/employees.jar
CMD ["java", "-jar", "/opt/app/employees.jar"]
```

Ez nagyon hasonló az előzőhöz, csak itt placeholderekkel vannak megadva a Maven property-jei. A Mavenen keresztül kiadható parancsok a következők:

- `mvn package docker:build`: legyártja az adott image-et.
- `mvn docker:start`: elindítja az image alapján a konténert.
- `mvn docker:stop`: leállítja és egyben le is törli a konténert. (Minden egyes indításkor új konténert hoz létre.)

A Twelve-Factor Appra hivatkozva elmondható, hogy Docker használatakor fontos betartani a következőket:

- Az alkalmazásaink nagyon gyorsan induljanak el és álljanak le, ne legyenek hosszú inicializációs műveletek.
- Működjön a *graceful shutdown*, azaz az alkalmazás egy szabályos leállításkor tegyen rendet maga után.
- Biztosítani kell, hogy ne maradjon a leállítás után ellentmondásokkal teli adat (batch folyamatoknál is!). Ez egyrészt úgy biztosítható, hogy tranzakciókat használunk, hogy még véletlenül se fordulhasson elő az, hogy ha esetleg két művelet között leállításra kerül a konténer, akkor inkonzisztens adat marad a rendszerben. Másrészt pedig azt is lehet csinálni, hogy a műveleteink idempotensek, azaz nem probléma, ha kétszer egymás után elindításra kerülnek, mert mindenképpen ugyanazt az eredmény adják vissza.

Docker Layers

A Docker az image-eket nem egyetlen állományban tartja, hanem többen. Ezeket *layer*-eknek nevezzük. Úgy működik, hogy amikor egy létező image-ből indulunk ki, akkor az egy (vagy több) layert alkot, és az eredeti image-et nem duplikálja, vagyis az csak egyszer szerepel a fájlrendszerben. Az új, létrehozott image-nek létrehoz egy új layert, amelybe már csak a két image közötti különbségeket rakja. Ez azért hasznos, mert így spórolni tud a tárhellyel és a hálózattal. Egy image-hez le lehet kérdezni a `docker image inspect` (+ image neve) paranccsal a hozzá tartozó layereket. A legjobb gyakorlat az, ha úgy építjük fel az image-einket, hogy a leggyakrabban változó állományokat külön layerbe tesszük. Tehát az operációs rendszerhez tartozó állományokat, a különböző third party library-ket alacsonyabb layeren érdemes tartani, mert ezek ritkábban változnak, viszont az alkalmazás saját dolgait (Java alkalmazás esetén az osztályok lefordított `.class` állományait) érdemes ezektől különböző layeren tárolni.

Manuálisan úgy lehet megoldani, hogy a Java alkalmazásunk több layerben legyen, hogy mi állítjuk össze a Docker image-et, figyelve arra, hogy a különböző fájlokat különböző `COPY` paranccsal másoljuk be az image-be. Először a ritkán változó részeket (a third party library-ket), majd utána a gyakran változó állományokat (a `.class` állományokat) kell bemásolni. Ez azt jelenti, hogy nem a teljes JAR állományt másoljuk be egyszerre, hanem kicsomagoljuk és részenként másoljuk be a JAR állomány különböző részeit.

A Spring alkalmazás felépítése úgy néz ki, hogy a `BOOT-INF/lib` könyvtárban vannak a third party library-k, a `META-INF` könyvtárban vannak a különböző leíró állományok, és a `BOOT-INF/classes` könyvtárban vannak az alkalmazáshoz tartozó állományok, vagyis az általunk írt osztályok lefordított CLASS állományai kerülnek ide. Ezeket az itt leírt sorrendben érdemes bemásolni az image-be, így azok különböző, egymásra rakódó layerekre fognak kerülni.

Az alkalmazás elindítása kicsit módosul, ugyanis ekkor már nem a `java -jar` paranccsal lesz indítható, hanem a `java -cp` paranccsal kell megadni a classpath-t, és paraméterül át

kell adni több könyvtárat is, amelyeknek a tartalmát szeretnénk, ha bekerülne a classpath-ba.

```
FROM adoptopenjdk:14-jdk-hotspot as builder
WORKDIR app
COPY target/employees-0.0.1-SNAPSHOT.jar employees.jar
RUN jar xvf employees.jar

FROM adoptopenjdk:14-jre-hotspot
WORKDIR app
COPY --from=builder app/BOOT-INF/lib lib
COPY --from=builder app/META-INF META-INF
COPY --from=builder app/BOOT-INF/classes classes

CMD ["java", "-cp", "classes:lib/*", \
                "training.employees.EmployeesApplication"]
```

A Spring a `2.3.0.M2` verziója óta beépített támogatást tartalmaz arra, hogy layerekkel tudjunk Docker image-eket létrehozni. Kétféle megoldást támogat: a *layered JAR* és a *buildpack* létrehozását.

A *layered JAR* megoldás választásával arra utasítjuk a Spring Boot plugint, hogy úgy állítsa össze az alkalmazást, hogy a JAR maga eleve layereken helyezkedjen el, és utána, amikor ki akarjuk csomagolni, akkor csak egy parancsot kelljen ehhez kiadnunk. Ezt a `spring-boot-maven-plugin`-nak a következő konfigurációval lehet megadni:

```
<plugin>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-maven-plugin</artifactId>
  <configuration>
    <layers>
      <enabled>true</enabled>
    </layers>
  </configuration>
</plugin>
```

Majd a következő parancsokat kell megadni ahhoz, hogy önmaga kitömörítse a JAR állományt:

```
java -Djarmode=layertools -jar target/employees-0.0.1-SNAPSHOT.jar list

java -Djarmode=layertools -jar target/employees-0.0.1-SNAPSHOT.jar extract
```

A `list` parancs kilistázza az adott JAR-ban lévő layerek tartalmát, az `extract` pedig ki is tömöríti azokat.

Az ehhez tartozó `Dockerfile` így néz ki:

```
FROM adoptopenjdk:14-jre-hotspot as builder
WORKDIR application
COPY target/employees-0.0.1-SNAPSHOT.jar employees.jar
```

```
RUN java -Djarmode=layertools -jar employees.jar extract

FROM adoptopenjdk:14-jre-hotspot
WORKDIR application
COPY --from=builder application/dependencies/ ./
COPY --from=builder application/spring-boot-loader/ ./
COPY --from=builder application/snapshot-dependencies/ ./
COPY --from=builder application/application/ ./
CMD ["java", \
  "org.springframework.boot.loader.JarLauncher"]
```

Ennek az első blokkja létrehoz egy image-et (ez a *builder*), ez kitömöríti a JAR állományt. A második blokk pedig létrehoz egy második image-et úgy, hogy a kitömörített állományokat másolja át az első image-ből. A `COPY` parancs minden esetben egy új layert hoz létre.

Buildpack használatakor Mavenből vagy Gradle-ből tudunk közvetlenül Docker image-et gyártani. Ez egy magasabb absztrakciós szintet képvisel a `Dockerfile`-hoz képest. Ekkor a következő parancsokat kell kiadni:

- `mvn spring-boot:build-image` (ezzel létrejön az image)
- `docker run -d -p 8080:8080 –name employees employees:0.0.1-SNAPSHOT` (ezzel az image-ből létrejön és el is indul a konténer).

A Twelve-Factor Appnak is van iránymutatása a függőségekkel kapcsolatban:

- Az első szabály az, hogy az alkalmazás nem függhet az őt futtató környezetre telepített semmilyen csomagtól. Ez azt jelenti, hogy nem indulhatunk ki egy olyan image-ből, amely már eleve mindenféle JAR állományokat tartalmaz, hanem ezt magának az alkalmazásnak kell hordoznia.
- Másik szabály, hogy a függőségeket explicit módon deklarálni kell, vagyis meg kell mondani, hogy milyen külső függőségei vannak az alkalmazásnak. (Nem minősülnek függőségnek a különböző háttérszolgáltatások, pl. adatbázis, mail-szerver, stb., ezeket *backing-service*-nek hívja a Twelve-Factor App).
- Fontos, hogy az alkalmazást egybe kell csomagolni a függőségekkel, hiszen a futtató környezetben szükség van ezekre.
- A gyakran változó részeket külön Docker layerbe tegyük.
- Vigyázni kell arra, hogy a build folyamat megismételhető legyen: ne használjunk intervallumokat a verziószámok megadásakor, hanem adjuk meg a verziószámot explicit módon.

Feltöltés Git repository-ba

Amennyiben a projektünket szeretnénk feltölteni egy Git repository-ba (például a GitHubra), akkor először létre kell hoznunk ezt a repository-t. Ezt megtehetjük a GitHub webes felületén, ahol meg kell adnunk a projektünknek a nevét. Ezután ez a repository elérhető a következőhöz hasonló címen: `https://github.com/username/employees.git` (itt a `username` helyére természetesen a saját felhasználónevünk, az `employees` helyére

pedig a saját repository-nk neve kerüljön.) Ezután a saját lokális számítógépünkön a következő parancsokat kell kiadni:

- `git init` - létrehoz egy lokális repository-t az adott könyvtárba
- `git add .` - ehhez a lokális repository-hoz hozzáadjuk az adott könyvtárban található állományokat
- `git commit -m "First commit"` - ezzel a paranccsal véglegesítjük, tehát feltöltjük a lokális repository-ba ezeket a fájlokat (a parancs maga a `git commit`, a `-m "First commit"` pedig csupán egy üzenet ehhez a commithoz)
- `git remote add origin https://github.com/username/employees.git` - ezzel a paranccsal összekötjük a lokális repository-t a távoli (a GitHubon létrehozott) repository-val
- `git push` - ezzel a paranccsal feltöltjük a lokális változtatásainkat a távoli repository-ba, a GitHubra

A GitHub webes felületén ezután azonnal meg is nézhető már, hogy fel lettek töltve a változtatások.

Természetesen az IDEA fejlesztőeszköz a parancssori műveletek helyett grafikus interfészt biztosít. Lásd a *Git / GitHub/ Share Project on GitHub* menüpontot!

A Twelve-Factor App-nak a verziókezelésre is vannak ajánlásai. Fontos, hogy egy repository egy alkalmazást tartalmazzon. Minden más, amit például függőségként definiálunk, az külön repository-ba kerüljön. Ennek a szabálynak a megszegése például az, ha egy több modulból álló alkalmazást modulonként külön repository-ban tárolunk. Ez elsőre jó ötletnek tűnhet, de a build-folyamatot nagyon nagy mértékben megbonyolítja. A szabályt sérti az is, ha ugyan több külön futtatható alkalmazásunk van, de ugyanazon a domainen, ugyanazon az adatbázison dolgoznak, mégis különböző repository-kban vannak. Ez azért nem igazán jó megoldás, mert ezek az alkalmazások nagyon összefüggenek, ha az egyik közülük módosul, akkor nagyon valószínű, hogy módosulnia kell a másiknak is. Ebben az esetben érdemes őket egy repository-ba összevonni, még akkor is, ha ezek különböző alkalmazások. Az viszont természetesen kerülendő, hogy teljesen különálló alkalmazások kerüljenek egy repository-ba. A különböző környezetekre telepített alkalmazásoknál alapvető igény, hogy egyszerűen meg tudjuk nézni, hogy mely verzióból készült. Ez nagyban megkönnyíti a hibakeresést. Fontos, hogy ez látható legyen a felhasználói felületen.

Fontos figyelni a *Conway törvény*-re: “azok a szervezetek, amelyek rendszereket terveznek, …kénytelenek olyan terveket készíteni, amelyek saját kommunikációs struktúrájuk másolatai”. A fejlesztő csapatok száma tipikusan együtt jár az alkalmazás moduljainak számával. Erre érdemes figyelni, hogy a különböző csapatok jól körülhatárolható keretek között, egymástól külön tudjanak dolgozni, így hatékonyabbak lesznek, és kevesebbet is kell majd a csapatok között szinkronizálni. Ha túl nagy az alkalmazás és egyszerre több csapat kezdene el dolgozni rajta, akkor érdemes valamit kitalálni, hogy ez hosszú távon ne legyen így, mert a közös munka nagyon be tudja lassítani a csapatokat. Tehát érdemes úgy felosztani a feladatot, hogy minden fejlesztő csapatnak meglegyen a maga külön kis microservice-e, és ezzel lehessen hatékonyabbá tenni a közös munkát.

REST webszolgáltatások bevezetés

Ebben a fejezetben részletesen megtanulhatod, hogyan tudsz RESTful webszolgáltatásokat implementálni. Webes alkalmazások esetén gyakran JavaScript frontend REST webszolgáltatásokon keresztül hívja a backendet. Megnézzük, hogyan lehet implementálni a különböző HTTP metódusokat, amelyekkel a CRUD műveletek elvégezhetők. Szó esik a különböző HTTP státuszkódok jelentéséről és kezeléséről, valamint hogy mi a teendő, ha valamilyen hiba keletkezik az alkalmazás futása során. Megismerhetsz többféle integrációs tesztelési módot és eszközt, és egy olyan eszközt is, amellyel egyszerűen, grafikus felületen ellenőrizheted a webszolgáltatásaid működését. Röviden megnézzük, hogyan választhatod ki, milyen formában szeretnéd a HTTP választ visszakapni. A fejezet végén szót ejtünk arról, hogyan írhatsz validációt a felhasználótól az alkalmazásodhoz beérkező adatokra.

REST webszolgáltatások – GET művelet

A modern alkalmazások manapság úgy néznek ki, hogy a *backend* és a *frontend* különböző technológiákkal van megoldva. Egy megoldás, hogy a backend, azaz az üzleti logika és a perzisztencia Javaban van megírva. A frontend, vagyis a böngészőben futó része az alkalmazásnak pedig HTML, CSS és JavaScript használatával, és minderre általában ráépül még egy JavaScript alapú keretrendszer (pl. Angular, React, Vue.js). Ahhoz, hogy az alkalmazás két része kommunikálni tudjon egymással, tipikusan még valamilyen integrációs technológia is szükséges. Manapság a legelterjedtebb a **REST webszolgáltatások**.

A REST webszolgáltatások alkalmasak két alkalmazás közötti integrációra, de alkalmas arra is, hogy a frontend és a backend között teremtse meg a kommunikációs kapcsolatot.

A REST a *Representational state transfer* rövidítése, és Roy Fielding találta ki 2000-ben. Ő azt gondolta, hogy egy alkalmazásra tekintsünk úgy, mint erőforrások gyűjteményére, és minden egyes erőforrást egy egyedi azonosítóval meg lehessen címezni, és rajtuk csak a *CRUD* műveleteket lehessen elvégezni (*Create*, *Read*, *Update*, *Delete*).

A REST erőteljesen épít a HTTP-protokollra. Az erőforrások címzésére az URI-t használja, a különböző műveleteket a HTTP-metódusokkal lehet elvégezni (GET, PUT, POST, DELETE), és ugyanazok a státuszkódok is megtalálhatóak benne (pl. 200 – minden rendben, 404 – nem található).

A REST webszolgáltatás nem mondja meg azt, hogy milyen formátumban közlekedjen az adat a kommunikáció során, azonban a leggyakrabban JSON-t használnak. Ezt nagyon egyszerű böngészőben feldolgozni, ember és gép által is könnyen értelmezhető, ezért is tudott nagyon elterjedtté válni.

A REST webszolgáltatások tervezésénél figyelembe vették azt, hogy egyszerű, skálázható és platformfüggetlen legyen. A HTTP protokollt azért választották, mert minden platformon elérhető.

Springben REST webszolgáltatások implementálásához a Spring MVC keretrendszert lehet használni, abban ugyanúgy controller osztályokat kell létrehozni, a különbség csak annyi, hogy a metódusok visszatérési értéke innentől kezdve nem egy template neve lesz, hanem azt valamilyen módon (tipikusan JSON formátumban) szerializálva vissza kell küldeni az ügyfélnek, azaz tipikusan a böngészőnek. Ehhez Springben minden metódusra rá kell rakni a `@ResponseBody` annotációt. Azért, hogy ezt ne kelljen ténylegesen minden metóduson megtenni, létrehozták a `@RestController` annotációt, amely a `@Controller` annotációt annyival egészíti ki, hogy az osztályban lévő összes metódust implicit ellátja a `@ResponseBody` annotációval. A `@RequestMapping` annotáció továbbfejlesztéseként pedig már nem csak paraméterként tudjuk minden Java metódusra megmondani, hogy milyen HTTP metódust szolgál ki, hanem külön annotációk lettek, pl. `@GetMapping`, `@PostMapping`, stb.

```
@RestController
@RequestMapping("/api/employees")
public class EmployeesController {

    private final EmployeesService employeesService;

    public EmployeesController(EmployeesService employeesService) {
        this.employeesService = employeesService;
    }

    @GetMapping
    public List<EmployeeDto> listEmployees() {
        return employeesService.listEmployees();
    }
}
```

Az alkalmazás egyes rétegei közötti adattranszferre *DTO*-kat használunk, azaz *data transfer object*-eket. Ezek olyan egyszerű objektumok, amelyek kizárólag arra valók, hogy átvigyék az adatot az egyik rétegből a másikba. Speciális DTO-k a *command*-ok, amely olyan bejövő DTO, amelynél történik valamilyen állapotváltozás (pl. adatbázis módosítás). Szükség van valamiféle konverzióra is az entitások és a DTO-k között.

Ahhoz, hogy felesleges gépeléstől megóvjuk magunkat, érdemes a **Lombok** nevű third party library-t használni, amely képes arra, hogy boilerplate kódot generáljon. Elég például egy egyszerű annotációt rátenni az osztályra, és a Lombok ennek hatására automatikusan elvégzi a getter és setter metódusok generálását. Ezenkívül tud még konstruktort illetve `toString()`, `equals()` és `hashCode()` metódusokat, valamint SLF4J logger deklarációt is generálni. Ahhoz, hogy egy működjön, be kell kapcsolni az *annotation processor*-t, ugyanis ez a kódgenerálás fordítási időben történik. A Lombokot az IDEA is támogatja, ehhez két lépésre van szükség: be kell kapcsolni az *Enable annotation processing*-et, másrészt pedig fel kell telepítenünk a Lombok plugint.

A `@Data` annotációval egyszerre helyettesíthetjük a `@ToString`, az `@EqualsAndHashCode`, a `@Getter` (minden attribútumon), a `@Setter` (minden nem *final* attribútumon) és a `@RequiredArgsConstructor` annotációkat, ezek a metódusok és a konstruktor ezáltal mind

legenerálásra kerülnek az adott osztályban. A `@NoArgsConstructor` annotációval pedig legenerálódik egy paraméter nélküli konstruktor.

```
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Employee {

    private long id;

    private String name;

    public Employee(String name) {
        this.name = name;
    }
}
```

A Lombokot a használathoz fel kell venni függőségként:

```
<dependency>
  <groupId>org.projectlombok</groupId>
  <artifactId>lombok</artifactId>
</dependency>
```

A DTO-k és az entitások közötti konverzióra nagyon sok eszköz létezik, ezek közül az egyik a **ModelMapper**, amely *reflection* alapon működik, vagyis végigmegy reflectionnel a Java entitáson, kiolvassa az attribútumok értékeit (getter metódusokon keresztül), és behív a DTO osztályba, meghívja annak setter metódusait a korábban kiolvasott értékekkel. Ha az attribútumok neve megegyezik, akkor ezeket automatikusan össze tudja párosítani, ha viszont ez nincs így, vagy valamiféle egyéb konverzióra is szükség van, akkor erre egy fluent API interfészt ad.

```
Employee employee = // load
EmployeeDto dto = modelMapper.map(employee, EmployeeDto.class);
```

Lista esetén kicsit bonyolultabb:

```
List<Employee> employees = // load
java.lang.reflect.Type targetListType = new TypeToken<List<EmployeeDto>>()
{}.getType();
List<EmployeeDto> dtos = modelMapper.map(employees, targetListType);
```

A ModelMapper használatához fel kell venni függőségként:

```
<dependency>
  <groupId>org.modelmapper</groupId>
  <artifactId>modelmapper</artifactId>
  <version>${modelmapper.version}</version>
</dependency>
```

Valamint ahhoz, hogy injektálni lehessen, fel kell venni beanként is, pl. az `Application` osztályba.

```
@Bean
public ModelMapper modelMapper() {
  return new ModelMapper();
}
```

REST webszolgáltatások - GET művelet

A Spring Boot 2.4.4 verzióban lévő Lombok nem működik a Java 16-tal. Ezért a `pom.xml`-ben újabbat kell megadni:

```
<dependency>
  <groupId>org.projectlombok</groupId>
  <artifactId>lombok</artifactId>
  <version>1.18.20</version>
</dependency>
```

GET műveletek paraméterezése

REST webszolgáltatások esetén az alkalmazásban található erőforrásokat az URL-ek azonosítják. Ezért szeretnénk az URL-ek által biztosított előnyöket kihasználni, vagyis például URL-paramétereket használni, valamint szeretnénk, hogy az URL bizonyos részei dinamikusan változtathatóak legyenek. Ezt természetesen támogatja a Spring MVC.

Ahhoz, hogy az URL-nek paramétereket adhassunk át, a controller osztály metódusában fel kell vennünk egy paramétert és ellátni a `@RequestParam` annotációval. Arról is dönthetünk, hogy ez egy kötelező paraméter legyen vagy csak opcionális. Utóbbi esetben két megoldást választhatunk. Vagy a `@RequestParam` annotációnak beállítjuk paraméterül, hogy `required = false`. Ekkor ha nem adunk meg értéket, akkor az alapértelmezett értéke `null` lesz, ha viszont kötelező megadni (`required = true`), és mégsem kerül megadásra, akkor hibajelzést fogunk kapni. Másik eset, hogy az adott paramétert `Optional` típussal vesszük föl, és innentől kezdve a Spring MVC már tudni fogja, hogy ezt nem kötelező kitölteni. A paraméterek megadásakor a háttérben automatikus típuskonverzió történik, vagyis oda kell figyelni arra, hogy vagy az elvárt, vagy az elvártra konvertálható típusú értéket kell megadni, különben itt is hibajelzést kaphatunk.

```
public List<EmployeeDto> listEmployees(@RequestParam Optional<String> prefix)
{
	return employeesService.listEmployees(prefix);
}
```

Fontos elvárás az is, hogy URL részleteket adhassunk meg. Ebben az esetben `@PathVariable` annotációt kell használnunk, és a `@GetMapping` annotáció paramétereként, amikor definiáljuk, hogy milyen URL-en legyen elérhető az adott metódus, akkor ott egy kapcsos zárójelekkel megadott placeholdert kell használnunk. Természetesen itt is működik a típuskonverzió.

```
@GetMapping("/{id}")
public EmployeeDto findEmployeeById(@PathVariable("id") long id) {
    return employeesService.findEmployeeById(id);
}
```

A Twelve Factor App ezekre a dolgokra az “API first” címszó alatt tartalmaz útmutatást. Ez azt jelenti, hogy amikor fejlesztjük az alkalmazást, akkor ezt úgy tegyük, hogy először alkossuk meg az API leírást, és utána implementáljuk mögé a megoldást. Ennek vannak vitatói, akik szerint ez éppen fordítva tud jól működni, természetesen mindkét megoldásnak vannak előnyei és hátrányai is. Fontos, hogy ez az API nagyon jól tesztelt és dokumentált legyen, erre vannak megfelelő eszközök, sőt, már szabványok is.

A másik idevonatkozó dolog, amit a Twelve-Factor App tartalmaz: az adott alkalmazásunk állapotmentes folyamatokból álljon. Állapot csak egy kérésen belül létezhet, de két kérés között már nem lehet állapot, vagyis nem tárolhatunk a memóriában semmit, csak addig, amíg egy kérés aktív. Ez több előnnyel is jár, például nem baj, ha valamiért elveszik a legutolsó lépés, mert csak azt kell majd pótolni, és nem az egész addigi folyamatot (például ha egy webáruházban sikertelen a legutolsóként kiválasztott termék kosárba tétele, akkor nem veszik el a kosár egész addigi tartalma). Ezenkívül nem kell clusterezni, kevesebb erőforrást használnak így a műveletek, és nem merülnek föl párhuzamossági problémák sem. Viszont valamilyen módon mindenképpen szeretnénk eltárolni az eddigi folyamat eredményét (a példánál maradva: azt, hogy mi került eddig a kosárba), ezt *backing service*-ben kell tárolni és nem az alkalmazásban, tehát adatbázisban vagy egy elosztott cache-ben.

Ha állapotmentesen dolgozunk, vagyis nem veszünk fel az osztályainkba attribútumokat, amikben állapotot tárolunk, hanem az értékeket kizárólag paraméterként adjuk át, akkor nem lesznek konkurencia problémáink. Ez lehetővé teszi a horizontális skálázást is, mivel így a HTTP kéréseket kiszolgáló szálak függetlenek lesznek egymástól, ezáltal ezek akár külön számítógépen is lehetnek. Ha mégis szükséges valami miatt konkurenciakezelés, akkor azt backing service-ben kell intézni.

REST webszolgáltatások - POST és DELETE művelet

Amennyiben erőforrást akarunk lekérni, a HTTP GET metódust kell használnunk. Ha viszont erőforrást szeretnénk felvinni vagy módosítani, a POST és a PUT művelet lesz a segítségünkre. Sokszor úgy használják ezeket, hogy a POST-tal történik az *update*, a PUT-tal pedig a *create*, azonban a HTTP szabványban ez nem így van benne. Valójában a két művelet közötti különbség abban nyilvánul meg, hogy a PUT idempotens, a POST pedig nem, azaz ha a PUT-ot többször egymás után elvégezzük, akkor mindig ugyanazt az eredményt fogjuk visszakapni, az adatbázis ugyanolyan állapotban lesz. A POST többszöri meghívásakor viszont a művelet minden egyes alkalommal végbe is megy, azaz minden művelet után más állapot marad az adatbázisban.

> Bár a videó szerint a mind a *create*, mind az *update* művelet végezhető POST-tal és PUT-tal is, a különbség mindössze annyi, hogy ha a kérést az adatbázis felé “id” nélkül küldjük el, akkor egy új erőforrást szeretnénk létrehozni, ha viszont az elküldött kérés tartalmaz “id”-t is, akkor pedig egy már meglévő erőforrást

szeretnénk módosítani. Azonban érdemes a POST-tal erőforrást létrehozni, és PUT-tal pedig módosítani.

A Java metóduson a `@PostMapping` vagy a `@PutMapping` annotációk használhatóak.

```
@PostMapping
public EmployeeDto createEmployee(
    @RequestBody CreateEmployeeCommand command) {
  return employeesService.createEmployee(command);
}

@PutMapping("/{id}")
public EmployeeDto updateEmployee(
    @PathVariable("id") long id,
    @RequestBody UpdateEmployeeCommand command) {
  return
    employeesService.updateEmployee(id, command);
}
```

Ahhoz, hogy a HTTP kérés törzsében adatot lehessen beküldeni, a `@RequestBody` annotációt kell használnunk. Az adatot JSON formátumban kell beküldeni, és a HTTP válaszban visszakapott adat is ilyen módon érkezik meg.

Ha törölni szeretnénk, akkor a Java metódusra a `@DeleteMapping` annotációt kell tenni.

```
@DeleteMapping("/{id}")
public void deleteEmployee(@PathVariable("id") long id) {
    employeesService.deleteEmployee(id);
}
```

Státuszkódok és hibakezelés

Ha egy művelet sikeresen végrehajtásra kerül, akkor HTTP 200-as státuszkóddal térünk vissza, ami azt jelenti, hogy a művelet hiba nélkül lefutott. De nem mindig van ez így, hanem van olyan eset, amikor a művelet futtatása közben valami hiba történik. Ekkor valami olyan státuszkóddal kell visszatérnünk, amellyel jelezzük, hogy nem úgy történt minden, ahogy szerettük volna.

A különböző hibakódoknak jelentésük van. A 300-as kódok azt jelentik, hogy alapvetően jó volt a kérés, de a kliensnek még valami teendője van ahhoz, hogy a kérés lefuttatása végbe tudjon menni (például be kell jelentkeztetni vagy át kell irányítani valahova). A 400-as hibakódok azt jelentik, hogy a kliens valamilyen rossz adatot küldött (például a 404-es hibakód azt jelenti, hogy a kliens olyan erőforrást kért le, ami nem található). Az 500-as státuszkódok pedig azt jelentik, hogy a hiba a szerver oldalán található.

Ha státuszkódot szeretnénk kapni a metódus visszatérési értékeként, akkor `ResponseEntity`-t kell visszaadnunk és nem pedig DTO-t. `ResponseEntity` példányokat fluent builder API-val lehet létrehozni, például a `ResponseEntity.ok()` metódus 200-as státuszkódot ad vissza, a `ResponseEntity.notFound()` pedig 404-est.

```
@GetMapping("/{id}")
public ResponseEntity findEmployeeById(@PathVariable("id") long id) {
    try {
        return ResponseEntity.ok(employeesService.findEmployeeById(id));
    }
    catch (IllegalArgumentException iae) {
        return ResponseEntity.notFound().build();
    }
}
```

Ha a metódusra rátesszük a `@ResponseStatus(HttpStatus.XXX)` annotációt, akkor az annotációnak megadhatjuk paraméterül, hogy milyen státuszkóddal térjen vissza a metódus (pl.: a `HttpStatus.CREATED` a 201-es státuszkódot jelenti, azaz azt, hogy minden rendben, az adatbázisban létrejött az új erőforrás.)

```
@PostMapping
@ResponseStatus(HttpStatus.CREATED)
public EmployeeDto createEmployee(
    @RequestBody CreateEmployeeCommand command) {
  return employeesService.createEmployee(command);
}
```

Abban az esetben, ha törlés van, nem a 200-as státuszkóddal érdemes visszatérni, hanem a 204-essel, ami azt jelzi, hogy nem hiba miatt üres a HTTP válasz törzse, hanem ez a kérés normál lefutásának az eredménye, ekkor a `@ResponseStatus(HttpStatus.NO_CONTENT)` annotációt kell használni.

```
@DeleteMapping("/{id}")
@ResponseStatus(HttpStatus.NO_CONTENT)
public void deleteEmployee(@PathVariable("id") long id) {
    employeesService.deleteEmployee(id);
}
```

Ha eldobásra kerül egy kivétel, akkor a Spring MVC ezt elkapja, és az alább találhatóhoz hasonló JSON-t küld vissza a kliens oldalra, 400-as státuszkóddal:

```
{
  "timestamp": 1596570258672,
  "status": 500,
  "error": "Internal Server Error",
  "message": "",
  "path": "/api/employees/3"
}
```

Nézzük meg, milyen lehetőségek vannak a Spring MVC-ben, ha saját magunk akarjuk kezelni a hibát!

- Minden Javas webes alkalmazásnál a `web.xml` állományban lehet deklarálni, hogy hányas státuszkódú hibáknál és milyen kivételeknél milyen másik oldalra történjen az átirányítás.

- A Spring keretrendszeren belül rá lehet tenni egy saját Exceptionre a `@ResponseStatus` annotációt, paraméterül megadva neki a státuszkódot, így ha egy ilyen kivételt kap el a Spring, akkor automatikusan ezt a státuszkódot fogja visszaadni.
- Lehet globálisan `ExceptionResolver` osztályokat definiálni.
- Egy adott controlleren belül lehet lokálisan is lehet kivételeket kezelni `@ExceptionHandler` annotációval ellátott metódusokkal.
- Globálisan lehet készíteni olyan, `@ControllerAdvice` annotációval ellátott osztályokat, amelyekben `@ExceptionHandler` annotációval ellátott hibakezelő metódusok találhatóak.

```
@ExceptionHandler(IllegalArgumentException.class)
@ResponseStatus(value = HttpStatus.NOT_FOUND)
public void handleNotFound() {
    System.out.println("Employee not found");
}
```

A hibakezelésre van egy szabvány is: **RFC 7807 Problem Details for HTTP APIs**, amely azt deklarálja, hogy ha valami probléma van, akkor a választ egy `application/problems+json` mime-type-pal kell visszaadni, amely az alább következő formátumban kell, hogy kinézzen:

```
{
    "type": "employees/invalid-json-request",
    "title": "JSON error",
    "status": 400,
    "detail": "JSON parse error: Unexpected character..."
}
```

Ennek a következő kötelező mezői vannak:

- `type`: URI, az adott hiba típusának egyedi azonosítója, amelyet mi magunk deklarálhatunk.
- `title`: rövid szöveges leírás (ember által olvasható formában).
- `status`: maga a HTTP státuszkód, amely visszamegy a kliensnek.
- `detail`: részletesebb, hosszabb leírás (szintén ember által olvasható formában).
- Opcionálisan be lehet tenni egy `instance` mezőt, amely a hibának az egyedi elérhetőségét tartalmazza, például egy naplózó rendszerbe vezető URL.
- Saját, egyedi mezők is definiálhatók.

Javaban több különböző implementációja is létezik ennek a szabványnak, például a **Problem** nevű third party library, amelynek használatához a következőket kell a `pom.xml`-be függőségként felvenni:

```
<dependency>
  <groupId>org.zalando</groupId>
  <artifactId>problem</artifactId>
  <version>${problem.version}</version>
</dependency>
<dependency>
  <groupId>org.zalando</groupId>
```

```
  <artifactId>jackson-datatype-problem</artifactId>
  <version>${problem.version}</version>
</dependency>
```

Az alább található metódusban a kivételt a metódus paramétereként definiáljuk, a metódus törzsében létrehozunk egy `Problem` példányt, megadjuk neki a *Problem Details* szabvány alapján kötelező mezőket, és a végén, egy `ResponseEntity`-be csomagolva a `Problem`-et, a 404-es státuszkóddal térünk vissza a metódusból.

```
@ExceptionHandler({IllegalArgumentException.class})
public ResponseEntity<Problem> handleNotFound(IllegalArgumentException  e) {
    Problem problem = Problem.builder()
            .withType(URI.create("employees/employee-not-found"))
            .withTitle("Not found")
            .withStatus(Status.NOT_FOUND)
            .withDetail(e.getMessage())
            .build();

    return ResponseEntity
      .status(HttpStatus.NOT_FOUND)
    .contentType(MediaType.APPLICATION_PROBLEM_JSON)
      .body(problem);
}
```

Ahhoz, hogy a HTTP válaszban megfelelő JSON formátumban kerüljön kiírásra a `Problem` példány, egy `@Configuration` annotációval ellátott osztályban meg kell adni a következő metódust:

```
@Bean
public ObjectMapper objectMapper() {
  return new ObjectMapper()
  .findAndRegisterModules();
}
```

Integrációs tesztelés

Ha a webszolgáltatásainkat szeretnénk tesztelni, akkor erre több lehetőségünk is van. A Springgel el lehet indíttatni kizárólag a controller réteget (ekkor a service réteget mockolni kell), de elindíthatjuk a teljes alkalmazást is. Ez utóbbit is két módon tehetjük meg: vagy valódi konténerrel (ebben az esetben fut egy Tomcat, lefoglal egy portot és ugyanúgy lehet megszólítani, mint egy teljes alkalmazást), vagy pedig konténer nélkül (ekkor ki lesz mockolva a Servlet API).

Ez utóbbi esetben nem valódi HTTP kérések közlekednek, hanem egyszerű metódushívások vannak a háttérben.

A controller réteg teszteléséhez a `@WebMvcTest` annotációt kell használni.

A mock objektumokat a `@MockBean` annotációval kell ellátni és a Mockito keretrendszerrel lehet létrehozni és paraméterezni őket. Ezenkívül használható a **MockMvc** keretrendszer, amely a Springen belül található. Ez arra használható, hogy kéréseket lehet beküldeni, a válaszokat lehet ellenőrizni, a választ lehet naplózni, illetve a program a választ különböző formátumokban képes feldolgozni. A Spring ebben az esetben nem indít valódi konténert, hanem a Servlet API-t teljes mértékben kimockolja, a JSON szerializácó viszont teljes mértékben megtörténik.

```
import static
org.springframework.test.web.servlet.request.MockMvcRequestBuilders.*;
import static
org.springframework.test.web.servlet.result.MockMvcResultHandlers.*;
import static
org.springframework.test.web.servlet.result.MockMvcResultMatchers.*;
import static org.hamcrest.Matchers.equalTo;
import static org.mockito.ArgumentMatchers.any;
import static org.mockito.Mockito.when;

@Test
void testListEmployees() throws Exception {
    when(employeesService.listEmployees(any())).thenReturn(List.of(
            new EmployeeDto(1L, "John Doe"),
            new EmployeeDto(2L, "Jane Doe")
    ));

    mockMvc.perform(get("/api/employees"))
      .andDo(print())
            .andExpect(status().isOk())
            .andExpect(jsonPath("$[0].name", equalTo("John Doe")));
}
```

Amint látható, egy ilyen tesztesetet úgy kell implementálni, hogy az első részében a service réteget mockoljuk, a második részében pedig a MockMvc segítségével vizsgáljuk, hogy az elküldött HTTP kérésre azt a választ kaptuk-e, amit vártunk.

Abban az esetben, ha a teljes alkalmazást szeretnénk tesztelni, de konténer nélkül, a `@SpringBootTest` annotációt használjuk, az `@AutoConfigureMockMvc` annotációt pedig azért, hogy a MockMvc keretrendszer automatikusan fel legyen konfigurálva. Ebben az esetben nem kell mockolni a service osztályunkat, mert a valódi service-t használjuk, az kerül lepéldányosításra.

Ha viszont konténeren belül szeretnénk tesztelni a teljes alkalmazást, vagyis azt szeretnénk, hogy a teszteset elindítson egy Tomcat konténert is, a tesztosztályra a `@SpringBootTest()` annotációt kell tenni és paraméterül meg kell adni, hogy ez a konténer egy random porton kerüljön elindításra (amelyik éppen szabad).

```
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
```

Amennyiben tudni szeretnénk, hányas porton fut a konténer, akkor a port számát a `@LocalServerPort` annotációval injektálhatjuk a tesztesetbe.

Ezenkívül van egy **RestTemplate** keretrendszer is a Springen belül, amely REST kliensként működik, tehát Javaban tudunk vele REST kéréseket kiadni, és ennek kis van egy teszt verziója, amelyet automatikusan injektálni lehet (`TestRestTemplate`). Ekkor a háttérben `RestTemplate`-et használunk, megtörténik a JSON kommunikáció, és maga a `RestTemplate` alakítja vissza Java objektummá a JSON dokumentumot.

```
@Test
void testListEmployees() {
  List<EmployeeDto> employees =
    restTemplate.exchange("/api/employees",
      HttpMethod.GET,
      null,
      new ParameterizedTypeReference<List<EmployeeDto>>(){})
    .getBody();

  assertThat(employees)
          .extracting(EmployeeDto::getName)
          .containsExactly("John Doe", "Jane Doe");
}
```

Ebben a tesztesetben először a `RestTemplate`-nek meghívjuk a `getForObject()` metódusát, amelynek két paramétert kell átadni, egy URL-t és azt, hogy milyen típusú visszatérési értékre számítunk. Ezután egy asserttel vizsgáljuk a visszatérési értéket.

Swagger UI

A **Swagger UI** egy régóta létező third party library. Arra való, hogy a RESTful webszolgáltatásokhoz egy felületet, egy API dokumentációt lehet vele generálni. Ráadásul ez nem csak egy statikus oldal, hanem még arra is lehetőséget biztosít, hogy a REST webszolgáltatásokat ott ki is tudjuk próbálni. Ezenkívül képes arra is, hogy *OpenAPI Specification* szabványnak megfelelő API leírást generáljon le. Ez egy JSON vagy YAML formátumú egyszerű dokumentum, ami leírja a webszolgáltatásokat. Leírja tehát, hogy az erőforrások milyen URL-en érhetőek el, milyen paramétereket lehet átadni, milyen formátumú adatokat lehet beküldeni, illetve milyen formátumban adja vissza a lekért adatokat, valamint hogy milyen metódusokat használhatunk lekérésre, létrehozásra, módosításra, törlésre. Az OpenAPI Specificatont a Swagger fejlesztői hozták létre, régen ezt *Swagger Specification*-nek hívták.

A Swagger UI keretrendszer-független. Nem kapcsolódik a Springhez, hanem van Spring-integrációja is. A forráskód alapján képes legenerálni a dokumentációt, de ha szeretnénk bele egyéb leírásokat is tenni, akkor ezt annotációk használatával tehetjük meg.

Ezenkívül létezik még a *Springdoc-OpenAPI project*, ami egy integrációs projekt, és arra való, hogy megkönnyítse a Spring és a Swagger UI összekapcsolását. Ha ezt felvesszük függőségként a `pom.xml`-be az alább látható módon, akkor a Swagger UI is automatikusan használható és elérhető a `/swagger-ui.html` címen. Az OpenAPI Specificationnek megfelelő

dokumentum ekkor a `/v3/api-docs` címen JSON formátumban, a `/v3/api-docs.yaml` pedig YAML formátumban érhető el.

```
<dependency>
    <groupId>org.springdoc</groupId>
    <artifactId>springdoc-openapi-ui</artifactId>
    <version>1.5.9</version>
</dependency>
```

Van lehetőség ezt a dokumentumot személyre szabni is. A globális részeket (a dokumentum elejét) úgy lehet testreszabni, hogy egy `@Configuration` annotációval ellátott osztályban felveszünk egy `OpenAPI` típusú beant (`@Bean` annotációval ellátva), és ennek az `info()` metódusával lehet megadni például a dokumentum címét, verziószámát vagy akár egy hosszabb szöveges leírást is.

```
@Bean
public OpenAPI customOpenAPI() {
  return new OpenAPI()
  .info(new Info()
  .title("Employees API")
  .version("1.0.0")
  .description("Operations with employees"));
}
```

A Swagger UI figyelembe veszi a Bean Validation annotációkat is, tehát például egy olyan attribútumot, amelyen a `@NotNull` annotáció szerepel, piros csillaggal jelöl meg a dokumentációban, hogy azt nem lehet üresen hagyni. Személyreszabható az is a `@Schema` annotációval, hogy az egyes attribútumok mellett mi jelenjen meg névként a felhasználói felületen, sőt, egy értéket is meg lehet adni példaként. Ezt a példa értéket a Swagger bele fogja generálni a dokumentációba is, és ez segítségünkre lehet az API kipróbálásában is.

```
public class CreateEmployeeCommand {

    @NotNull
    @Schema(description="name of the employee", example = "John Doe")
    private String name;
}
```

A Swagger figyelembe veszi a Spring MVC annotációkat is, például egy `@GetMapping` annotációval ellátott metódusnál a felületen is az fog megjelenni, hogy egy HTTP GET metódussal lekérdezhető erőforrásról van szó. Meg lehet viszont mondani azt, hogy ne jelenjen meg a dokumentációban az osztály neve (pl. `EmployeesController`), hanem a `@Tag` annotációval lehet megadni egyedi elnevezést. Magáról az adott metódusról az `@Operation` annotációval tudunk megadni rövidebb és hosszabb leírást. Meg lehet adni azt is, hogy a metódus milyen státuszkódokkal térjen vissza. Itt a Swagger figyelembe veszi a Spring MVC konfigurációt, vagyis tudja, hogy ez a metódus sima lefutás esetén milyen státuszkóddal tér vissza, és ezt külön definiálás nélkül is belegenerálja a dokumentációba. De `@ApiResponse` annotációval felvehetjük azt is, hogy a metódus milyen további státuszkódokkal képes még visszatérni, ezek mit jelentenek, és a dokumentációban természetesen ez is meg fog jelenni.

```
@RestController
@RequestMapping("/api/employees")
@Tag( name = "Operations on employees")
public class EmployeesController {

  @GetMapping("/{id}")
  @Operation(summary = "Find employee by id",
    description = "Find employee by id.")
  @ApiResponse(responseCode = "404",
    description = "Employee not found")
  public EmployeeDto findEmployeeById(
      @Parameter(description = "Id of the employee",
        example = "12")
      @PathVariable("id") long id) {
    // ...
  }

}
```

Tesztelés Rest Assured használatával

A **Rest Assured** egy különálló third party library, amivel REST webszolgáltatásokat tudunk tesztelni. Teljes mértékben keretrendszer-független, de integrálható a Springgel is. A dinamikus nyelvek egyszerűségét próbálja hozni Java nyelven egy fluent API-val. Képes kezelni a JSON és XML dokumentumokat, tud ezekből Java objektumokat gyártani, illetve Java objektumokból is tud dokumentumot készíteni. A mögötte lévő szerializációs keretrendszer cserélhető (használható Jackson, Gson, a JAXB valamelyik implementációja, stb.)

HTTP kérés összeállításakor meg lehet adni az URL-t, paramétereket, headert, cookie-kat, content-type-ot, stb., és a HTTP válasszal kapcsolatban is mindent lehet ellenőrizni, például a státuszkódot vagy az üzenet tartalmát. A Rest Assured támogatja a különböző autentikációs módokat is.

Nézzük meg, miket lehet ellenőrizni!

- Ha XML tartalom jön vissza, akkor XMLPath nyelven lehet megadni lekérdezéseket. Ez valójában egy, a Groovy nyelvből ismert GPath nyelvű lekérdezés, hasonló az XPath-hoz. Ezenkívül XML esetén még lehet DTD és XSD validációkat is futtatni.
- JSON tartalom esetén JSONPath kifejezéseket is lehet kiértékeltetni, és ezeknek az eredményét ellenőrizni. Ez esetben JSON Schema alapján lehet validációkat végezni.
- Természetesen a HTTP egyéb dolgaira is lehet ellenőrzéseket végezni (header, státuszkód, cookie, content-type, stb.)
- Lehet azt is ellenőrizni, hogy egy adott hívás mennyi idő alatt jön vissza (response time).

Ahhoz, hogy a Rest Assuredöt használni tudjuk, fel kell venni a `pom.xml`-be az alábbiakat függőségként:

```
<dependency>
  <groupId>io.rest-assured</groupId>
  <artifactId>json-path</artifactId>
  <version>${rest-assured.version}</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>io.rest-assured</groupId>
  <artifactId>xml-path</artifactId>
  <version>${rest-assured.version}</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>io.rest-assured</groupId>
  <artifactId>rest-assured</artifactId>
  <version>${rest-assured.version}</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>io.rest-assured</groupId>
  <artifactId>spring-mock-mvc</artifactId>
  <version>${rest-assured.version}</version>
  <scope>test</scope>
</dependency>
```

Érdemes egy tesztosztályban úgy inicializálni, hogy ne kelljen minden egyes HTTP kérésnél beállítani, hogy JSON-t küldünk és JSON-t várunk, tehát például egy `@BeforeEach` annotációval ellátott metódusban kell meghívni a `contentType()` és az `accept()` metódusát. Ehhez még a `WebApplicationContext`-et kell injektálni a tesztosztályba és átadni a `RestAssuredMockMvc`-nek.

```
import io.restassured.http.ContentType;
import io.restassured.module.mockmvc.RestAssuredMockMvc;

import static
io.restassured.module.jsv.JsonSchemaValidator.matchesJsonSchemaInClasspath;
import static io.restassured.module.mockmvc.RestAssuredMockMvc.*;
import static org.hamcrest.Matchers.equalTo;

@Autowired
WebApplicationContext webApplicationContext;

@BeforeEach
void init() {
  RestAssuredMockMvc.requestSpecification = given()
    .contentType(ContentType.JSON)
    .accept(ContentType.JSON);
```

```
  RestAssuredMockMvc
    .webAppContextSetup(webApplicationContext);
}
```

Egy teszteset az alábbi módon néz ki:

```
@Test
void testCreateEmployeeThenListEmployees() {
    with().body(new CreateEmployeeCommand("Jack Doe")).
    when()
      .post("/api/employees")
      .then()
      .body("name", equalTo("Jack Doe"));

    when()
      .get("/api/employees")
      .then()
      .body("[0].name", equalTo("Jack Doe"));
}
```

Itt először, a teszteset *Given* részében a `with()` metódussal meg kell adni, hogy mit fogunk beküldeni ezt a Rest Assured leképezi JSON dokumentummá. Utána a *When* részben kell leírni, hogy mit csinálunk, majd a `then()` metódus hívásával lehet megvizsgálni, hogy milyen választ kaptunk. (Jelen példában egy `CreateEmployeeCommand`-ot küldünk be POST-tal a `/api/employees` címre, a `then()` metódussal lekérjük a választ, és végül egy JSONPath kifejezéssel megvizsgáljuk, hogy a visszakapott JSON-ben lévő `name` mező értéke megegyezik-e az általunk megadott névvel (`Jack Doe`). A teszteset további részében egy újabb `when()` metódushívással lekérjük a `/api/amployees` címet. Itt egy tömböt kapunk vissza, és ebből a tömbből kérdezzük le az első elem `name` mezőjét, vizsgálva, hogy ez megegyezik-e az általunk elvárttal.)

Rest Assured séma validáció

A Rest Assured képes séma validációra is, tehát képes egy *JSON schema* alapján validálni egy JSON dokumentumot. Vagyis azt ellenőrzi, hogy egy JSON dokumentum megfelel-e a JSON sémában leírt szabályoknak, struktúrának. Ahhoz, hogy ezt használni tudjuk, a `pom.xml`-be fel kell venni függőségként a következőt:

```
<dependency>
    <groupId>io.rest-assured</groupId>
    <artifactId>json-schema-validator</artifactId>
    <scope>test</scope>
</dependency>
```

Nézzük meg, hogyan kell egy JSON sémát leírni! Az alább látható JSON séma azt a JSON dokumentumot írja le, amelyet visszaad az a REST webszolgáltatásunk, amely kilistázza az alkalmazottakat:

```
{
  "$schema": "https://json-schema.org/draft/2019-09/schema",
  "title": "Employees",
  "type": "array",
  "items": [
    {
      "title": "EmployeeDto",
      "type": "object",
      "required": ["name", "id"],
      "properties": {
        "id": {
          "type": "integer",
          "description": "id of the employee",
          "format": "int64",
          "example": 12
        },
        "name": {
          "type": "string",
          "description": "name of the employee",
          "example": "John Doe"
}}}]}
```

Ez valójában egy tömb, amelyben objektumok vannak, és ezen objektumok mindegyike tartalmaz egy `id` és egy `name` mezőt. A sémában benne van, hogy egy tömböt várunk (`"type": "array"`), amelyben az elemek az `"items"` mezőn belül vannak részletesen deklarálva. Külön érdekesség, hogy példa értéket is meg lehet adni (`example` mező), amely aztán a dokumentációba is belekerül majd, vagyis azonnal kipróbálható lesz a REST webszolgáltatás.

Maga a séma validáció a következő módon néz ki:

```
when()
     .get("/api/employees")
     .then()
     .body(matchesJsonSchemaInClasspath("employee-dto-schema.json"));
```

Itt a `when()` metódus hívásával lekérjük az `/api/amployees` címet, majd a `then()`-részben meg kell hívni a `matchesJsonSchemaInClasspath()` metódust, és paraméterül át kell adni neki a séma dokumentumot. Fontos, hogy a séma dokumentum ez esetben a classpath-on legyen megtalálható.

Content Negotiation

A content negotiation egy olyan mechanizmus, amely azt teszi lehetővé, hogy a kliens meg tudja mondani, hogy az adott erőforrást milyen formátumban szeretné megkapni. Az egyéni preferenciák függhetnek például attól, hogy az adott kliens milyen formátumú dokumentumot képes feldolgozni.

Ezt úgy lehet megoldani, hogy a kliens egy `Accept` nevű HTTP-fejlécben elküld egy Mime Type-ot, amelyben megadja a válaszként kívánt formátumot (nagyon sokféle formátum elképzelhető). Ezenkívül azt is lehetővé teszi ez a mechanizmus, hogy nyelvet is lehessen váltani, tehát a kliens megmondhatja az `Accept-Language` headerben, hogy milyen nyelven várja a választ.

Ennek a megvalósítására a Spring Boot is támogatást nyújt. A controllerben a `@RequestMapping` annotáció paramétereként lehet megadni, hogy az adott metódus milyen formátumban fogja visszaadni a tartalmat. Itt többféle formátumot is meg lehet adni. Innentől kezdve a Spring elvégzi helyettünk a visszakapott válasz megfelelő formátumba való leképezését.

```
@RequestMapping(value = "/api/employees",
  produces = {MediaType.APPLICATION_JSON_VALUE,
    MediaType.APPLICATION_XML_VALUE})
```

XML formátumba való szerializációhoz a JAXB valamely implementációját fel kell vennünk függőségként a `pom.xml`-be.

```
<dependency>
  <groupId>org.glassfish.jaxb</groupId>
  <artifactId>jaxb-runtime</artifactId>
</dependency>
```

valamint a Dto osztályainkra rá kell tenni az `@XmlRootElement` annotációt, amelyből a JAXB tudni fogja, hogy ezt XML-ben le lehet képezni. (JSON-nél ilyenre azért nincs szükség, mert a Springes projekt létrehozásakor a `pom.xml`-be már automatikusan bekerül a Jackson, amely pedig a JSON szerializációért felelős.)

XML esetén abból adódhat nehézség, ha a metódus egy kollekcióval tér vissza. Ha ezt személyre akarjuk szabni, hogy ez XML-ben hogy is nézzen ki, akkor ezt mindenképpen be kell burkolni egy Java osztályba. Itt az osztályon az `@XmlRootElement` annotációt, a metóduson pedig az `@XmlElement` annotációt kell használni. Ezek paramétereként tudjuk megadni az XML tagek neveit.

```
@Data
@NoArgsConstructor
@AllArgsConstructor
@XmlRootElement(name = "employees")
@XmlAccessorType(XmlAccessType.FIELD)
public class EmployeesDto {

    @XmlElement(name = "employee")
    private List<EmployeeDto> employees;
}
```

Ebben a kódrészletben szerepel még a Dto osztályon az `@XmlAccessorType()` annotáció. Ez azért van, mert a JAXB alapértelmezetten a getter metódusokra rátett annotációkat veszi figyelembe, de mivel itt ezeket helyettünk a Lombok implementálja (és ezért nem lehet ezeket annotálni), ezért az attribútumra tettük rá az `@XmlElement` annotációt. Viszont ezt

meg is kell mondani a JAXB-nek, ezért szerepel az osztályon az `@XmlAccessorType(XmlAccessType.FIELD)` annotáció.

Validáció

A Spring Boot támogatja a bemeneti adatok ellenőrzését. Ehhez beépítetten tartalmaz **Bean Validation 2.0 (JSR 380)** támogatást. Ez egy különálló szabvány, amely arra való, hogy különböző Java objektumokhoz tartozó ellenőrzéseket, validációkat lehet definiálni. Úgy alkották meg, hogy maga a validáció ne réteghez legyen kötve, hanem magán az adaton legyen megadva, az adatot hordozó beanen. Egy Springes alkalmazás esetén tehát azokon a Dto-kon, amelyek a controller metódus paramétereként szerepelnek (vagyis a bemeneti paramétereken). Ennek használatához a következő függőséget kell felvenni a `pom.xml`-be:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-validation</artifactId>
</dependency>
```

Validációt megadni annotációk használatával lehetséges, tehetjük őket az adott Dto-k mezőire vagy akár az osztályra is. Vannak beépített annotációk, de sajátot is lehet írni. Meg lehet mondani azt is, hogy a validáció mikor fusson. A validációk meghatározása magán a Dto-n vagy az entitáson történik, és később lehet megmondani, hogy ezek hol kerüljenek meghívásra.

Beépített annotációk:

- `@AssertTrue` és `@AssertFalse` - Egy `boolean` típusú adat esetén lehet vizsgálni, hogy `true` vagy `false`-e az értéke.
- `@Null` és `@NotNull` - Referencia típusú adat esetén lehet megmondani, hogy az adott mező értéke lehet-e `null` vagy nem.
- `@Size` - String vagy kollekció esetén meg lehet határozni méretkorlátokat.
- `@Max`, `@Min`, `@Positive`, `@PositiveOrZero`, `@Negative`, `@NegativeOrZero`, `@DecimalMax`, `@DecimalMin`, `@Digits` - Egész- és lebegőpontos számoknál meg lehet határozni minimum és maximum értékeket, elvárást az előjelre vonatkozóan, illetve azt is, hogy hány számjegyből álljon.
- `@Future`, `@Past`, `@PastOrPresent`, `@FutureOrPresent` - Különböző dátum és idő típusú attribútumoknál meg lehet határozni, hogy az a múltban vagy a jövőben legyen.
- `@Pattern` - Meg lehet határozni, hogy egy string feleljen meg egy reguláris kifejezésnek.
- `@Email` - Meg lehet nézni egy stringről, hogy e-mail cím formátumú-e.
- `@NotEmpty` - Meg lehet határozni, hogy egy string vagy kollekció nem lehet üres.
- `@NotBlank` - Meg lehet határozni, hogy egy string nem lehet üres, `null`, illetve nem tartalmazhat csupa whitespace karaktereket.

Először a Dto attribútumaira kell tenni az előbbiekben említett annotációk valamelyikét, ezzel meghatározva, hogy mit kell a Springnek vizsgálnia az adott attribútummal kapcsolatban. Ezeknél az annotációknál paraméterként azt is meg lehet határozni, hogy

milyen a hibaüzenetet küldjön vissza a Spring MVC, ha az adat nem felelne meg a validációs kritérium(ok)nak.

```
@Data
@NoArgsConstructor
@AllArgsConstructor
public class CreateEmployeeCommand {

    @NotNull(message = "Name can not be null")
    private String name;
}
```

Ahhoz, hogy a Spring automatikusan megvizsgálja a paramétereket (a Dto-kat) az adott controller metódus meghívásakor, az átadott paraméter elé el kell helyezni a `@Valid` annotációt.

```
@PostMapping
public EmployeeDto createEmployee(
    @Valid @RequestBody CreateEmployeeCommand command) {
  return employeesService.createEmployee(command);
}
```

A validáció lezajlása után a Spring egy JSON-t küld vissza válaszként, amely nagyon részletesen tartalmazza a validáció eredményét. Ez azért nagyon fontos, hogy a felületen mezőnként lehessen megmondani, hogy milyen probléma volt a megadott adatokkal.

```
{
  "timestamp": 1596570707472,
  "status": 400,
  "error": "Bad Request",
  "message": "",
  "path": "/api/employees"
}
```

A Spring által visszaküldött JSON azonban nem szabványos. Ha mi szabványos hibaüzenetet szeretnénk visszaküldeni, akkor használhatjuk a Problem third party library-t. Itt egy `@ExceptionHandler`-t kell deklarálnunk. Ehhez ismernünk kell a Spring belső működését validációs hiba esetén. Először megpróbálja meghívni a metódust, a `@Valid` annotációval ellátott paramétert megpróbálja levalidálni, és ha ez nem sikerül, akkor dob egy `MethodArgumentNotValidException`-t, amelyet nekünk kezelnünk kell. Ettől a kivételtől le lehet kérdezni a `getBindingResult().getFieldErrors()` metódusával a mezőkre vonatkozó hibaüzeneteket. Ehhez először létre kell hozni egy segédosztályt (jelen példában `Violation`), amelynek két attribútuma a mező neve és a hibaüzenet. Majd a Spring által visszaadott hibalistát egy `Violation`-listává konvertáljuk át, és ezt a listát felvesszük attribútumként a `Problem` objektumon belül. A `Problem` metódusai utána ezzel dolgoznak tovább, és végül ezt a Spring egy szabványos JSON dokumentumként adja vissza.

```
@ExceptionHandler({MethodArgumentNotValidException.class})
public ResponseEntity<Problem>
handleValidationError(MethodArgumentNotValidException e) {
```

```
    List<Violation> violations =
e.getBindingResult().getFieldErrors().stream()
            .map((FieldError fe) -> new Violation(fe.getField(),
fe.getDefaultMessage()))
            .collect(Collectors.toList());

    Problem problem = Problem.builder()
            .withType(URI.create("employees/validation-error"))
            .withTitle("Validation error")
            .withStatus(Status.BAD_REQUEST)
            .withDetail(e.getMessage())
            .with("violations", violations)
            .build();

    return ResponseEntity
      .status(HttpStatus.BAD_REQUEST)
      .contentType(MediaType.APPLICATION_PROBLEM_JSON)
      .body(problem);
}
```

```
@Data
@AllArgsConstructor
public class Violation {

    private String field;

    private String defaultMessage;
}
```

A hibát tartalmazó JSON:

```
{
  "type": "employees/validation-error",
  "title": "Validation error",
  "status": 400,
  "detail": "Validation failed for argument [0] in public ...",
  "violations": [
    {
      "field": "name",
      "message": "Name can not be null"
    }
  ]
}
```

Validáció - saját validáció létrehozása

Saját validációs annotáció létrehozásakor először a `@Retention` annotáció paramétereként meg kell adni, hogy mikor kerüljön feldolgozásra (egy legyen futásidőben feldolgozandó!),

majd a `@Target` annotáció paramétereként azt, hogy mire lehessen rátenni (tehát attribútumra, metódusra, stb.) Ezután három kötelező dolgot kell megadni:

- Egy alapértelmezett hibaüzenetet (`message()`), amely megjelenik akkor, ha a bemenő adat nem felel meg a validációs kritériumnak.
- Egy `Class` típusú `groups()`, amely alapértelmezetten egy üres tömb. A Bean Validation ugyanis megengedi, hogy ugyanazon az osztályon meg tudjuk mondani, hogy mely ellenőrzések fussanak le. Pl. más ellenőrzések, ha adatot szeretnénk beszúrni, és más ellenőrzések, ha módosítani. Ehhez használhatóak a validation groupok.
- Valamint lennie kell benne egy ún. `Payload`-nak, melynek az alapértelmezett értéke szintén egy üres tömb. (Ezt nagyon speciális esetekben használjuk, csak fogadjuk el, hogy definiálni kell.)
- Ezeken felül felvehetünk saját paramétereket is alapértelmezett értékkel.
- Ezeket az alapértelmezett értékeket felül lehet bírálni később, az annotáció konkrét használatakor.

```
@Constraint(validatedBy = NameValidator.class)
@Target({METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER})
@Retention(RUNTIME)
public @interface Name {

    String message() default "Invalid name";
    Class<?>[] groups() default {};
    Class<? extends Payload>[] payload() default {};

    int maxLength() default 50;

}
```

Ahhoz, hogy ez tényleg egy validációs annotáció legyen, rá kell tenni a `@Constraint` annotációt, amelynek paraméterül meg kell adni a validálást elvégző osztály nevét. Ennek egy olyan osztálynak kell lennie, amely implementálja a `ConstraintValidator` interfészt. Az interfész neve utáni generikusok között meg kell adni az általunk létrehozott validációs annotációt, valamint hogy milyen típusú adatra lehet ezt az annotációt rátenni. Ennek az interfésznek két metódusa van, amelyet implementálni kell.

```
public class NameValidator implements ConstraintValidator<Name, String> {

  private int maxLength;

  @Override
  public boolean isValid(String value, ConstraintValidatorContext context) {
    return value != null &&
      !value.isBlank() &&
      value.length() > 2 &&
      value.length() <= maxLength &&
      Character.isUpperCase(value.charAt(0));
  }
```

```
    @Override
    public void initialize(Name constraintAnnotation) {
        maxLength = constraintAnnotation.maxLength();
    }
}
```

Konfiguráció és naplózás bevezetés

Ebben a fejezetben megismerheted a Spring Boot egyéni konfigurálási lehetőségeit és azt, hogyan naplózhatod egyszerű módon az alkalmazásod belső eseményeit.

Spring Boot konfiguráció

Az alkalmazások konfigurációjára azért van szükség, mert ugyanazt az alkalmazást akár több környezetben is szeretnénk üzemeltetni. A különböző környezetekre jellemző egyedi konfigurációs értékeket tipikusan külön konfigurációs állományban tároljuk, és nem magán az alkalmazáson belül.

A Spring egy ún. `Environment` absztrakción keresztül támogatja a konfigurációkat, ezen keresztül lehet lekérni a különböző konfigurációs paramétereket. Ezek kulcs-érték párok. Az `Environment` több `PropertySource`-t használ, ami nem más, mint ezeknek a konfigurációknak a forrása, amelyek között prioritás van. Leggyakoribb az `application.properties` állomány, ez az alkalmazás mellett található egyszerű szöveges fájl, amelyet be tud tölteni a Spring Boot. De nem csak properties, hanem YAML formátum is használható, amelyben jobban látszódik a hierarchia.

A források közül az elöl szereplők felülírják a később szereplőket. A források közül, amelyeket a Spring képes felolvasni, a legfontosabbak, amelyeket meg kell említeni:

- A parancssori paraméterek, amelyet a JVM futtatásakor a Java parancsnak tudunk paraméterként átadni.
- Az operációs rendszer környezeti változóinak értékei.
- Az `application.properties` állomány, amely az JAR állomány mellett, a fájlrendszerben található.
- Egy másik `application.properties` állomány, amely a JAR fájlon belül található.

Ez a sorrend prioritási sorrendet is jelent.

Nézzük meg, hogyan kell forráskód-szinten beolvasni ezeket a konfigurációkat! Erre többféle lehetőségünk van:

- Dependency injection formájában a `@Value` annotációnak megadva a kívánt konfigurációs érték kulcsát.
- `Environment`-tel, ami egy Springes interfész, ezt is dependency injection formájában kell átadni egy osztálynak. A `getProperty()` metódusával lehet lekérdezni egy konfigurációs kulcshoz tartozó értéket.

- `@ConfigurationProperties` annotáció használatával, ezt egy olyan osztályra szokás rátenni, amelynek több konfigurációs paramétere is van. Itt használható a `@Validated` annotáció is, amely által a Spring erre az osztályra rá fogja futtatni a Bean Validationt, és ezáltal ellenőrizni lehet azt is, hogy a konfigurációs értékek helyesen lettek-e megadva.

A Spring nagyon sok előre definiált konfigurációs paramétert tartalmaz, az összes third party library-hez találhatunk ilyeneket. Ezek felsorolva megtalálhatóak a *Spring Boot Reference Documentation*-ben, és fel vannak töltve intelligens alapértékekkel. Ez azt jelenti, hogy a Spring a leggyakoribb, fejlesztők által használt beállításokat használja, és ha ettől el akarunk térni, csak akkor érdemes ezeket felüldefiniálni a saját `application.properties` állományunkban.

Felül lehet írni például azt, hogy a Tomcat webkonténer melyik porton induljon el alapértelmezetten.

A Twelve-Factor App erre azt az ajánlást tartalmazza, hogy az alkalmazásba nem szabad "beégetni" a portnak a számát, hanem annak konfigurálhatónak kell maradnia.

Dockerrel a legjobb környezeti változóként megadni a konfigurációs paramétereket.

Elnevezési konvenciók is vannak. A Springben általában a szavakat kisbetűvel szoktuk megadni, és pont (`.`) karakterrel választjuk el őket, míg a környezeti változók nevét nagybetűvel, és aláhúzás karakter (_) az elválasztó. A Spring ezeket automatikusan konvertálja számunkra.

Magával a konfigurációval kapcsolatban is van a Twelve-Factor Appnak ajánlása:

- A környezetenként eltérő értékeket nevezi konfigurációnak. Leggyakoribbak a backing service-ek elérhetőségei, egy adatbázis címe, portja, sémája, felhasználóneve, jelszava, stb. Ide tartozik minden autentikációs konfiguráció, azaz felhasználónevek, jelszavak, titkos kulcsok és tanúsítványok.
- Fontos, hogy a konfigurációs paraméterek a környezet részét kell, hogy képezzék és nem az alkalmazás részét.
- Lehetőleg ne fájlokat használjunk konfigurációs beállításokra, hanem környezeti változókat!
- Kerüljük az alkalmazásban a környezetek nevesítését, hiszen a környezetek változhatnak!
- Fontos a verziókövetés is, hogy láthatóak legyenek a módosítások.

A Spring ezenkívül még tartalmaz további lehetőségeket a konfigurációk beolvasására. A *Spring Cloud Config* egy szerveralkalmazás, amely arra való, hogy konfigurációkat tároljon és kiszolgálja a különböző alkalmazások számára, ráadásul mindezt verziókövetett módon teszi. Ezzel egy központi konfigurációkezelést lehet megvalósítani.

Tegyük fel, hogy a következő konfigurációs paramétert akarjuk felolvasni az `application.properties` állományból:

```
employees.hello = Hello Spring Boot Config
```

Ezt a `@Value` annotáció használatával így lehet:

```
@Service
public class HelloService {

  private String hello;

  public HelloService(@Value("${employees.hello}") String hello) {
    this.hello = hello;
  }

  public String sayHello() {
    return hello + " " + LocalDateTime.now();
  }
}
```

Az `Environment` használatával:

```
@Service
public class HelloService {

  private Environment environment;

  public HelloService(Environment environment) {
    this.environment = environment;
  }

  public String sayHello() {
    return environment.getProperty("employees.hello") + " " +
LocalDateTime.now();
  }
}
```

`@ConfigurationProperties` annotáció használatával:

```
@ConfigurationProperties(prefix = "employees")
@Data
public class HelloProperties {

    private String hello;
}
```

```
@Service
@EnableConfigurationProperties(HelloProperties.class)
public class HelloService {

  private HelloProperties properties;

  public HelloService(HelloProperties properties) {
    this.properties = properties;
```

```
  }

  public String sayHello() {
    return properties.getHello() + " " + LocalDateTime.now();
  }
}
```

Konfiguráció felülírása Docker paranccsal operációs rendszer környezeti változóját használva:

```
docker run -d -p 8080:8081 -e SERVER_PORT=8081 -e EMPLOYEES_HELLO=HelloDocker
employees
```

Spring Boot naplózás

A naplózás egy nagyvállalati alkalmazásnak elengedhetetlen része. A Java-világban elég sok keretrendszer van erre. A legelterjedtebbek a Java Standard Editionben lévő *Java Util Logging*, a *Log4J2* és a *Logback*. A Spring Bootban ezek az előzőekben említett naplózó keretrendszerek már automatikusan be vannak konfigurálva, ez azért is fontos, mert a különböző third party library-k használják ezeket.

A Spring fejlesztői úgy döntöttek, hogy a Springen belül az Apache **Commons Logging** projektjét fogják használni. Ez később nem bizonyult a legjobb döntésnek, ezért a Spring Bootban már ennek egy javított változata található. Úgy van beállítva, hogy alapértelmezetten a konzolra ír, és csak `INFO` szintű naplóüzeneteket. A naplózás szintje és helye is egyedileg konfigurálható az `application.properties` állományban.

Fejlesztéskor a javasolt eljárás (best practice) a naplózásra az, hogy az SLF4J-t használjuk, amely elrejti a különböző naplózó implementációk közötti különbségeket. Ezt a legegyszerűbben úgy tehetjük meg, ha a Lombokot is használjuk, és ekkor elég csak az osztályunkra rátenni az `@Slf4j` annotációt, amely által a Lombok legenerálja és példányosítja nekünk a következő statikus `Logger` típusú attribútumot:

```
private static final org.slf4j.Logger log =
  org.slf4j.LoggerFactory.getLogger(LogExample.class);
```

Az SLF4J tud paraméterezett üzeneteket is kiírni. Amennyiben ilyet szeretnénk használni, akkor kapcsos zárójeles placeholderrel jelölhetjük a paraméter helyét az üzenetben. Varargs-os, azaz bármennyi paramétert megadhatunk.

```
log.info("Employee has been created");
log.debug("Employee has been created with name {}",
  employee.getName());
```

A konfigurációja történhet az `application.properties`-ben, itt (mint már korábban említésre került) a naplózás szintje és helye állítható be. Amennyiben ettől eltérő dolgokat szeretnénk konfigurálni, akkor lehetőség van arra, hogy az adott naplózó keretrendszer saját konfigurációs állományát tegyük a classpath-ra, és abban adjuk meg ezeket.

```
logging.level.training = debug
```

A Twelve-Factor Appnak a naplózásra is van szabálygyűjteménye:

- Először definiálja a naplóüzenet fogalmát. Eszerint a naplóüzenet nem más, mint információ bizonyos eseményekről, amelyek időben rendezetten, folyamatosan keletkeznek (time ordered event stream).
- Azt mondja, hogy nem az alkalmazásnak a feladata a naplózás helyének megadása, hanem az a legegyszerűbb megoldás, ha a fejlesztők azt állítják be, hogy alkalmazás a konzolra naplózzon, és az üzemeltetők irányítják a naplóüzeneteket a megfelelő helyre (például egy központi szerverre). Ezért nem szükséges fájlba való naplózást beállítani egy Springes alkalmazásban, ráadásul nagyon egyszerűen le tudjuk kérni az alkalmazástól a naplóüzeneteket.

Adatbáziskezelés bevezetés

Ebben a fejezetben megismerheted két olyan eszköz (a JdbcTemplate és a Spring Data JPA) használatát, amelyek a Java nyelven történő adatbáziskezelést jóval egyszerűbbé, gyorsabbá, kevesebb kóddal leírhatóvá teszik. Szót ejtünk két különböző adatbázis használatáról és arról, hogyan vonhatod be őket az alkalmazásod integrációs tesztelésébe. Megnézzük, hogyan futtathatod az alkalmazásodat és az adatbázist egyszerre Docker konténerben, és hogyan végezheted a séma inicializálását Springes alkalmazás esetén Flyway-jel.

Spring JdbcTemplate

Javaban adatbázishoz hozzáférni a **JDBC** technológia segítségével tudunk. A JDBC a Java Standard Edition része, és valójában interfészek gyűjteménye. Minden adatbázisnak saját JDBC Drivere van, amelyben az interfészeket implementáló osztályok találhatóak. A JDBC azonban egy régi, elavult technológia, sok probléma vetődik fel a használata közben. Egyrészt túl "bőbeszédű", azaz túl sokat kell gépelni ahhoz, hogy akár csak a legegyszerűbb adatbázis műveleteket elvégezzük. A JDBC másik hátránya, hogy nagyon elavult a kivételkezelése. Itt is nagyon sok az ismétlődő kódrészletet kell írnunk, hiszen minden esetben kezelni kell a checked `SQLException` kivételt. Az is probléma, hogy ezt az egy fajta kivételt dobja minden esetben, így a kivétel típusa alapján nem lehet rájönni a hiba pontos okára, csak az üzenetből.

A Spring Framework úgy oldja meg ezeket a problémákat, hogy egy template-et hoz létre, amellyel a JDBC fölé egy plusz réteget húz. Ez a **JdbcTemplate**, és ez gyakorlatilag segédmetódusokat tartalmaz a gyakran ismételt adatbázis műveletekre.

Az alkalmazás architektúrájában a perzisztens réteg felelős az adatbázis hozzáférésért és az adatbázis műveletek elvégzéséért. Ezeket az osztályokat a Spring repository-nak nevezi és a `@Repository` annotációval kell őket ellátni.

Ahhoz, hogy a JdbcTemplate-et használni tudjuk, először fel kell venni a `pom.xml`-be függőségként a következőt:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jdbc</artifactId>
</dependency>
```

Ezzel együtt tranzitív függőségként a **H2** beépített, Javaban implementált, memóriában futó adatbázis kezelő rendszer is használhatóvá válik. Developer Tools használata esetén ennek a webes konzolja is elérhető böngészőből a `/h2-console` címen, ahol meg lehet nézni a táblákat, a bennük lévő adatokat, valamint SQL-kifejezéseket is futtathatunk.

Ha az előbbi függőséget felvettük, akkor az alkalmazás indításakor ez az adatbázis is alapértelmezetten elindul. Adatbázis műveletek végzéséhez a repository osztályainkba be kell injektálni egy JdbcTemplate-et.

A legegyszerűbb műveletek (új adat beszúrása, adat módosítása, törlés) az `update()` metódus segítségével végezhetőek el. Megtehetjük mindezeket paraméterezetten is, placeholderek (kérdőjelek) használatával. Soha ne konkatenáljuk a paramétereket az SQL-utasításokhoz, mert így az alkalmazásunkat egy lehetséges SQL injection támadás veszélyének tesszük ki.

```
jdbcTemplate.update(
  "insert into employees(emp_name) values ('John Doe')");

jdbcTemplate.update(
  "insert into employees(emp_name) values (?)", "John Doe");

jdbcTemplate.update(
  "update employees set emp_name = ? where id = ?", "John Doe", 1);

jdbcTemplate.update(
  "delete from employees where id = ?", 1);
```

Abban az esetben, ha az adatbázis generálja számunkra a beszúrt adat egyedi azonosítóját, lehetőség van arra is, hogy ezt lekérdezzük. Ebben az esetben először egy `GeneratedKeyHolder`-t kell példányosítani. Majd a JdbcTemplate `update()` metódusát kell meghívni úgy, hogy egy `PreparedStatementCreator`-t adunk át neki paraméterül, implementálva annak egyetlen, `createPreparedStatement()` nevű metódusát. Itt arra kell vigyázni, hogy olyan `PreparedStatement`-et hozzunk létre, amely alkalmas arra, hogy visszaadja az adatbázis által generált egyedi azonosítót. Tehát két paramétert kell neki átadni, az első a placeholdereket tartalmazó SQL-utasítás, a második pedig egy konstans, a `Statement.RETURN_GENERATED_KEYS`. Az SQL-utasítás placeholderei helyére a konkrét paraméterek behelyettesítését nekünk kell megírnunk. Az `update()` metódus második paramétere a korábban példányosított `KeyHolder`, amely majd a generált azonosítót fogja tartalmazni. Ezt lekérni a `getKey().longValue()` láncolt metódusokkal tudjuk. A videóban látható példában az előbbiekben leírtak lambda kifejezéssel, egyszerűbben vannak leírva:

```
KeyHolder keyHolder = new GeneratedKeyHolder();

jdbcTemplate.update(
  con -> {
      PreparedStatement ps =
        con.prepareStatement("insert into employees(emp_name) values (?)",
          Statement.RETURN_GENERATED_KEYS);
      ps.setString(1, employee.getName());
      return ps;
  }, keyHolder);

employee.setId(keyHolder.getKey().longValue());
```

Az adatbázisból a JdbcTemplate `query()` és `queryForObject()` metódusaival lehet lekérdezni, amelyeknek első paraméterként át kell adnunk egy SQL-lekérdezést. Amennyiben nem valamilyen primitív értéket szeretnénk megkapni, hanem objektumokat, akkor második paraméterként pedig egy olyan metódust kell átadnunk, amely a visszakapott `ResultSet` egy konkrét sorát objektummá képezi le. A `query()` metódus objektumok listáját adja vissza, a `queryForObject()` pedig egy konkrét objektumot. Utóbbinak harmadik, varargs-os paraméterként meg kell adnunk az SQL-lekérdezésbe behelyettesítendő paramétereket is.

```
List<Employee> employees = jdbcTemplate.query(
  "select id, emp_name from employees",
  this::convertEmployee);

Employee employee =
  jdbcTemplate.queryForObject(
    "select id, emp_name from employees where id = ?",
    this::convertEmployee,
    id);

private Employee convertEmployee(ResultSet resultSet, int i)
    throws SQLException {
  long id = resultSet.getLong("id");
  String name = resultSet.getString("emp_name");
  Employee employee = new Employee(id, name);
  return employee;
}
```

Ahhoz, hogy mindez működjön, természetesen az adatbázisban benne kell lennie a megfelelő táblának és abban esetleg az adatoknak. Ezt többféle módon is meg lehet oldani, jelen esetben egy `@Component` annotációval ellátott osztályban egy `CommandLineRunner` interfészt implementálunk, amelynek van egy `run()` metódusa. A Spring az alkalmazás indulásakor lefuttatja ezt a metódust. Ebbe az osztályba injektáljuk a JdbcTemplate-et és meghívjuk az `execute()` metódusát, amivel különböző, az adattáblát létrehozó és néhány új rekordot beszúró SQL-utasításokat futtatunk le.

```
@Component
public class DbInitializer implements CommandLineRunner {
```

```
  @Autowired
  private JdbcTemplate jdbcTemplate;

  @Override
  public void run(String... args) throws Exception {
    jdbcTemplate.execute("create table employees " +
      "(id bigint auto_increment, emp_name varchar(255), " +
      "primary key (id))");

    jdbcTemplate.execute(
      "insert into employees(emp_name) values ('John Doe')");
    jdbcTemplate.execute(
      "insert into employees(emp_name) values ('Jack Doe')");
  }
}
```

Spring Data JPA

A **JPA** egy **ORM – Object Relational Mapping** eszköz, amely automatikusan megfelelteti a Java objektumokat az adatbázis rekordjainak. Ez egy API, amelynek különböző implementációi vannak. A legelterjedtebb ezek közül a *Hibernate* és az *EclipseLink*. A Spring Data JPA egy third party library, amely Spring Boot esetén még könnyebbé teszi a JPA használatát. Olyan repository osztályokat tudunk vele létrehozni, melyek a leggyakoribb műveleteket (CRUD, valamint rendezés és lapozás) automatikusan elvégzik.

Úgy használható, hogy mindössze egy interfészt kell implementálnunk, amelynek generikus paraméternek meg kell adnunk az entitást és az entitás azonosítójának a típusát. Ezáltal az interfészhez tartozó implementációt a Spring Data JPA automatikusan elkészíti. Ezenkívül egy *query by example* funkcióval is rendelkezik, amely azt jelenti, hogy nem nekünk kell paramétereznünk a lekérdezést, hanem átadunk egy olyan objektumot, amelynek ki vannak töltve bizonyos mezői, és az olyan objektumokat fogja lekérdezni, amelyeknek a mezői hasonlóan vannak kitöltve. A Spring Data JPA segítségével tehát nagyon nagy mértékben lerövidíthető az alkalmazásunk kódja, mert sok kódrészletet automatikusan generál helyettünk.

Ahhoz, hogy használni tudjuk, a következő lépéseket kell tennünk:

- Fel kell vennünk függőségként a következőt, amely tranzitívan hozza magával a JPA API-t és a Hibernate-et:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

- Ezután a JPA annotációk segítségével létre kell hoznunk egy entitást.
- Majd implementálnunk kell a `JpaRepository` interfészt (vagy saját interfész esetén ebből leszármaztatnunk).

- Létrehozhatunk további olyan metódusokat is, amelyekkel testre szabjuk a magunk számára a JPA által legenerált SQL-utasításokat.
- Ha több metódust egy tranzakcióban szeretnénk lefuttatni, akkor a service rétegben kell a kívánt publikus metódusokra rátenni a `@Transactional` annotációt.
- A JPA-t és a Spring Data JPA-t konfigurálni az `application.properties` állományban lehet. Sok előre legenerált property van, a leggyakrabban használt a `spring.jpa.show-sql=true`. Ha ezt megadjuk, akkor a konzolra íródnak a generált SQL utasítások.

A `JpaRepository` interfész által tartalmazott metódusok a következők:

- `save()`, `saveAll()`: entitás mentésére.
- `saveAndFlush()`: entitás mentése úgy, hogy meghívódik a `flush()` metódus.
- `findById()`: entitás lekérdezése azonosító alapján. Ez `Optional` példánnyal tér vissza, így kezelve azt az esetet, ha nem találják meg az adatbázisban a keresett entitást.
- `findAll()`: összes entitás lekérdezése. Ennek a metódusnak van olyan paraméterezésű változata is, amelyben meg lehet adni a lapozást, valamint olyan is, amely egy `example`-t, egy konkrét entitást vár paraméterül.
- `findAllById()`: egy listát vár, amely azonosítókat tartalmaz, és lekérdezi az összes azonosítóhoz tartozó entitást.
- `getOne()`: ez is azonosító alapján kérdez le, ez nem `Optional` példánnyal tér vissza.
- `exists()` és `existsById()`: meg lehet nézni, hogy létezik-e ilyen entitás.
- `count()`: meg lehet számolni az entitásokat.
- `delete()`, `deleteById()`, `deleteAll()` és további `delete`-kezdetű metódusok, többféle paraméterezéssel: entitás(ok) törlésére.
- `flush()`: ugyanúgy működik, mint a JPA ugyanilyen nevű metódusa.

Egy entitást ugyanúgy kell létrehozni, mint egy sima JPA-s entitást. Vagyis minimálisan a következőket kell megtenni: kötelező az osztályra `@Entity` annotációt tenni, valamint kötelező megadni egy egyedi azonosítót tartalmazó mezőt, melyet az `@Id` annotációval kell ellátni.

```
@Data
@NoArgsConstructor
@AllArgsConstructor
@Entity
@Table(name = "employees")
public class Employee {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long id;

    @Column(name = "emp_name")
    private String name;

    public Employee(String name) {
        this.name = name;
```

```
    }
}
```

A repository ki kell, hogy terjessze vagy implementálja a `JpaRepository` interfészt, melynek generikus típusnak meg kell adni az entitás és annak egyedi azonosítójának típusát. Ezáltal a metódusokhoz tartozó lekérdezéseket a JPA már automatikusan generálja számunkra. Ha ezek nem elegendőek nekünk, akkor más paraméter alapján is tudunk keresni, például név alapján, amint az alábbi kódrészletben látható:

```
public interface EmployeesRepository extends JpaRepository<Employee, Long> {

    @Query("select e from Employee e where upper(e.name) like upper(:name)")
    List<Employee> findAllByPrefix(String name);

}
```

Látható, hogy itt a `@Query` annotációt van elhelyezve a metóduson. Az annotációnak paraméterül meg lehet adni az általunk kívánt lekérdezést, amelyben a metódus paraméterére lehet hivatkozni (`:name`).

MariaDB

Ahhoz, hogy az alkalmazásunkban MariaDB adatbázishoz tudjunk kapcsolódni, először el kell azt indítani. Docker konténerben a következő módon tehetjük meg ezt:

```
docker run
  -d
  -e MYSQL_DATABASE=employees
  -e MYSQL_USER=employees
  -e MYSQL_PASSWORD=employees
  -e MYSQL_ALLOW_EMPTY_PASSWORD=yes
  -p 3306:3306
  --name employees-mariadb
  mariadb
```

Ezzel a már létező `mariadb` image-ből indulunk ki, ami a DockerHubon megtalálható. Ennek megadhatóak környezeti változóként az elinduláshoz szükséges paraméterek (adatbázis létrehozása és az ehhez való kapcsolódási paraméterek).

Ekkor létrehoz egy `employee` nevezetű adatbázist, valamint egy felhasználót (felhasználónév/jelszó: `employees/employees`). A root felhasználó jelszava üres. A `3306`-os portját kivezetjük a host gép `3306`-os portjára. A konténer neve `employees-mariadb`, a későbbiekben így hivatkozhatunk rá a Docker parancsokban.

Ezután az alkalmazásunkat kell úgy módosítani, hogy a már futó adatbázishoz hozzá tudjon kapcsolódni. A `pom.xml` állományba fel kell venni függőségként a JDBC drivert.

```
<dependency>
  <groupId>org.mariadb.jdbc</groupId>
```

```
  <artifactId>mariadb-java-client</artifactId>
  <scope>runtime</scope>
</dependency>
```

Nem kell megadni verziószámot, mert a Spring Boot azt definiálja. A `scope` `runtime`, mivel a driver osztályaira direktben nem akarunk hivatkozni.

az `application.properties` állományban pedig előre definiált property-k segítségével meg lehet adni az adatbázis JDBC URL-jét, a felhasználónevet és a jelszót. JPA használata esetén megadhatjuk a `ddl-auto` nevű kapcsolót, amely megmondja a JPA implementációnak, hogy mit csináljon az adatbázis sémával. A `create-drop` azt jelenti, hogy az alkalmazás minden egyes indításakor dobja el és hozza létre újra az adatbázis tábláit, majd újra dobja el. (Ez természetesen az alkalmazás tesztelésekor használható jól, éles alkalmazásban azonban nem megfelelő.)

```
spring.datasource.url=jdbc:mariadb://localhost/employees
spring.datasource.username=employees
spring.datasource.password=employees

spring.jpa.hibernate.ddl-auto=create-drop
```

A Twelve-Factor App ajánlásai a backing service-ekkel kapcsolatban a következők (backing service-nek nevezzük az összes külső erőforrást, amelyet az alkalmazásból el szeretnénk érni, például adatbázis, üzenetküldő middleware-ek, directory- és e-mail szerverek, elosztott cache-ek, Big Data eszközök, de microservice környezetben ide számítható egy másik alkalmazás is, stb.):

- Ezek telepítése történjen automatikusan, lehetőleg egy olyan eszköz segítségével, amelynek meg lehet adni az infrastruktúrát kód formájában (*Infrastructure as Code*). Ez annyit jelent, hogy valamilyen szöveges állományban megadhassuk, hogy mire van szükségünk, és ez alapján maga az eszköz hozza létre és telepítse föl a backing service-t. Ilyen eszköz például az *Ansible*, a *Chef* vagy a *Puppet*.
- A backing service-ekhez az alkalmazás hozzá tudjon férni. Az adatbázis elérését lehetőleg környezeti változóként lehessen megadni, mert így a legegyszerűbb paramétert bejuttatni a konténerbe.
- A fájlrendszer nem tekinthető megfelelő háttérszolgáltatásnak, mert olyan fontos tulajdonságok hiányoznak belőle, mint a tranzakcionalitás vagy a magas rendelkezésre állás. Fájlok tárolására is valamilyen erre alkalmas backing service-t érdemes választani.
- Léteznek az alkalmazásba beágyazható háttérszolgáltatások, ezek főleg tesztelésre használhatóak jól. Éles alkalmazásban érdemes külön konténerekben futó külön alkalmazásokat használni, mert ekkor azok külön kezelhetőek az alkalmazásunktól.
- Amennyiben leáll egy backing service és újra kell indítani, akkor ne kelljen az alkalmazást is újraindítani, hanem automatikusan újra tudjon kapcsolódni.
- Microservice architektúránál elég sok komponens létezik még, amit érdemes használni. Az egyik ilyen például a *circuit breaker* eszköz. Ez észreveszi, ha leáll egy backing service, és ekkor megszünteti a hozzáférést, és csak egy idő múlva engedi csak újra a

hozzá intézett kéréseket. Ez azért fontos, hogy ne legyen terhelve addig, amíg újra tud indulni.

MariaDB gyakorlat

MariaDB

A `h2` adatbázis függőséget tegyük megjegyzésbe!

IDEA-ban a `pom.xml` után Maven frissítés szükséges.

Ha elindítjuk az alkalmazást, akkor létrehozza a táblákat az adatbázisban, majd a REST webszolgáltatások meghívásakor adatokat lehet beszúrni, módosítani, törölni és lekérdezni. (Használjuk az IDEA http fájlját!)

Ellenőrizni lehet az adatbázis tartalmát, ha az IDEA-val hozzácsatlakozunk. Ehhez jobb oldalon a Database fülön fel kell venni egy MariaDB DataSource-ot, és adjuk meg a felhasználónevet, jelszót és az adatbázist (mindhárom értéke `employees`)! A *Test Connection* gombra kattintva tudjuk ellenőrizni, hogy tudunk-e az adatbázishoz kapcsolódni. Ha sikeres, akkor utána megnyílik egy konzol, melyen SQL parancsokat tudunk kiadni. Legyen ez:

```
select * from employees;
```

PostgreSQL

Amennyiben szeretnénk, használhatunk PostgreSQL-t is. Ehhez a következő parancsot kell kiadnunk:

```
docker run
  -d
  -e POSTGRES_PASSWORD=password
  -p 5432:5432
  --name employees-postgres
  postgres
```

Az alapértelmezett felhasználónév `postgres`, és az adatbázis neve is ez. A jelszó a környezeti változónak megfelelően `password`. A host `5432` portján lehet hozzá kapcsolódni. A konténer neve `employees-postgres` lesz.

Akkor a `pom.xml`-ben a következő legyen a függőség:

```
<dependency>
  <groupId>org.postgresql</groupId>
  <artifactId>postgresql</artifactId>
  <scope>runtime</scope>
</dependency>
```

És az `application.properties` fájlba a következőt kell beírni:

```
spring.datasource.url=jdbc:postgresql:postgres

spring.datasource.username=postgres
spring.datasource.password=password
spring.jpa.hibernate.ddl-auto=create-drop
```

Integrációs tesztelés

Amennyiben a tesztelésben az adatbázis réteget is be szeretnénk vonni, akkor már nem unit tesztről, hanem integrációs tesztről beszélünk.

A Spring Boot nagyon sok lehetőséget biztosít arra, hogy az alkalmazásunkat akár rétegenként, akár egyben tudjuk tesztelni. Ha csak a perzisztens réteget akarjuk tesztelni, akkor használhatjuk a tesztünkön a `@DataJpaTest` annotációt, és így csak az alkalmazásnak a repository rétegét indítja el. Ebben az esetben egy beágyazott H2 adatbázist indít el, és ekkor injektálhatjuk a tesztbe a saját repository-nkat, a `DataSource`-ot, a `JdbcTemplate`-et és az `EntityManager`-t is. Service vagy controller osztályokat viszont nem injektálhatunk, hiszen azok ilyenkor nem kerülnek elindításra. A repository réteg tesztelése arra szolgál, hogy jól helyeztük-e el az entitásainkon az annotációkat, illetve jól írtuk-e meg azokat a metódusneveket, amelyeknél azt szeretnénk, hogy a Spring Data JPA generálja a metódus törzsét, illetve hogy jól írtuk-e meg a JPQL lekérdezéseinket.

Egy ilyen tesztelés esetén minden tesztmetódus saját tranzakciót indít, és a végén rollback utasítást ad ki, ezáltal nem marad bent adat a tesztadatbázisban.

```
@DataJpaTest
public class EmployeesRepositoryIT {

  @Autowired
  EmployeesRepository employeesRepository;

  @Test
  void testPersist() {
    Employee employee = new Employee("John Doe");
    employeesRepository.save(employee);
    List<Employee> employees =
      employeesRepository.findAll();

    assertThat(employees)
      .extracting(Employee::getName)
      .containsExactly("John Doe");
  }
}
```

Ha a teljes alkalmazást szeretnénk tesztelni, akkor a tesztosztályra a `@SpringBootTest` annotációt kell tenni, ez minden réteget beindítja. Ilyenkor már érdemes valódi adatbázissal dolgozni, amit éles alkalmazásnál is használni fogunk. Ekkor foglalkozni kell azzal is, hogy a teszteset az adatbázis sémát is létrehozza és feltöltse adatokkal.

Ha a tesztesetek futása során az adatbázis elérési paramétereit felül szeretnénk írni, akkor ezt a Maven projektben az `src/test/resources/application.properties` fájlban tehetjük meg.

Döntenünk kell arról, hogy a séma inicializációt mi végzi. Elvégezheti ezt a JPA implementáció is, amely H2 esetén `create-drop` tulajdonsággal működik, azaz minden teszteset lefutása után eldobja és újra létrehozza az adatbázis tábláit. A JPA, ha talál a classpath-on egy séma SQL-állományt, akkor azt lefuttatja, és így tölthető fel az adatbázis kezdeti adatokkal. Komolyabb alkalmazás esetén érdemesebb azonban valamilyen séma inicializációra használatos eszközt alkalmazni például a Flyway-t vagy a Liquibase-t.

Ha lefuttatunk egy tesztesetet, szükségünk lehet arra, hogy legyenek benne valamilyen kezdeti adatok, amin a teszteset dolgozni tud. Ezt többféle módon is létrehozhatjuk:

- A legegyszerűbb, ha elhelyezünk a classpath-on egy `data.sql` állományt, és ezt a JPA automatikusan lefuttatja.
- SQL-utasításokat lefuttathatunk úgy is, hogy a tesztosztályon elhelyezünk egy `@Sql` annotációt. Ebben az esetben minden egyes tesztmetódus előtt le fog futni az itt megadott SQL-utasítás. (Ennek átadhatunk egy-vagy több script elérési útvonalát is.)
- `@BeforeEach` vagy `@AfterEach` annotációkkal megjelölt metódusokban programozottan, nagyon sokféle módon juttathatunk be adatokat az adatbázisba. Használhatjuk például a publikus API-t, hívhatunk injektált service vagy a controller példányokat vagy például egy `JdbcTemplate` segítségével közvetlenül is hozzáférhetünk az adatbázishoz.

Az adatbázist is használó tesztesetek futtatása közben fontos arra figyelni, hogy két tesztmetódus futása között van állapotátmenet, tehát ezáltal a különböző tesztesetek nem függetlenek egymástól. Ennek kivédésére sok módszer is van. Dönteni kell, hogy a teszteset maga előtt vagy maga után tegyen rendet (ez utóbbit el szokás felejteni, tehát érdemesebb az előzőt használni). Különböző lehetőségek vannak arra is, hogy pontosan hogyan is történjen ez: a teljes séma letörlődjön és újra létrejöjjön (ez tiszta, de lassú megoldás), lefuttathatunk egy teljes adatbázis importot minden teszteset előtt, vagy a konkrét teszteset lefutása előtt csak azokat a táblákat ürítjük ki, amelyeket az adott teszteset használni fog (ez a leggyorsabb megoldás, de ez sok pluszmunkával jár a tesztesetek megírásakor).

Adatbáziskezelés - Integrációs tesztelés - gyakorlat - H2

Vegyük fel a H2 függőséget `test` scope-pal!

```
<dependency>
  <groupId>com.h2database</groupId>
  <artifactId>h2</artifactId>
  <scope>test</scope>
</dependency>
```

Adatbáziskezelés - Integrációs tesztelés - gyakorlat - MariaDB

Indítsunk el egy külön adatbázis példányt, amit csak a teszt esetek fognak használni.

```
docker run
  -d
  -e MYSQL_DATABASE=employees
  -e MYSQL_USER=employees
  -e MYSQL_PASSWORD=employees
  -e MYSQL_ALLOW_EMPTY_PASSWORD=yes
  -p 3307:3306
  --name employees-test-mariadb
  mariadb
```

A `src/main/resources` könyvtárban lévő `application.properties` állományt felülírhatjuk a `src/test/resources` könyvtárban lévő `application.properties` állománnyal.

Hozzunk létre az `src/test/resources` könyvtárba egy `application.properties` fájlt, és másoljuk bele a `src/main/resources` könyvtárban lévő `application.properties` állomány tartalmát, és írjuk felül a következő property-ket:

```
spring.datasource.url=jdbc:mariadb://localhost:3307/employees
spring.jpa.hibernate.ddl-auto=create
```

Futtassuk az `EmployeesControllerRestTemplateIT` tesztosztályt!

A `deleteAll()` hívást kiválthatjuk, hogy a tábla tartalmát a tesztesetből töröljük:

```
@Sql(statements = "delete from employees")
```

Alkalmazás futtatása Dockerben MariaDB-vel

> A `--link` kapcsoló már deprecated a Dockerben, hanem helyette egy hálózatot kell létrehozni, és ahhoz kapcsolni a konténereket, ahogy a gyakorlati videóban is szerepel.

Fejlesztéskor érdemes az adatbázist Docker konténerben futtatni, hiszen ilyenkor különböző adatbázisokat is futtathatunk párhuzamosan, magát az alkalmazást viszont az egyszerűség kedvéért érdemes a saját gépünkön futtatni. Ekkor a saját gépen, a fejlesztőeszközben futó alkalmazás a Docker konténerben futó adatbázishoz tud kapcsolódni. Amikor azonban teszt- vagy éppen éles környezetre telepítjük az alkalmazást, akkor az adatbázis és az alkalmazás is Docker környezetben fog futni. Ilyenkor a két konténernek egymáshoz kell tudni kapcsolódni.

Ezt régebbi Docker verziókban *link*-kel lehetett megtenni. Valamint érdemes mindkettőnek a portját kivezetni a saját gépünkre (az alkalmazás konténerét például a 8080-asra, a MariaDB adatbázis portját pedig a 3306-osra).

```
docker run
    -d
```

```
    -e SPRING_DATASOURCE_URL=jdbc:mariadb://employees-mariadb/employees
    -e SPRING_DATASOURCE_USERNAME=employees
    -e SPRING_DATASOURCE_PASSWORD=employees
    -p 8080:8080
    --link employees-mysql:employees-mysql
    --name employees
    employees
```

A Dockerben való futtatáshoz tudni kell, hogy a parancsban meg lehet adni különböző értékeket környezeti változóként. Az adott konténerben ezek a környezeti változóként megadott paraméterek (`-e` kapcsolóval) felülírják az `application.properties`-ben megadott értékeket. Az `application.properties`-ben ezeket konvenció szerint kisbetűvel és ponttal elválasztva szokás megadni, a környezeti változókat viszont nagybetűvel és aláhúzás jellel elválasztva. A kettőt a Spring Boot automatikusan tudja egymásba konvertálni.

Alkalmazás futtatása Dockerben MariaDB-vel

Hozzunk létre egy hálózatot:

```
docker network create --driver bridge employees-net
```

Indítsunk el egy (immáron sokadik) adatbázist, mely csak a Dockerben futó alkalmazást szolgálja ki.

```
docker run
  -d
  -e MYSQL_DATABASE=employees
  -e MYSQL_USER=employees
  -e MYSQL_PASSWORD=employees
  -e MYSQL_ALLOW_EMPTY_PASSWORD=yes
  -p 3308:3306
  --network employees-net
  --name employees-net-mariadb
  mariadb
```

Majd indítsuk el az alkalmazást, Docker konténerben, hogy ehhez az alkalmazáshoz kapcsolódjon. Ne feledjük, hogy először állítsuk le az előző konténert, töröljük le, buildeljük le az alkalmazást, majd utána a Docker image-et is.

Végül:

```
docker run
    -d
    -e SPRING_DATASOURCE_URL=jdbc:mariadb://employees-net-mariadb/employees
    -p 8080:8080
    --network employees-net
    --name employees
    employees
```

Majd állítsd le a két konténert!

Alkalmazás futtatása Docker Compose-zal - gyakorlat

A két konténert el lehet indítani egyszerre a Docker Compose használatával is.

Szükség van a `wait-for-it.sh` szkriptre az alkalmazás konténerén belül. Ehhez ki kell egészítenünk a `Dockerfile` tartalmát:

```
RUN apt update \
     && apt-get install wget \
     && apt-get install -y netcat \
     && wget https://raw.githubusercontent.com/vishnubob/wait-for-
it/master/wait-for-it.sh \
     && chmod +x ./wait-for-it.sh
```

A projekten belül hozzunk létre egy `employees` könyvtárat, és helyezzünk el benne egy `docker-compose.yml` fájlt a következő tartalommal:

```
version: '3'

services:
  employees-mariadb:
    image: mariadb
    restart: always
    ports:
      - '3308:3306'
    environment:
      MYSQL_DATABASE: employees
      MYSQL_USER: employees
      MYSQL_PASSWORD: employees
      MYSQL_ALLOW_EMPTY_PASSWORD: yes

  employees-app:
    image: employees
    restart: always
    ports:
      - '8081:8080'
    depends_on:
      - employees-mariadb
    environment:
      SPRING_DATASOURCE_URL: 'jdbc:mariadb://employees-mariadb/employees'
    entrypoint: ['./wait-for-it.sh', '-t', '120', 'employees-mariadb:3306',
'--', 'java', 'org.springframework.boot.loader.JarLauncher']
```

Séma inicializálás Flyway-jel

Amennyiben adatbázist használunk, gondoskodni kell arról, hogy létrehozzuk az adatbázis sémát. Erre többféle eszköz is létezik. Érdemes figyelni arra, hogy ezek létrehozása verziókövetetten történjen. Ezek a rendszerek úgy oldják ezt meg, hogy egyrészt valahol le vannak tárolva azok a scriptek, amelyek létrehozzák magát a sémát, másrészt pedig le van

tárolva magában az adatbázisban, például egy metaadat táblában, hogy milyen verzióban van maga az adatbázis, milyen scriptek futottak le.

Nézzük meg, milyen elvárások vannak egy ilyen eszközzel szemben, milyen szempontok alapján érdemes választani:

- Meg lehessen adni a különböző verziókat SQL és XML formátumban is. Ez utóbbiból maga az eszköz tudjon SQL-utasításokat generálni.
- Az előbbiből következik, hogy egy ilyen eszköz platformfüggetlen legyen.
- Legyen pehelysúlyú (lightweight), egyszerű legyen a futtatása.
- Jó, ha vissza lehet állni egy korábbi verzióra.
- Lehessen indítani a séma létrehozást parancssorból vagy az alkalmazásból is.
- Támogassa a clusteres működést.
- Támogassa a placeholderek használatát, vagyis lehessen benne változókat használni.
- Jó, ha támogatja a modularizációt, vagyis ha az alkalmazásunk több különböző modulból áll, akkor a sémákat is lehessen az egyes modulokhoz külön definiálni, modulonként saját táblákat létrehozni.
- Több sémában is lehessen adatbázis objektumokat létrehozni.

Javaban két elterjedt megoldás van, a *Flyway* és a *Liquibase*. Ezek működése nagyon hasonlít egymáshoz. Az a különbség közöttük, hogy a Liquibase-nek XML-ben is meg lehet adni az adatbázis felépítését, és ő ez alapján az adott adatbázishoz megfelelő SQL-utasításokat fogja legenerálni, Flyway használata esetén pedig nekünk kell az SQL-utasításokat megírni.

Ahhoz, hogy a Flyway-t használni tudjuk, fel kell venni függőségként a `pom.xml`-be.

```
<dependency>
  <groupId>org.flywaydb</groupId>
  <artifactId>flyway-core</artifactId>
</dependency>
```

A Flyway-nek van integrációja a Spring Boottal, azaz ha a Spring Boot észreveszi, hogy a Flyway a classpath-on szerepel, akkor automatikusan felolvassa az `src/main/resources/db/migration` könyvtárat, és lefuttatja az abban szereplő scripteket. Ehhez semmilyen további konfiguráció nem szükséges, csak annyi, hogy ebben a könyvtárban legyenek ezek elhelyezve. Az elnevezési konvenció, amit be kell tartanunk, az, hogy a következőhöz hasonló módon adjuk meg az SQL-utasításokat tartalmazó fájlok neveit:

```
V1__employees.sql
```

Ebben a `V1` egy verziószámot jelöl, utána következik 2 db aláhúzás jel (_), majd egy tetszőleges név, a kiterjesztés pedig `.sql`.

A Flyway úgy működik, hogy felolvassa az összes, ilyen módon megadott SQL állományt, sorba rendezi a verziószámok alapján, és megpróbálja ezeket lefuttatni. A lefuttatás tényét a `flyway-schema-history` táblában rögzíti. Ha következőleg elindítjuk az alkalmazást, akkor először felolvassa ennek a táblának a tartalmát, és abban már látja, hogy mely scriptek

kerültek lefuttatásra. Ezáltal észreveszi azt is, ha olyan scriptet talált, amely még nem lett lefuttatva, és ekkor azt is lefuttatja. Abban az esetben, ha egy script lefuttatása nem sikerül, akkor az alkalmazás el sem indul. Ekkor a hibát is bejegyzi ebbe a táblába. Ha helyre akarjuk állítani az alkalmazást, akkor nekünk kell innen a megfelelő sort kitörölni és kijavítani az SQL scriptet.

Séma inicializálás Flyway-jel - gyakorlat

Mind a két `application.properties` fájlba fel kell venni, hogy ne a JPA provider hozza létre a táblákat:

```
spring.jpa.hibernate.ddl-auto=none
```

Létre kell hozni a `src/main/resources/db/migration/V1__employees.sql` fájlt. MariaDB esetén a tartalma:

```
create table employees (id bigint auto_increment,
  emp_name varchar(255), primary key (id));
insert into employees (emp_name) values ('John Doe');
insert into employees (emp_name) values ('Jack Doe');
```

Postgres esetén a tartalma:

```
create table employees (id int8 generated by default as identity,
  emp_name varchar(255), primary key (id));
insert into employees (emp_name) values ('John Doe');
insert into employees (emp_name) values ('Jack Doe');
```

Liquibase

Flyway helyett használható Liquibase is. Ehhez a `pom.xml`-be a következőt kell felvenni:

```
<dependency>
  <groupId>org.liquibase</groupId>
  <artifactId>liquibase-core</artifactId>
</dependency>
```

Az `application.properties` állományban:

```
spring.liquibase.change-log=classpath:db/changelog/db.changelog-master.xml
```

A `src/main/resources/db.changelog-master.xml` fájl tartalma legyen:

```
<databaseChangeLog
    xmlns="http://www.liquibase.org/xml/ns/dbchangelog"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:ext="http://www.liquibase.org/xml/ns/dbchangelog-ext"
    xsi:schemaLocation="http://www.liquibase.org/xml/ns/dbchangelog
      http://www.liquibase.org/xml/ns/dbchangelog/dbchangelog-3.1.xsd
      http://www.liquibase.org/xml/ns/dbchangelog-ext
      http://www.liquibase.org/xml/ns/dbchangelog/dbchangelog-ext.xsd">
```

```
  <changeSet id="create-employee-table" author="vicziani">
    <sqlFile path="create-employee-table.sql"
      relativeToChangelogFile="true" />
  </changeSet>
</databaseChangeLog>
```

Az `src/main/resources/create-employee-table.sql` fájl tartalma MariaDB esetén legyen:

```
create table employees (id bigint not null auto_increment, emp_name
varchar(255), primary key (id));
```

PostgreSQL esetén:

```
create table employees (id int8 generated by default as identity, emp_name
varchar(255), primary key (id));
```

MongoDB

Indítsuk el a MongoDB-t Docker konténerben:

```
docker run -d -p27017:27017 --name employees-mongo mongo
```

Másoljuk le az alkalmazás projekt könyvtárát `employees-mongo` néven! A `pom.xml`-ben az `artifactId`-t írjuk át erre!

A `pom.xml`-ből a következő függőségeket távolítsuk el: `spring-boot-starter-jdbc`, `spring-boot-starter-data-jpa`, `h2`, `mariadb-java-client`, `postgresql`, `flyway-core`, `liquibase-core`.

Vegyük fel függőségként:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-mongodb</artifactId>
</dependency>
```

Az `application.properties` fájlban:

```
spring.data.mongodb.database = employees
```

Módosítsuk az entitást:

```
@Data
@NoArgsConstructor
@AllArgsConstructor
@Document("employees")
public class Employee {

    @Id
    private String id;
```

```
    private String name;

    public Employee(String name) {
        this.name = name;
    }
}
```

Módosítsuk a repository-t:

```
public interface EmployeesRepository extends MongoRepository<Employee,
String> {

}
```

A Dto-kban, Controllerben, Service metódus paraméterekben mindenütt, ahol `Long` típusú azonosítót adtunk tovább, ki kell cserélni `String` típusra.

Az `EmployeesService.updateEmployee()` metódusát is módosítani kell, hogy legyen mentés:

```
public EmployeeDto updateEmployee(long id, UpdateEmployeeCommand command) {
    Employee employee = repository.findById(id)
      .orElseThrow(() -> new IllegalArgumentException("employee not found"));
    employee.setName(command.getName());
    employeesRepository.save(employee);
    return modelMapper.map(employee, EmployeeDto.class);
}
```

Töröljük le a teszteseteket!

Utána futtassuk az alkalmazást, és teszteljük a REST webszolgáltatásokat!

Az IntelliJ IDEA-val hozzá lehet kapcsolódni MongoDB adatbázishoz, jobb oldalt a *Database* fülön. Vegyünk fel egy MongoDB DataSource-t, és a *Database* értéke legyen `employees`. Ekkor már látjuk is a collectiont! Duplán kattintva meg is jelenik annak tartalma.

Konzolon is tudunk műveleteket kiadni. Lekérdezés:

```
db.employees.find()

db.employees.insertOne({"name": "John Doe"})

db.employees.findOne({'_id': ObjectId('60780cf974bc5648cf220a96')})

db.employees.deleteOne({'_id': ObjectId('60780cf974bc5648cf220a96')})
```