L4: Spring Boot, Spring Web

- A jelen laborfeladatban egy, **az eddigiektől független** alprojektet vezetünk be a multimodul projektünkbe, amely **Spring** segítségével oldja meg a korábban vázolt problémákat. Bármilyen releváns kódrészlet újrahasználható a korábbi saját feladatokból, **de csak a szükséges kódrészleteket vegyük át**. A feladat a következőket érinti:
  - a *dependency injection* tervezési minta
  - automatikus konfiguráció-olvasás
  - REST API kialakítása *Spring Web* segítségével
- A laborfeladatok továbbra is használják az első feladatban előírt projekttematikát (ld. `ubb-idde-lab1-build.pdf`).
- Továbbra is feltétel, hogy ne legyenek statikus kódelemző jelzések, illetve hogy helyesen fusson le a `check` taszk a folyamatos integrációban (ld. korábbi laborfeladatok leírása).

Dependency Injection

Dependency Injection

- Készítsünk egy új alprojektet, melynek neve lehet pl. *abcd1234-spring* (ahol az *abcd1234* a diák-ID-nk helytartója). Nem szükséges függősége legyen bármely eddigi alprojekthez.
- A Spring keretrendszer segítségével (ld. Spring Boot Starter függőség **és** Gradle plugin) vezessük be a dependency injection tervezési mintát.
- A későbbiekben bevezetett adatelérési objektumok és webkontrollerek használják kizárólag a dependency injection mintát.
- Ha két bean között függőségünk van, ne használjunk kézzel implementált factory mintát, ehelyett vagy használjuk a stereotype annotációkat (`@Component`, stb) a függőségen, vagy készítsünk Spring factory-t neki a `@Configuration` és `@Bean` annotációkkal.
- Ahol lehet, alkalmazzuk a réteg-specifikus stereotype annotációkat, hogy jelöljük egy beanről, hogy melyik réteghez tartozik a többrétegű architektúrában.
- Konfiguráljuk az `application.properties` vagy `application.yml`-ben az összes változó konfigurálható értéket, majd injektáljuk Springnek megfelelően.
- Használjuk a Spring profilokat, hogy különbséget tegyünk a memóriabéli és JDBC adatelérési rétegek között.

Spring Web

Spring Web

- Hozzuk be a Spring Web starter függőséget, melynek segítségével az alkalmazást embedded Tomcat segítségével tudjuk futtatni.
- Készítsünk egy REST API-t Spring *controller*rel **egy** entitásunknak a következő CRUD műveletekhez, *ügyelve a konvenciók betartására*:
  - minden entitás lekérése
  - entitás lekérése ID alapján
  - entitás létrehozása
  - entitás részleges vagy teljes frissítése (nem szükséges mindkettő)
  - entitás törlése
  - keresés vagy szűrés az entitások között, aszerint, hogy mire hoztunk létre DAO metódust korábban.
- Az API-nk mindig térítsen vissza és fogadjon JSON formátumú body-t. A kiküldött/fogadott body-k ne a modellosztályokat használják, hanem erre a célra létrehozott DTO-kat. A *bejövő* DTO-kat validáljuk a Jakarta Validation API segítségével. A konvertálást DTO és modell között végezzük dedikált mapper osztályokban–ezek használhatnak generált kódot pl. a `mapstruct` segítségével.
- Ügyeljünk a megfelelő hibakezelésre, pl. hiányzó entitás lekérése ID szerint, létrehozáskor/frissítéskor fellépő validációs hibák lekezelése, stb. Ilyen esetekben dobjunk egy általunk készített vagy Spring által nyújtott kivételt, amelyet általánosan kezeljünk le és mappeljük megfelelő státuszkódra a `@ResponseStatus` annotáció segítségével.
- Teszteljük az alkalmazásunkat **részletesen** Postmannel.