


# L+: Pluszfeladatok

- Az alább felvázolt pluszfeladatok közül szabadon lehet választani. Egy-egy számozott feladat kiválthat egy szemináriumi hiányzást, vagy érhet 4 pontot.
- Akik ki szeretnék váltani az elméleti vizsgát, tetszőleges számú pluszfeladatot megvalósíthatnak, ellenkező esetben összesen **max. 4-et** lehet bemutatni. A vizsgát ki lehet váltani, ha pluszpontokkal és laborvizsgával együtt legalább 80 pont kijön a hallgatónak.
- Ezeket személyesen be kell mutatni egy előre meghatározott időpontban. A bemutatások iratkozás alapúak lesznek, így kövessétek a hirdetéseket. Az elfogadás feltétele a helyes demózás előkészített Postman parancsok segítségével, illetve ezekkel kapcsolatos kérdésekre való válaszkészség.
- Lehet ajánlani más, a tantárgy anyagát kiegészítő pluszfeladatot is, illetve különböző pluszfeladatok helyes kombinációja érhet többet is, mint az alappontok összege. Mindkettővel kapcsolatban egyeztessünk a laboránsokkal **a karácsonyi vakáció előtt**, hogy tisztázzuk a pontos értékeket.
- Formális feltételek:
  - A pluszfeladatokat készítsük el külön, erre dedikált *git branch*-en. Elegendő 1 ág létrehozása több pluszfeladat megoldására is.
  - Továbbra is feltétel, hogy ne legyenek statikus kódelemző jelzések, illetve hogy helyesen fusson le a `check` taszk a folyamatos integrációban (ld. korábbi laborfeladatok leírása).
  - Tartsuk a korábban vázolt konvenciókat, pl. megfelelő csomagnevek, beszédes változónevek, megfelelő függőségek, stb.

---

## A. Kliensoldali pluszfeladatok

### *A.1. Thin kliens beépítése (max. összesen 2-szer alkalmazható)*

- A *Spring Web MVC* modul segítségével építsünk be sablonmotrot alkalmazó thin klienst az alkalmazásunkba. Az idetartozó új kontrollereket külön csomagban tároljuk, és külön HTTP útvonalakra figyeljenek a REST API-hoz képest.
- Ez a feladat kizárja a 2-es pluszfeladatot–csak egy típusú kliensoldal írható le pluszfeladatként.
- Bármely sablonmotor használható, de konfiguráljuk megfelelően Spring `@Configuration`ek segítségével.
- A rendszer tartalmazzon UI elemeket **egy entitás** teljes menedzseléséhez (két entitás lefedése esetén 2-szer jár a pont).
- Dokumentáció:
  - `https://www.baeldung.com/spring-mvc-tutorial`
  - `https://docs.spring.io/spring-framework/docs/5.3.0/reference/html/web.html#mvc`
  - `https://www.baeldung.com/spring-template-engines`

### *A.2. Rich kliens beépítése (max. összesen 2-szer alkalmazható)*

- Készítsünk egy rich klienst egy JavaScript kliensoldali eszköz segítségével (pl. React, Vue, Angular, stb.).
- Bármely eszköz használható, illetve futtathatjuk külön folyamatban is az API szervertől. Bármely nem Java forráskódot helyezzünk külön alprojektbe a verziókövetőn.
- Ez a feladat kizárja a 1-es pluszfeladatot–csak egy típusú kliensoldal írható le pluszfeladatként.
- A kliensoldal végezzen aszinkron kéréseket az API szerver felé, amelynek eredménye alapján rajzolja újra magát.
- A rendszer tartalmazzon UI elemeket **egy entitás** teljes menedzseléséhez (két entitás lefedése esetén 2-szer jár a pont).
- Dokumentáció:
  - `https://www.lambdatest.com/blog/best-javascript-framework-2020/`
  - `https://spring.io/guides/tutorials/react-and-spring-data-rest/`
  - `https://www.smashingmagazine.com/2020/06/rest-api-react-fetch-axios/`

---

## B. Spring pluszfeladatok

### *B.1. Pagination & sorting*

- Vezessünk be *oldalváltás*t (pagination) és *rendezés*t (sorting) mindazon REST endpointokhoz, amelyek **kollekciót** térítenek vissza, pl. összes entitás keresése, szűrés, keresés. Eszerint egy hívás se térítse vissza a kollekció összes elemét, ehelyett térítsen csak *n*-t.
- Fogadjuk query paraméterekben a következőket: hányadik oldalt kérjük, hány érték tesz ki egy oldalt, mely oszlopunk szerint rendezzen s milyen irányba. Mindezen értékeknek legyen default értékük, hogy opcionális legyen megadásuk.
- Alkalmazzuk a Spring Data `PagingAndSortingRepository`-jában deklarált metódusokat megfelelően.
- Ezen HTTP hívások térítsék vissza az összes lehetséges elem számát az `X-Total-Count` response header-ben, hogy a kliens tudja, hogy hány oldalt kérhet le.
- Dokumentáció
  - `https://www.baeldung.com/spring-data-jpa-pagination-sorting`
  - `https://docs.spring.io/spring-data/rest/docs/2.0.0.M1/reference/html/paging-chapter.html`

### *B.2. Spring Specification és Criteria Query API használata Spring Data JPA-val*

- Vezessük be a támogatást a komplex rétegződő szűrésre/keresésre a *Spring Specification API* és *JPA Criteria API* segítségével. Ezeknek segítségével dinamikusan kombinálhatjuk a szűrési feltételeket olyan endpointokon, amelyek kollekciót térítenek vissza. Pl. hirdetéseket lehet szűrni minimális ár, maximális ár, lejárati dátum és más paraméterek szerint, de nem kötelező egyik paramétert sem megadni.
- Készítsünk POJO-t a szűrési feltételeknek, melyek query paraméterként megadhatóak a kontroller metódusnak, majd a repository rétegünk kiterjesztésével szűrjünk a megadott (nem null) feltételek szerint.
- Dokumentáció:
  - `https://www.baeldung.com/spring-data-criteria-queries`
  - `https://dzone.com/articles/using-spring-data-jpa-specification`

### *B.3. Spring Security beépítése*

- Vezessük be a *Spring Security*-t alkalmazásunkba. Védjük le az alkalmazásunk egyes endpointjait, hogy csak bejelentkezett felhasználók tudják elérni. Az alkalmazás nyújtson login és logout lehetőséget **adatbázisban** tárolt userek és *hash-elt* jelszavak alapján.
- Ha szeretnénk az itt létrehozott „felhasználó”-t új entitásként használni a 3-as pluszfeladatnál, szükséges levédeni, hogy csak admin módosíthassa a felhasználókat, illetve az új felhasználó entitás továbbra is kell kapcsolódjon valamelyik meglévő domain-specifikusok entitáshoz (pl. mint tulajdonos, kommenter, stb.)
- Dokumentáció:
  - `https://www.baeldung.com/spring-security-login`
  - `https://www.baeldung.com/spring-security-authentication-with-a-database`

### *B.4. Interceptor*

- Építsünk be pár doméniumban hasznos **interceptor**t – ezek Spring esetén hasonlóan viselkednek a `Filter`ekhez.
- Készítsünk interceptort, amely minden HTTP hívás végén kiírja a hívásról az infót (method, URL, válasz-státuszkód)
- Készítsünk egy második „middleware” interceptort, melyet alkalmazhatunk több endpointra, és amely valamilyen jogosultságot vagy általános feltételt ellenőriz.
- Dokumentáció
  - `https://www.baeldung.com/spring-mvc-handlerinterceptor`
  - `https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/servlet/HandlerInterceptor.html`

### *B.5. Spring caching*

- Építsük be a *Spring database caching* mechanizmusát lekérő metódusoknak.
- Egy lekérő `GET` hívás cache-elje JSON kimenetét memóriában, hogy kerüljük a felesleges adatbázis-kéréseket.
- Módosító hívások (`POST`, `PATCH`, `DELETE`), amelyek érintenék a korábbi híváseredményt, *töröljék* a releváns bejegyzést, hogy a legközelebbi `GET` hívásnak muszáj legyen regenerálnia.
- Alternatívan külső cache rendszer (pl. `redis`) is használható. Ennek Dockerbe való beépítése további pluszot jelenthet.
- Elegendő egy entitásra végigcsinálni a caching beépítését, de ügyeljünk az entitások közti esetleges kapcsolatokra.
- Dokumentáció:
  - `https://www.baeldung.com/spring-cache-tutorial`
  - `https://reflectoring.io/spring-boot-cache/`

### *B.6. Időszakos takarító taszk*

- Építsük be az alkalmazásunkba az *időzített taszkok* futtathatóságát.
- Készítsünk egy karbantartó taszkot, amely időszakosan lefut, HTTP hívások fellépésétől függetelenül. Lehet bármi olyasmi, amit nem szükséges egyből meghívni egy HTTP hívással, de időszakosan hasznos lefuttatni. Példák:
  - elavult entitások törlése - lejárt hirdetések; felhasználók, akik túl régen nem léptek be a rendszerbe, stb.

  - statisztikák generálása

- Dokumentáció:
  - `https://www.baeldung.com/spring-scheduled-tasks`
  - `https://spring.io/guides/gs/scheduling-tasks/`

---

## C. Servlet és Reflection/annotáció-feldolgozási pluszfeladatok

Ezekben a feladatokban használjuk a Java Reflection API-t, hogy automatizálásokat hajtsunk végre **az első 3 laborfeladatban létrehozott projekteken**.

- Reflection segédeszközként használhatjuk a Reflections könyvtárat.
- Szükség esetén a `web` projektünk kitelepítésekor végrehajthatunk inicializáló (pl. annotáció-feldolgozási) lépéseket. Az inicializálást lekezelhetjük egy `Listener` segítségével, ld. `servlet-annotations` példa.

### *C.1. Dependency injection konténer*

Egy, a korábbiaktól külön alprojektben implementáljunk egy saját Dependency Injection konténert, amelyet használjunk függőségként s amellyel cseréljük le a factory-k használatát. Támogassuk a következő annotációkat:

- `@Injectable` - egy osztályra ráhelyezve az osztály injektálhatóvá válik. Az annotációban választhassunk `singleton` (ezen belül `lazy` vagy `eager`), `prototype` (minden injekciós pontra más példány) és `pooled` (round robin megadott példányszámmal) stratégiák között.
- `@Inject` - egy adattagra ráhelyezve injektáljunk automatikusan példányt a fentinek megfelelően. Ha injektált beanek tartalmaznak további injektált adattagokat, azokat is állítsuk be, illetve jelezzünk hibákat körkörös függőség vagy injektálási jelölt hiánya esetén.

### *C.2. Konfigurációk automatikus beolvasása*

Egy, a korábbiaktól külön alprojektben implementáljunk egy keretrendszert, mely automatikusan képes konfigurációs értékeket betölteni állományokból futás közben. Használjuk függőségként s cseréljük le a meglévő konfigurációs változóink használatát. Működjön a következőképpen:

- `Configuration.getConfiguration(ConfigClass.class)` - adjon vissza egy singleton példányt a paraméterként megadott osztályból, melynek adattagjait konfigurációból (környezeti változó vagy konfig állomány) olvassa.
- `@ConfigValue` - a fentinek átadott osztály adattagjaira helyezhető annotáció; megadja, hogy milyen környezeti változó és/vagy konfigurációs kulcsot alkalmazzon, illetve mi legyen az alapértelmezett érték, ha ezek nincsenek megadva. Támogassunk string, szám, boolean és komplex POJO típusokat (ezutóbbit elegendő csak file-ból beolvasni), s adjunk megfelelő hibákat rossz átalakítás esetén.

### *C.3. ORM kapcsolatok*

Alakítsuk át az absztrakt JDBC DAO-nkat, hogy támogasson táblák közötti kapcsolatokat automatikusan. Hogyha egy entitás tartalmaz egy másik `BaseEntity`-t öröklő adattagot, feltételezzünk kapcsolatot a két háttérben rejlő tábla között, s képezzük le a **lekéréseket** (`findAll`, `findBy*`) megfelelően. Ehhez a feladathoz az absztrakt JDBC DAO kell létrehozza a táblákat is, ellátva őket a megfelelő külső kulcsokkal.

### *C.4. JSON (de)szerializálás automatizálása Servletekkel*

Egy, a korábbiaktól külön alprojektben készítsünk egy `DispatcherServlet`-nek megfelelő Servlet implementációt, majd használjuk a saját web projektünkben. Ez a Servlet feleljen meg a következő specifikációnak:

- Figyeljen minden HTTP útvonalra.
- Olvasson be egy csomag-gyökérnevet konfigurációból. A megadott gyökérben keresse az összes `@ReqContainer`-rel annotált osztályt (ezek felelnek meg a kontrollereknek), s példányosítson mindegyikből egyet a paraméter nélküli konstruktor segítségével.
- Ezek az osztályok tartalmazhatnak `@Req`-kel annotált metódusokat, melyek egy HTTP hívás feldolgozásáért felelősek. Ebben az annotációban megadhatunk egy HTTP metódust és egy útvonalat. A metódusok tartalmazhatnak 0 vagy 1 paramétert, illetve visszatéríthetnek bármilyen POJO-t.
- A dispatcherünk irányítsa el a HTTP hívásokat a `@Req` annotációk alapján, megfelelő hibakezeléssel. Ha a célpont metódus elvár egy paramétert, azt olvassuk a HTTP hívás body-jából (feltételezzük, hogy JSON), s ha térít vissza egy nem null választ, azt írjuk a HTTP válasz body-jába, ugyancsak JSON formájában.

---

## D. DevOps pluszfeladatok

### *D.1. Docker-compose*

- Használjuk a Spring *beépített profilozási mechanizmus*át ahhoz, hogy két külön profilt támogasson az alkalmazásunk: `dev` és `prod`
  - `dev` - in-memory adatbázist használ (pl. h2)
  - `prod` - korábban használt produkciós adatbázist használja
- A `gradle bootRun` és `gradle bootJar` taszkokat tudjuk futtatni mindkét profillal. A `bootJar` esetén az elkészült futtatható JAR ne tartalmazza a másik profil DB Driver függőségét.
- Készítsünk `Dockerfile`-t és `docker-compose.yml` állományt mindkét profil használatának: a `dev` variáns nem tartalmaz külső DB service-t, míg a `prod` igen. Teszteljük mindkettőt.
- Dokumentáció
  - `https://www.baeldung.com/spring-profiles`
  - `https://docs.spring.io/spring-boot/docs/1.2.0.M1/reference/html/boot-features-profiles.html`
  - `https://www.credera.com/insights/gradle-profiles-for-multi-project-spring-boot-applications/`

### *D.2. Cloud szolgáltatóra való kitelepítés*

- Használjunk egy CI (Continuous Integration) eszközt s telepítsük ki a konténerizált alkalmazásunkat egy cloud szolgáltatóra (Continuous Deployment). Egy példa lehet az Amazon ECS.
- Szükséges **mind az adatbázis**, mind a backend service-ek automatikus futtatása és összekötése az orchestration rendszer segítségével.
- Hogyha külön service-eket telepítünk ki CI segítségével, de nem automatizáljuk a kapcsolatukat (pl. a Heroku nem támogatja a compose-t), akkor a feladatért **2 pont** jár.
- A feladat egyszer hajtható végre–két külön cloud szolgáltatóért nem járt kétszer a pont.
- Dokumentáció:
  - `https://aws.amazon.com/ecs/`
  - `https://docs.docker.com/engine/context/ecs-integration/`