Spring

Imre Gábor

Q.B224

gabor@aut.bme.hu

Automatizálási és
Alkalmazott
Informatikai Tanszék

Tartalom

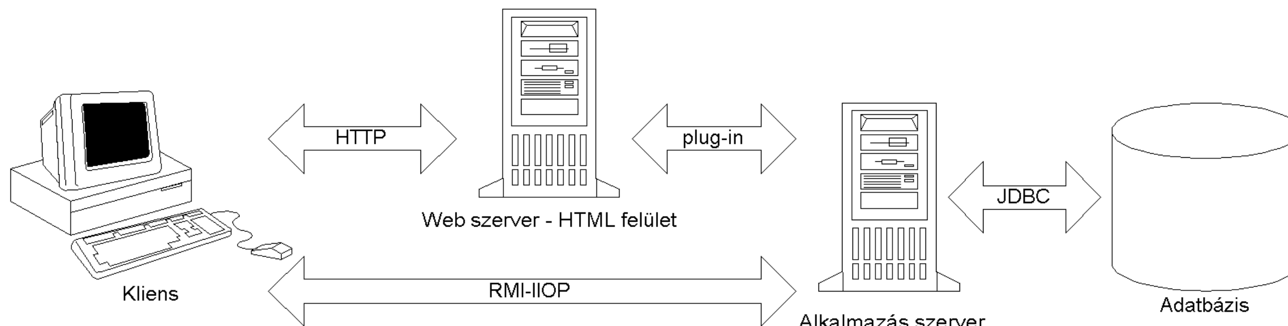
- Áttekintés
- Függőséginjektálás
- Spring Boot
- Adatelérés támogatása
- Tranzakciókezelés

Szerver oldali fejlesztés Java-ban

- Ismétlődő megoldandó feladatok → Keretrendszer bevezetése
- Tipikusan elvárt szolgáltatások
  - Többszálúság
  - Perzisztencia
  - Tranzakciókezelés
  - Biztonság
  - Távoli elérés (HTTP + egyéb)
  - Aszinkron üzenetkezelés
  - Skálázhatóság

Egy megoldás: Java Enterprise Edition

- A Java EE egy architektúra vállalati méretű alkalmazások fejlesztésére, a Java nyelv és internetes technológiák felhasználásával
- Nyílt szabvány, több lehetséges implementációval (interfészekkel definiált API-k)
- A Java EE szabványt megvalósító szoftver termék: alkalmazásszerver
  - pl. Glassfish, WebLogic, WebSphere, WildFly
- Háromrétegű architektúra



Spring: történeti áttekintés

- A Spring erdeti célja: a korai J2EE-nél fejlesztőbarátabb keretrendszer
- A Java EE később sok alapelvet innen vett át
- A Spring is épít több Java EE által definiált API-ra
- Springes alkalmazás futtatásához nincs szükség Java EE alkalmazásszerverre
- Idővel számos modul épült a Spring keretrendszerre
  - pl. Spring Boot, Spring Security, Spring Data

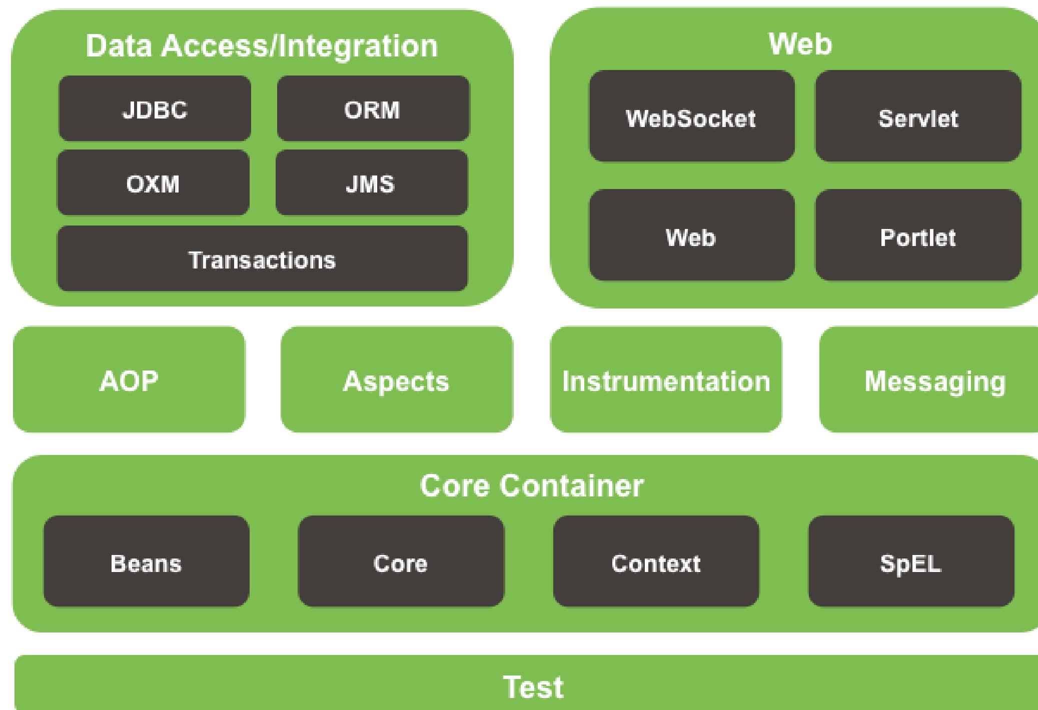
Alapelvek

- Nagyfokú rugalmasság, testreszabhatóság
  - A felhasznált modulok szabadon választhatók
  - Pl. JPA implementáció, view technológia a webrétegben szabadon választható
- Erős visszafelé kompatibilitás
- Szép, intuitív API, OO design
- Magas minőségű, tesztelt kód
- Pontos dokumentáció

Mit nyújt a Spring?

- Egy pehelysúlyú, nem-invazív konténert, amely segít az alkalmazás objektumainak konfigurálásában, „összedrótózásában”
- Adatelérést támogató osztályokat
  - JDBC-hez, több ORM technológiához
- A tranzakciókezelés egységes absztrakcióját, lecserélhető tranzakciómenedzserrel (lokális/JTA)
- Teljes AOP támogatást (pl. deklaratív tranzakciókezelést)
- Támogatást üzenetkezeléshez
- Támogatást webalkalmazások fejlesztéséhez
  - Spring MVC, WebSocket, WebFlux

Spring modulok



Tartalom

- Áttekintés
- Függőséginjektálás
- Spring Boot
- Adatelérés támogatása
- Tranzakciókezelés

Függőséginjektálás

- Dependency Injection (DI), Inversion of Control (IoC)
- Az objektumok nem drótozzák be az általuk használt konkrét osztályokat, hanem tagváltozó, metódus- vagy konstruktorparaméter formájában egy ún. injektortól kapja meg
- Az injektor végzi el a komplex objetumgráfok előállítását
- Komplex inicializáló kód megspórolható
- Különböző környezetekben, eltérő működést érhetünk el csak az injektor átkonfigurálásával
- Unit teszteknél mock objektumokkal helyettesíthetjük a tesztelendő objektum függőségeit

Függőséginjektálás: bevezető példa

- Egy `Stopwatch` osztály függ egy `TimeSource` osztálytól:

```
Public class Stopwatch {
  private final TimeSource timeSource;
  private long startedAt;

  public Stopwatch () {
    timeSource = new AtomicClock(...);
  }
  void start() { ... }
  long stop() { ... }
}
```

- Hátrány: be van drótozva a TimeSource implementáció → nem rugalmas

Függőséginjektálás: bevezető példa

- Egy megoldás egy Factory vagy ServiceLocator alkalmazása:

```
timeSource = DefaultTimeSource.getInstance();
```

- Hátránya:
  - több kód (Factory megírása),
  - unit tesztelés nehézkes, pl.

```
TimeSource original = DefaultTimeSource.getInstance();
DefaultTimeSource.setInstance(new MockTimeSource());
try {
  // tényleges teszt
  Stopwatch sw = new Stopwatch();
       ...
} finally {
   DefaultTimeSource.setInstance(original);
}
```

Függőséginjektálás: bevezető példa

- Megoldás függőséginjektálással

```
class Stopwatch {
   final TimeSource timeSource;
   @Autowired Stopwatch(TimeSource timeSource){
     this.timeSource = timeSource;
   }
   void start() { ... }
   long stop() { ... }
}
```

- Spring 4.3 óta a `@Autowired` elhagyható, ha csak egy konstruktor van
- A `StopWatch` is injektálható pl. egy GUI osztályba:

Függőséginjektálás: bevezető példa

```
class StopwatchWidget {
  @Autowired StopwatchWidget(Stopwatch sw) { ... }
  ...
}
```

- Fontos, hogy a `StopwatchWidget`-et az injektortól kérjük el, ne közvetlenül példányosítsuk, mert az injektor tudja a függőségeket megkeresni, szükség esetén azokat is létrehozni, akár rekurzívan
- Unit tesztnél explicit átadható a mock:

```
void testStopwatch() {
  Stopwatch sw = new Stopwatch(new MockTimeSource());
  ...
}
```

A függőséginjektálás Springben

- Tagváltozó-, konstruktor- és setterinjektáltást is támogat
- A Spring által kezelt osztályok megnevezése: bean (DI + esetleg más szolgáltatásokat kap)
  - A beanekbe injektált más beanek a függőségek, vagy kollaborátorok
- Minden springes modul is erre épül, így azok viselkedését az általuk nyújtott beanekbe injektált más (beépített vagy saját) beanekkel tudjuk testre szabni
- A szükséges konfiguráció evolúciója:
  1. XML fájl
  2. Annotációk + Java osztályok (JavaConfig)
  3. Spring Boot: automatikus default konfiguráció classpath alapján
     - Testreszabható .properties vagy .yaml fájllal
     - Teljesen felülírható JavaConfig-gal

Spring beanek annotációkkal

- A beaneket annotációval jelöljük meg (`@Component` vagy más, pl. `@Service, @Controller, @Repository`)
- `@Autowired`-del jelöljük meg az injektálási pontokat

```
@Service
public class ProductService {
  @Autowired
  private PriceCalculator priceCalculator;
...
}
@Component
public class PriceCalculator {...}
```

Spring beanek definiálása konfig osztályban

- Ha egy injektálási pontra több bean jelölt is van (több gyerekosztály/interfész-implementáció az adott típusból), egyértelműsíteni kell, melyiket akarjuk használni → @Bean metódus, @Configuration osztályban elhelyezve

```java
@Configuration
public class MyAppConfig {
  @Bean
  public MyService myService() {
       return new MyServiceImpl();
  }
}
```

- Meg kell mondanunk a Springnek, melyik package-ekben keresse a bean-eket
  - Spring Boot esetén by default a „root” package alatt (akár alpackage-ekben), amelyben a main metódust tartalmazó osztályunk van, de a @ComponentScan annotációval másokat is hozzáadhatunk (szintén konfig osztályra kell tenni)
- A Spring Boot megtalálja a root package alatti konfig osztályokat, ha másokat akarunk behúzni → @Import(OtherConfig.class)

Beanek élettartama

- By default minden bean singleton (app. contexten belül!)
- További lehetőségek:
  - prototype (minden injektálási ponton új példány)
  - request (egy HTTP kérés idejéig)
  - session (egy HTTP session idejéig)
  - application (a webalkalmazáshoz kötött →több app. contextre közös)
  - websocket (egy websocket élettartamához kötött)
- Scope definiálása, pl.

```
@Bean
@Scope("session") //konstans is van rá
public UserPreferences userPreferences() {
    return new UserPreferences();
}
```

Vagy

```
@SessionScope
@Component
public class UserPreferences {    // ...}
```

További Spring DI lehetőségek

- Életciklus callback metódusok definiálhatók a beanekben pl. `@PostConstruct` és `@PreDestroy` annotációkkal
- Nem kielégíthető függőségek kezelése (nem pontosan egy kért típusú bean)
  - Egyetlen konstruktor `@Autowired` nélkül → kivétel
  - `@Autowired` setteren, tagváltozón vagy konstruktoron → kivétel
  - `@Autowired (required=false)` → null marad, kivéve hiányzó default fallback konstruktor esetén, akkor kivétel
  - `@Nullable` (konstruktor esetén az argumentum kapja) → null marad
  - `Optional<X>` → null marad
- Több jelölt kezelése:
  - Az összes elkérése: `@Autowired X[]` vagy `Set<X>` vagy `Map<String, X>`
  - Elsődleges bean kijelölése: `@Primary`
  - Jelöltek minősítése, és választás közöttük: pl. `@Qualifier("main")`

Tartalom

- Áttekintés
- Függőséginjektálás
- Spring Boot
- Adatelérés támogatása
- Tranzakciókezelés

Spring Boot: Alapok

- Cél: a Spring egyszerűbb használata
- Megfelelő jar függőségek révén automatikus default konfiguráció
- A default-tól való eltérés beállítása
  - Egyszerű esetben properties vagy yaml fájlban
  - Egyébként JavaConfig-gal
  - XML-t sosem kell írni
- Webalkalmazások is önálló Java alkalmazásként futtathatók, beágyazott Tomcat/Jetty/Undertow webkonténeren → nem szükséges külön deploy

Spring Boot: Hello World

```
@SpringBootApplication
public class HelloWorldApplication implements CommandLineRunner {

  @Autowired
  HelloService helloService;

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

  @Override
  public void run(String... args) throws Exception {
    System.out.println(helloService.hello());
  }
}
```

```
@Component
public class HelloService {

  public String hello() {
    return "Hello world";
  }
}
```

Függőségek kezelése

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.1.5.RELEASE</version>
  <relativePath/>
</parent>
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter</artifactId>
  </dependency>
  <!-- további függőségek … -->
</dependencies>
```

- Tipikus megoldás: a spring-boot-starter-parent-et jelöljük meg szülőként
  - Ez (közvetve) felsorol egy sor függőséget a *dependencyManagement* tagben (+ plugineket is)
- A *dependencies* tagben felsoroljuk azokat, amiket használni szeretnénk, de a *version* megadása itt már nem szükséges
  - A spring-boot-starter-parentben lévőek garantáltan kompatibilisek egymással
  - De felül is definiálhatjuk a menedzselt verziót, maven property-kel, pl.

```
<hibernate.version>5.2.9.Final</hibernate.version>
```

- Ha más szülő projektet szeretnénk, arra is van megoldás

Kód struktúra

- Az application osztályt célszerű (nem kötelező) egy "root" package-be tenni, a többi osztály az alatti alpackage-ekben legyen pl.
  - com.myapp.model: entitások
  - com.myapp.service: üzleti logikai osztályok
  - com.myapp.web vagy com.myapp.controller: Spring MVC kontroller osztályok
- A @SpringBootApplication valójában 3 másik annotációra alias
  - **`@EnableAutoConfiguration`** → ez engedélyezi, hogy a Spring Boot megkeresse és érvényre juttassa a classpath-on elérhető automatikus konfigurációkat
  - **`@ComponentScan`** → emiatt fogja megtalálni a konfig osztályokat és beaneket az alpackage-ekben
  - **@Configuration** → emiatt @Bean metódusokat tehetünk magába az application osztályba is

Konfigurációs osztályok

- A JavaConfig preferált az XML-lel szemben
- A konfigot szétoszthatjuk több **`@Configuration`** osztályban, ezek érvényre jutnak, ha
  - a root package alatt vannak (akár alpackage-ekben), és az application osztály **`@ComponentScan`** vagy **`@SpringBootApplication`** annotációt kap
  - vagy explicit behúzzuk őket, pl. **`@Import(MyConfig.class)`**
    - Tipikus, ha külön library-ben van a konfig osztály, és nem a mi root package-ünk alatt
- Ha mégis XML-t használnánk, az **`@ImportResource`** annotációval húzható be egy **`@Configuration`**-ös osztályon

Autokonfiguráció

- A **@EnableAutoConfiguration** (vagy **@SpringBootApplication**) annotációval engedélyezzük az autokonfigurációkat, by default az összeset a classpath-on
- Az autokonfigurációk "intelligensek", pl. ha nem webes az alkalmazásunk, nem próbál Spring MVC-t konfigolni, vagy ha mi magunk JavaConfig-ot írunk valamire, nem fogja felülírni, tipikus példa:

```
@Configuration
@ConditionalOnWebApplication
@ConditionalOnClass({ Servlet.class, DispatcherServlet.class,
WebMvcConfigurerAdapter.class })
@ConditionalOnMissingBean(WebMvcConfigurationSupport.class)
...
public class WebMvcAutoConfiguration {…}
```

- Egyes autokonfigurációk explicit kikapcsolása:
  - **@EnableAutoConfiguration(exclude={DataSourceAutoConfiguration.class})**
  - **spring.autoconfigure.exclude** propertyve
- Autokonfigurációk nyomon követése: alkalmazás indítása --debug kapcsolóval

Property alapú konfiguráció

- Az autokonfig osztályok működése testre szabható propertykkel
- + saját kódban is igény lehet, hogy külső konfig fájlból vegyünk adatokat, pl.

```
@Component
public class MyBean {

  @Value("${myprop}")
  private String myField;
  // ...
}
```

- A property értéket megadhatjuk pl. a classpath-ra helyezett application.properties fájlban:

```
myprop=value
```

Property alapú konfiguráció

- application.properties helyett application.yml is használható → sokszor tömörebb, pl.

```
environments.dev.url=http://dev.bar.com
environments.dev.name=Developer Setup
environments.prod.url=http://foo.bar.com
environments.prod.name=My Cool App
```

```
environments:
  dev:
    url: http://dev.bar.com
    name: Developer Setup
  prod:
    url: http://foo.bar.com
    name: My Cool App
```

- Az application.properties/.yml fájlokat több helyről is betölti a Spring Boot, a precedencia (minden helyen belül előbb a .properties, utána a .yml):
  - Az aktuális könyvtár /config könyvtárában
  - Az aktuális könyvtár
  - A classpath /config könyvtára
  - A classpath
- Ha egy property csak egy helyen van definiálva, az érvényre jut, bárhol is van
- De ha ugyanaz a property több helyen kap értéket, a sorrend szerinti legkorábbi helyen lévő jut érvényre

Property alapú konfiguráció

- A propertyk nemcsak fájlokból jöhetnek, hanem számos más helyről, így rugalmasan felüldefiniálhatók, pl.
  - parancssori argumentumok,
  - webes context-param,
  - JNDI,
  - **`System.getProperties()`**,
  - **`@PropertySource`** annotáció,
  - programozottan
- A teljes lista, a precedencia sorrendjében:

http://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-external-config

Property alapú konfiguráció

- A propertyk egyik elérési módja kódból a **`@Value("${myprop}")`** annotáció
- Ha sok propertynk van, amelyek akár hierarchikusak, célszerű külön osztályt bevezetni nekik, pl.

```
@ConfigurationProperties("foo")
public class FooProperties {
  private boolean enabled; //foo.enabled property
  private InetAddress remoteAddress;
//foo.remoteAddress
  private final Security security = new Security();

  //... getterek, setterek

  public static class Security {
    private String username; //foo.security.username
    private String password; //foo.security.password
    //... getterek, setterek
  }
}
```

```
@Service
public class MyService {
  @Autowired
  private final FooProperties properties;
}
```

```
@Configuration
@EnableConfigurationProperties(FooProperties.class)
public class MyConfiguration { }
```

vagy

```
@Component
@ConfigurationProperties(prefix="foo")
public class FooProperties {
  // ...
}
```

Property alapú konfiguráció

- A **`@ConfigurationProperties`** előnyei:
  - Egyszerre injektálhatunk több, akár hierarchikus property-t
  - Nem tudjuk elgépelni a **`@Value`**-nak átadott értéket
  - Megengedőbb a property nevekkel, pl. person.firstName, person.first-name, person.first_name, PERSON_FIRST_NAME mind megfelelő
  - A spring-boot-configuration-processor metaadatokat tud generálni az osztályokból, így a konfig fájlok szerkesztésekor kódkiegészítés is működhet az IDE-kben

Profile-ok

- Cél: bizonyos környezetekben más beállításokkal (konfig osztályok + propertyk) fusson az alkalmazás
- Hasonlít a maven profile-okhoz, de ott a build során adjuk meg a profile-okat, míg itt futtatáskor
- Profile aktiválása: spring.profiles.active nevű property-vel, vagy paracssori kapcsolóval, pl.

`spring.profiles.active=dev,hsqldb` vagy

`--spring.profiles.active=dev,hsqldb`

- Ha mindkét módon megadjuk, a parancssori *felülírja* a property-ben megadottakat. Ha azokat megtartva plusz profile-okat szeretnénk aktiválni:
  - `spring.profiles.include`, vagy
  - `SpringApplication.setAdditionalProfiles()`

Profile-ok

- Profile-függő konfigurációs osztály példa:

```
@Configuration
@Profile("production")
public class ProductionConfiguration
{
// ...
}
```

- Profile-függő konfigurációs fájlok:
- application-{profile}.properties/.yml
- A profile aktiválásakor lép életbe, magasabb precedenciával, mint a profile-mentes verzió
- Ha több profile-t adunk meg, az utolsóhoz tartozó konfig fájlnak van a legnagyobb precedenciája

Tartalom

- Áttekintés
- Függőséginjektálás
- Spring Boot
- Adatelérés támogatása
- Tranzakciókezelés

Adatelérés támogatása Springben

- Támogatás a JDBC, JPA, JDO, Hibernate, iBatis egyszerűbb használatához:
  - Injektáltatható az **EntityManager**, Hibernate-es **SessionFactory**, **JDBCTemplate**
  - Egységes tranzakciókezelési modell (lokális/globális tranzakciók, akár deklaratívan)

JDBCTemplate

- Mit old meg?:
  - **Connection** nyitása
  - **Statement** létrehozása
  - iteráció a **ResultSet**-en
  - kivételek kezelése
  - **Connection**, **Statement**, **ResultSet** bezárása

JDBCTemplate példa

```
@Repository
public class JdbcMovieFinder implements MovieFinder {
  private JdbcTemplate jdbcTemplate;

  @Autowired
  public void init(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
  }
  public void sampleQueries() {
        int countOfActorsNamedJoe = this.jdbcTemplate.queryForInt(
        "select count(*) from t_actor where first_name = ?", "Joe");

        String lastName = this.jdbcTemplate.queryForObject(
        "select last_name from t_actor where id = ?",
        new Object[]{1212L}, String.class);
  }
}
```

JDBCTemplate példa

```
public List<Actor> findAllActors() {
   return this.jdbcTemplate.query(
   "select first_name, last_name from t_actor",
   new ActorMapper());
}
private static final class ActorMapper implements RowMapper<Actor> {
   public Actor mapRow(ResultSet rs, int rowNum) throws SQLException{
          Actor actor = new Actor();
          actor.setFirstName(rs.getString("first_name"));
          actor.setLastName(rs.getString("last_name"));
          return actor;
   }
}

this.jdbcTemplate.update(
"insert into t_actor (first_name, last_name) values (?, ?)",
"Leonor", "Watling");
```

JPA használata

- Az EntityManager injektáltatható a spring beanekbe a JPA előadáson látott módon
- Spring Boot-tal szinte konfigmentes megoldás:
  - classpath-ba spring-boot-starter-data-jpa
  - application.properties-be:
    - spring.datasource.jndi-name=jdbc/mydb
      - és a szerveren kell a JDNI név mögé regisztrálni a JDBC URL-t, usert, jelszót, VAGY
    - spring.datasource.url=jdbc:mysql://localhost/test
      spring.datasource.username=dbuser
      spring.datasource.password=dbpass
      - Ilyenkor a spring fog létrehozni ezekkel az adatokkal DataSource-t, connection poolinggal, és azt használja
  - persistence.xml sem szükséges, minden benne lévő konfig az application.properties-ben megadható

Spring Data

- Külön projekt az adatelérés támogatására, egyik modulja a Spring Data JPA
- Magasabb szintű támogatás, mint az EntityManager injektálása (azt maga a Spring keretrendszer tudja)
- Az ún. Repository interfészek segítségével az adatelérési kód nagy része megspórolható

```
public interface CrudRepository<T, ID extends Serializable>
  extends Repository<T, ID> {

  <S extends T> S save(S entity);

  Optional<T> findById(ID primaryKey);

  Iterable<T> findAll();

  Long count();

  void delete(T entity);

  boolean exists(ID primaryKey);
...
}
```

Spring Data repository-k

- A repositoryk használata
  - Saját entitásra specifikus repository-t írunk, pl.

```
interface PersonRepository extends
      JpaRepository<Person, Long> { }
```

  - Konfig Spring Boot-tal:
    - spring-boot-starter-data-jpa függőség
    - datasource definiálása application.properties-ben
  - Injektálás másik beanbe:

```
@Autowired private PersonRepository repository;
```

- Az implementációt a Spring Data generálja!

Spring Data repository-k

- Lekérdezések
  - Automatikusan generált lekérdezések az interfészbe felvett findBy… metódusok nevei alapján, pl.

```
interface PersonRepository extends JpaRepository<Person, Long> {
   List<Person> findByLastname(String lastname);
   List<Person> findDistinctPeopleByLastnameOrFirstname(String lastname,
         String firstname);
}
```

- Hosszú metódusnevek elkerülésére

```
interface PersonRepository extends JpaRepository<Person, Long> {

  @Query("SELECT DISTINCT p FROM Person p WHERE p.lastName=:lastname
 OR o.firstName=firstName")
  List<Person> findByNames(String lastname, String firstName);

  //A Person entitáson definiált Person.findByNames2 named queryt
 fogja meghívni
  List<Person> findByNames2(String lastname, String firstName);
}
```

A Spring Data további lehetőségei

- Lapozás, rendezés támogatására `PagingAndSortingRepository`
  - `Iterable<T> findAll(Sort sort);`
  - `Page<T> findAll(Pageable pageable);`
- Repository közvetlen publikálása REST interfészen (spring-data-rest)
- A JPA mellett más adatelérési technológiák is támogatottak:
  - JDBC, MongoDB, Neo4j, Redis, Couchbase, Solr, Elasticsearch

Tartalom

- Áttekintés
- Függőséginjektálás
- Spring Boot
- Adatelérés támogatása
- Tranzakciókezelés

JPA entitások és tranzakciók

- A persistence.xml-ben megadható, hogy lokális vagy JTA (elosztott) tranzakciókezelést választunk
- Lokális: az EntityManager-től elkért EntityTransaction interfészen keresztül kell kezelni a tranzakciót
  - Tipikus Spring alkalmazások ezt használják (ezért a Spring Boot ezt autokonfigolja)
- JTA:
  - Elosztott (több DB-t átívelő) tranzakciók esetén mindenképp szükséges
  - A 2-fázisú kommit protokollt támogató ún. tranzakciómenedzsert a JTA API-n keresztül megszólítva indítjuk a tranzakciókat, ilyenkor a tranzakció a perzisztencia providert kikerülve, közvetlenül a JDBC drivert szólítja meg
  - Java EE környezetben tipikus ez a tipikus, mert az alkalmazásszerverben ott a JTA
  - Spring esetén is megoldható,
    - ha a Springes alkalmazás Java EE alkalmazásszerveren fut,
    - vagy sima webkonténer esetén beágyazott JTA-s tranzakciómenedzser integrálásával, pl. Atomikos, Bitronix

JPA entitások és tranzakciók

- Az entitások módosításával járó műveleteket (persist, merge, remove, refresh) csak tranzakción belül szabad meghívni
- Ha a lekérdezésekkel megtalált példányok tranzakción belül olvasódnak be → menedzselt állapotba kerülnek
- Két finomhangolási lehetőség anomáliák megakadályozására/detektálására, alacsonyabb izolációs szint esetén
  - Optimista konkurenciakezelés (@Version)
  - Explicit zárkezelés (em.lock())

Tranzakciókezelés Springben

- Egységes API a tranzakciókezelésre, ami mögött bekonfigolható a konkrét tranzakciómanager implementáció
  - Működhet JDBC connection szinten
  - Használhatja a JPA EntityTransaction-jét
  - Elérhet elosztott tranzakciómenedzsert JTA API-n keresztül
  - Használhatja a JDO API-t (a JPA régebbi alternatívája)

Tranzakciómenedzser konfig példa

```
@Configuration
@EnableTransactionManagement
public class TxConfig{

    @Bean
    public PlatformTransactionManager transactionManager(
        EntityManagerFactory emf){

            JpaTransactionManager transactionManager =
                new JpaTransactionManager();
            transactionManager.setEntityManagerFactory(emf);

            return transactionManager;
    }
}
```

- Spring Boot esetén nem szükséges, ez az autokonfig

Tranzakciókezelés típusok

- Programozott:
  - A Spring által adott API-n keresztül kódból indítjuk/zárjuk le a tranzakciót
  - Ritkán használjuk
- Deklaratív:
  - Metódus szinten annotációkkal vagy XML-ben szabályozzuk a tranzakciók indítását/végét
  - Metódusnál kisebb egységekről nem rendelkezhetünk

Programozott tranzakciókezelés

```
@Autowired
PlatformTransactionManager txManager;
...
DefaultTransactionDefinition def = new DefaultTransactionDefinition();
def.setPropagationBehavior(TransactionDefinition.PROPAGATION_REQUIRED);
TransactionStatus status = txManager.getTransaction(def);
try {
    // execute your business logic here
}
catch (MyException ex) {
    txManager.rollback(status);
    throw ex;
}
txManager.commit(status);
```

Deklaratív tranzakció példa

```
public class LogService {
  @PersistenceContext
  EntityManager em;

  @Transactional
  public void create(LogItem logItem) {
        em.persist(logItem);
  }
  ...
}
```

- A `@Transactional` paraméterezhető

A @Transactional paraméterei

- `rollbackFor`: milyen kivételek esetén legyen rollback (default: a RuntimeException és gyermekei)
  - További lehetőségek: `noRollbackFor`, `rollbackForClassName`, `noRollbackForClassName`
- `timeout`
- `isolation`: izolációs szint,
  - de lehet, hogy az adott technológia nem támogatja (pl. JPA)
- `value`: tranzakciómanager azonosító
  - Ha egy alkalmazásból több különböző adatbázist érünk el, lehet, hogy azokhoz külön tranzakciómanagert akarunk
- `propagation`: mi történjen, ha tranzakcionális metódusból másik tranzakcionális metódust hívunk

Propagation

Propagation érték	Létező tranzakció	Metódus tranzakciója
REQUIRED (default)	Nincs T1	T1 T1
REQUIRES_NEW	Nincs T1	T1 T2, T1-től független, T1 felfüggesztve
SUPPORTS	Nincs T1	Nincs T1
NOT_SUPPORTED	Nincs T1	Nincs Nincs, T1 felfüggesztve
MANDATORY	Nincs T1	Kivétel T1
NEVER	Nincs T1	Nincs Kivétel
NESTED (nem mindenhol támogatott)	Nincs T1	T1 T2, de T1 része

Repository metódusok tranzakciókezelése

- Az örökölt módosító Repository metódusokon @Transactional annotáció van → ha egy másik bean metódusból egyetlen repository metódust hívunk, megfelelő a működés
- De ha több repository metódust hívunk (pl. egy service metódusból), minden repository metódus külön tranzakció lesz, pedig általában nem ezt szerenénk
- Megoldás: a service metódusra is tegyük ki a @Transactional annotációt → a REQUIRED propagation miatt egy tranzakcióba fogjuk a repository metódusokkal végzett műveleteket

Deklaratív tranzakciók működése