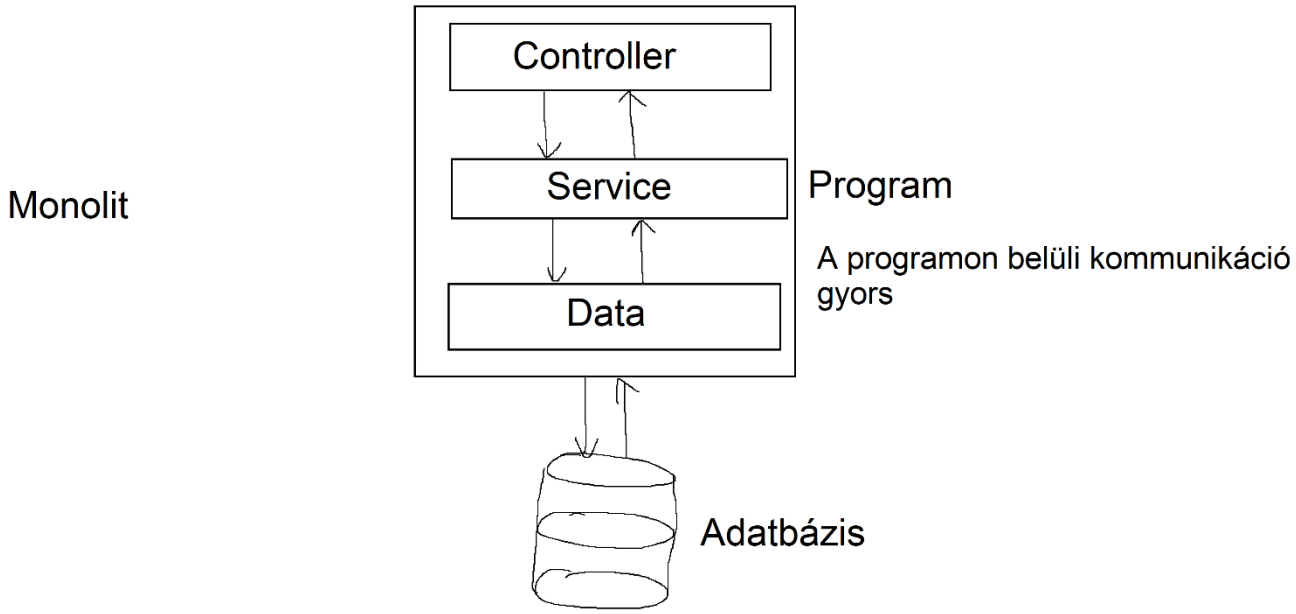
Webfejlesztés – 2022

JSF

A JSF a **Java Server Faces** rövidítése, ami egy szerveroldali, komponens alapú felhasználó felület keretrendszer. A komponens alapú itt azt jelenti, hogy a felhasználó felületet kis komponensekből állítjük össze. Ennek eléréséhez a JSF rendelkezik egy komponenskönyvtárral, amiben általános komponensek találhatóak. Ezek segítségével fog elkészülni a **webes** felhasználói felület. Java alapú, monolit, MVC keretrendszerről van szó. A monolit jelentését az alábbi ábra szemlélteti:



A JSF-es **komponensek újrafelhasználhatóak**, ez elősegíti a konzisztenciát. Jól támogatja az EL kifejezéseket (alakjuk: #{…}). Hátránya, hogy **nem skálázható jól**, a hirtelen megnövekedett, nagy terhelést nem bírja.

Az MVC-ből származó előnyök leszek a könnyű fenntarthatóság, kiterjeszthetőség és tesztelhetőség. Mivel szerveroldali keretrendszerről van szó, ezért ha valami hiba van, azt a szervernél érdemes keresni.

A JSF view template-eknek, vagy **facelet-eknek** nevezett XML fájlokat használ a megjelenítési modell leírására. A kéréseket a FacesServlet dolgozza fel, ami ezután betölti a megfelelő view template-et, felépíti a komponensfát, kezeli az eseményeket és létrehozza (generálja) a választ (többnyire HTML vagy XHTML formátumban) a kliensnek. A felhasználói felület komponenseit (és egyéb objektumokat) minden lekérés végén elmenti, majd ugyanazon view következő előállításakor újra betölti.

A JSF részei: event-ek, listener-ök, felhasználói felület komponensek, converter-ek, validator-ok, …

1. **Renderer:** Tulajdonképpen az a feladata, hogy a megírt kódot szétbontsa HTML-re és Javascriptre. Ez fogja a komponenseket reprezentáló HTML-kódot legenerálni.
2. **Validator:** Az egésznek az az alapötlete, hogy a felhasználó által a webes felületen bevitt adatokat ellenőrizni kell, hogy megfelelőek, helyesek-e. Itt tulajdonképpen a felhasználói felület komponenseinek validálása, ellenőrzése zajlik. Ez történhet JavaScripttel, Java-val, illetve a Controller, Service szintjén, plusz az adatbázisban is elvégezhető.
3. **Converter:** A konverter dolga az adatok átalakítása lesz – amint azt a neve is mutatja. A felhasználó által a webes felületen bevitt adatok string-ek lesznek. A konverter feladata az, hogy ezen string-eket átalakítsa Java-objektumokká, és fordítva. Igazából itt a komponensek adatainak konvertálása folyik.
4. **Event-ek és Listener-ök:** A felhasználói felületen a komponensekkel interakcióba lehet lépni, van egy eseménykészletük. Egy gombnak például a következő eseményei vannak: onClick, onHover, onChange, onBlur, stb. Ezen események közül értelemszerűen csak annyit kell megírni és felhasználni, amennyire szükségünk van. Az ilyen események az event-ek. Ezek tulajdonképpen „jelek”, amik akkor kerülnek „kibocsátásra”, amikor a felhasználó interakcióba lép egy komponenssel. A listener-ök dolga, hogy figyeljenek, hogy egy event/esemény bekövetkezzen, és megfelelően lekezeljék a bekövetkezett eseményt. A listener-öket általában párosítani kell a komponensekkel.

Managed Bean: A Managed Bean egy, a JSF-hez regisztrált Java-osztály (faces-config.xml-ben), ami a felhasználói felület és az üzleti logika közötti interakciót lehetővé teszi. A @ManagedBean annotációval szokták ezeket az osztályokat megjelölni. Például:

```java
package com.journaldev.jsf.helloworld;

import java.io.Serializable;
import javax.faces.bean.ManagedBean;
import javax.faces.bean.SessionScoped;


@ManagedBean(name="helloWorld")
@SessionScoped
public class HelloWorld implements Serializable{

    private static final long serialVersionUID = -6913972022251814607L;

    private String s1 = "Hello World!!";

    public String getS1() {
        System.out.println(s1);
        return s1;
    }

    public void setS1(String s1) {
        this.s1 = s1;
    }

}
```

A @ManagedBean annotációval jelezzük, hogy ez az osztály egy Managed Bean lesz. A name=”helloWorld”-del pedig nevet is adhatunk a bean-nek. Általában érdemes valamilyen találó névvel ellátni a managed bean-eket. Request scope-pal, session scope-pal vagy application scope-pal lehet őket definiálni. Automatikusan kerülnek létrehozásra, amikor szükség van rájuk. A Managed Bean-ek igazából Java Bean osztályok. A Java beaneknek nevez minden olyan osztályt, amely tulajdonságokat és eseményeket tesz elérhetővé külső környezet számára. A Java Bean egy olyan osztály, amely:

1. Szerializálható, azaz implementálja a Serializable inferfészt,
2. Van publikus, paraméter nélküli konstruktora,
3. A tulajdonságokhoz való hozzáférést getter-ek és setter-ek segítségével teszi lehetővé.

(A kép forrása: https://www.digitalocean.com/community/tutorials/jsf-tutorial-for-beginners

)

Backing Bean: Egy JSF alkalmazásnak lehet egy vagy több backing bean-je is, amelyek egyfajta managed bean-ek, amik hozzá társíthatók az oldalon használt komponensekhez. Request scope-pal lehet őket definiálni. Az oldal komponenseihez tartozó objektumokat adattagokként tartalmazzák.

A JSF alkalmazásoknak van egy életciklusa, amin végigmennek:

A JSF esetében alapvetően kétfajta kérés létezik: az initial és a postback. Az initial request (HTTP GET) hatására a szerver oldalon létrejön a kért oldalhoz tartozó

komponensfa, melynek a csomópontjai az oldalon található elemek lesznek (UIForm, UIOutput, UIInput, stb...). Az initial kérések során csak a Restore View és az Render Response fázis fog lefutni, azaz felépül a komponensfa (+ eseménykezelők, validátorok, konverterek jóváhagyásra kerülnek, majd a view elmentésre kerül a FacesContext-be) ami alapján lerenderelődik a HTML oldal. Postback kérések esetén tipikusan egy form mezői kerülnek átküldésre a szerver oldalra (HTTP POST), ahol létezik már az oldalhoz tartozó komponensfa, egy korábbi initial kérés következtében. Alapesetben a postback kérések elküldésekor mind a 6 JSF fázis lefut, ahol is a komponensfa aktualizálása mellett a megfelelő lépések után a managed bean-ekbe beíródnak az értékek.

1. **A nézet visszaállítása (Restore view): A Restore View fázis visszatölti a komponensfát,** azaz végrehajtódik a dekódolás, **amennyiben az már létrejött egyszer, vagy újat hoz létre, ha először kerül megjelenítésre az oldal.** A már korábban megjelenített oldal esetén az összes komponens a legutóbbi állapotába töltődik vissza. Amennyiben a lekérésben nincs adat a JSF a Render Response fázisban folytatja a végrehajtást és a köztes lépések kimaradnak (például ez történik, mikor először jön létre a komponensfa).
2. **A kérésben szereplő értékek érvényesítése (Apply request values):** A JSF implementációk végiglépkednek a komponensfán, és az összes komponens kiválaszthatja melyik lekérési adat tartozik hozzá, majd a kiválasztott adatokat eltárolhatja. Az itt keletkező események, például egy gomb megnyomása esetén, egy eseménysorban tárolódnak, amit majd a fázis lezárulása után feldolgozhatnak az azokra feliratkozott eseménykezelők.
3. **Validációk (Process validations):** A Process Validations fázisban először a felhasználó által a lekérdezésbe tárolt szöveges adatok kerülnek konvertálásra. Amennyiben adtunk meg validátorokat az oldalhoz, azok is ebben a fázisban futnak le. Ha ebben a fázisban hiba történik a Render Response fázis következik, ahol a hibaüzenetekkel együtt megjelenítésre kerülnek a felhasználó által beírt adatok is, így azok javíthatóak lesznek, és nem kell újra begépelni őket. A konvertálások és az értékek helyességének ellenőrzése után a JSF feltételezi, hogy a megadott adatok helyesek, így továbblép az Update Model Values fázisba.
4. **A modell értékeinek frissítése (Update model values):** Az Update Model Values fázisban a komponensekhez kötött bean-ek adattagjai kerülnek beállításra, azaz a tulajdonságokat beállító metódusok futnak le.

5. **Az alkalmazás meghívása (Invoke application):** Az Invoke Application fázisban az űrlap elküldését aktiváló gomb vagy link action attribútuma által indukált metódusok kerülnek végrehajtásra. A metódus tetszőleges műveletek végrehajtása után egy eredmény Stringet szolgáltat, amit a navigáció-kezelő használ fel, a következő megjelenítendő oldal meghatározására.
6. **A válasz generálása (Render response):** Végül a Render Response fázis következik, amikor is a keletkezett választ kódolja a JSF és elküldi a kliensnek.

Scope-ok:

1. **@RequestScoped:** Ez rendelkezik a legkisebb hatáskörrel, az itt elhelyezett objektumok csak a lekérdezés ideje alatt – a HTTP kérés-válasz leteltéig - élnek. A request scope-ban elhelyezett bean-ből minden lekérdezéskor új példány jön létre, és ez a példány törlődik a válasz elküldésekor.
2. **@SessionScoped:** A következő, az előzőnél tágabb scope. **A session az egyazon kliens által történő sorozatos csatlakozást jelenti.** A servlet konténer minden egyes felhasználót nyomon követ a sessionazonosítók segítségével. Ezek az azonosítók vagy sütiken keresztül, vagy a kliensnél letiltott süti küldés esetén az URL-en keresztül kerülnek átadásra. Pontosabban megfogalmazva **a session scope attól a pillanattól fogva tárolja az objektumokat, amikor egy kliens csatlakozik, addig a pillanatig, amíg a session meg nem szűnik.** A session megszűnik, ha valami meghívja a HttpSession objektumon az invalidate() metódust, vagy a session az előre beállított futási időt túl nem lépi. Az adott osztályból mindössze 1 példány lesz az adott HTTP-session-ben.
3. **@ApplicationScoped:** A legtágabb scope. A webalkalmazás teljes időtartama alatt tárolja az objektumokat, és ezen a hatáskörön az összes request és session objektum osztozik. Az application scope-ban definiált bean a webalkalmazás bármely példányának első lekérésekor születik meg, és a webalkalmazás webszerverről való eltávolításakor szűnik meg. Egy ilyen hatókörű managed bean-nek csak 1 példánya létezik a webalkalmazásban.
4. **@ViewScoped: Tovább elérhető, mint a request scope, de nem él olyan soká, mint a session scope. Az itt elhelyezett adatok addig élnek, amíg a felhasználó be nem fejezi az adott nézeten való operálást.** A Bean addig él, amíg a JSF View egy GET-kéréssel kezdődik, és addig tart, amíg a felhasználó beküld valamilyen POST form-ot egy action method-höz ami null-t/void-ot ad vissza, és így visszavisz ugyanarra a view-ra. Amint frissíted

az oldalt, elnavigálsz egy másik oldalra, vagy visszaadásra kerül egy nemnull string navigálás eredményeképpen – ami akár üres string is lehet -, a view scope véget ér.

Más annotációk, amik előjöttek:

1. **@Named:** Az ezzel az annotációval ellátott bean-eket nem mi példányosítjuk, hanem a keretrendszer. A CDI - Contexts and Dependency Injection – kezeli ezeket a bean-eket.
2. **@Inject:** A keretrendszerre bízzuk a példányosítást, akkor fogja létrehozni, ha kell

Spring Boot

A Spring egy nyílt forrású, alkalmazásfejlesztési keretrendszer és inversion of control (IoC) konténer a Java platformhoz. Több modult is tartalmaz, amelyek több szolgáltatást is nyújtanak. Az inversion of control konténere a Spring keretrendszernek lehetővé teszi a Java-objektumok konfigurálását és menedzselését reflexió segítségével. A konténer ezen kívül még felelős az objektumok életciklusainak a kezelésért is: az objektumok létrehozása, az inicializáló metódusaik meghívása, és az objektumok konfigurálása az objektumok összekötésével is az ő dolga.

A Java Spring Boot a Spring keretrendszer egy olyan eszköze, kiterjesztése, ami a webes alkalmazások, microservice-ek Spring-gel történő fejlesztését gyorsabbá és könnyebbé teszi. A Spring Boot a Spring keretrendszernek egy „convention over configuration”/”coding by convention” megoldása, ami egy olyan programtervezési paradigma, amely megkísérli a keretrendszert használó fejlesztő által meghozandó döntések számát csökkenteni anélkül, hogy feleslegesen veszítene a szoftver a rugalmasságából, vagy, hogy megszegné a DRY-alapelveket. A Spring Boot segítségével önálló, „production-grade”, Spring alapú alkalmazásokat lehet készíteni. Előre be van konfigurálva a Spring-et fejlesztő csapat „opinionated view”-ja, azaz véleménye szerint arra vonatkozólag, hogy szerintük mi a legjobb konfiguráció az alkalmazáshoz, illetve a Spring platform és harmadik féltől származó könyvtárak használatához, így a Spring konfigurálása csak minimális erőfeszítést igényel a fejlesztő részéről.

A Spring Boot fő jellemzői a következők:

- Lehetővé teszi önálló Spring-alkalmazások létrehozását
- A Tomcat és a Jetty (webszerver/alkalmazásszerver) közvetlenül beágyazható

- Véleményezett/ajánlott kezdő/starter POM fájlokat biztosít azért, hogy leegyszerűsítse a Maven/Gradle konfigurációját
- Ahol csak lehetséges, automatikusan konfigurálja a Spring-et
- A piacra dobásra való készen állást felmérő szolgáltatásokat nyújt, mint a metrikák, health-check-ek (állapotfelmérések?), és kiterjesztett konfiguráció
- Semmi szükség nincs kódgenerálásra, és nem szükséges az XML-konfiguráció sem

A Spring Boot-hoz elérhető számos „Starter Dependency", amik nagyban megkönnyítik a programozó életét. Ezek nélkül, ha egy nagy Spring-es projektet csinált valaki, akkor a szoftver minden egyes függőségét egyesével kellett hozzáadni – Maven projekt esetén – a pom.xml-hez a helyes verziószámmal és scope-pal. Ezt a munkát spórolják meg a starter dependency-k. Ezek is a pom.xml-hez hozzáadható függőségek, de van egy különlegességük: olyan speciális függőségek, amik aggregálják, összegyűjtik a leggyakrabban használt függőségeket, így nem kell azokat egyesével hozzáadni a pom-hoz, elég csak a starter dependency-t. Előadáson Tanár Úr a Starter Web starter dependency-t használta.

Inversion of Control – IoC: Az Inversion of Control egy szoftverfejlesztési alapelv, ami megfordítja a flow of control-t, azaz a vezérlésfolyamot – „az a sorrend, amelyben egy imperatív program egyes utasításai vagy függvényhívásai végrehajtásra vagy kiértékelésre kerülnek." (https://en.wikipedia.org/wiki/Control_flow) - a hagyományos vezérlésfolyamhoz képest. A hagyományos módja ennek az, hogy a saját kódunk végez hívásokat más, külső könyvtárak felé. Az IoC lehetővé teszi a keretrendszer számára, hogy irányítása alá vegye a programvégrehajtást, és hívásokat eszközöljön a saját kódunk felé. Az IoC lényegében tehát átadja az objektumok, vagy programrészek feletti irányítást egy konténernek vagy keretrendszernek. Ezt az alapelvet általában objektumorientált környezetben használják.

Dependency injection – DI: A Dependency Injection, vagy magyarul Függőség-befecskendezés egy olyan szoftverfejlesztési minta, amelyben egy objektum vagy függvény olyan objektumokat vagy függvényeket kap, amelyektől függ. A lényege, hogy „egy objektum más objektumok függőségeit elégíti ki. A függőséget felhasználó objektum szolgáltatást nyújt, az injekció pedig ennek a függőségnek az átadása a kliens részére. A szolgáltatás a kliens állapotának része. A minta alapkövetelménye a szolgáltatás kliensnek való átadása ahelyett, hogy a szolgáltató objektumot a kliens hozná létre." (forrás:

https://hu.wikipedia.org/wiki/A_f%C3%BCgg%C5%91s%C3%A9g_befecskendez%C3%A9se)

A vezérlés megfordítása (IoC – inversion of control) nevű architekturális minta alkalmazásának egy speciális esete – egy olyan minta, amivel implementálni lehet az IoC-t. **A függőség befecskendezés olyan szoftvertervezési elvek és minták összessége, melyek lazán csatolt kód fejlesztését teszik lehetővé.** A lazán csatoltság kiterjeszthetővé teszi a kódot, a kiterjeszthetőség pedig karbantarthatóvá.

Egy objektumra egy olyan szolgáltatásként tekintünk, melyet más objektumok kliensként használnak. Az objektumok közötti kliens-szolgáltató kapcsolatot függésnek nevezzük. Ez a kapcsolat tranzitív.

Függőség (dependency): egy kliens által igényelt szolgáltatást jelent, mely a feladatának ellátásához szükséges.

Függő (dependent): egy kliens objektum, melynek egy függőségre vagy függőségekre van szüksége a feladatának ellátásához.

Befecskendezés (injection): egy kliens függőségeinek megadását jelenti.

DI konténer (DI container): függőség befecskendezési funkcionalitást nyújtó programkönyvtár.– Az Inversion of Control (IoC) container kifejezést is használják rájuk.

Forrás:
https://arato.inf.unideb.hu/jeszenszky.peter/download/swe/presentations/hu/oo.pdf

A minta szerint, ha egy objektum vagy függvény, ami egy bizonyos szolgáltatást akar használni, nem kell tudnia azt, hogy ezeket a szolgáltatásokat hogyan kell létrehozni. Ehelyett, a függőséget kapó kliens – objektum vagy függvény – egy olyan külső kódtól – a befecskendezőtől – kapja meg a függőségeit, amiről nem tud.

Az IoC-t implementáló keretrendszerek egyik fő része az IoC-konténer. A Spring keretrendszerben ezt a szerepet az ApplicationContext nevű interfész tölti be. Ez a Spring konténer felelős a bean-nek is nevezett objektumok példányosításáért, konfigurálásáért, felépítéséért és az életciklusuk kezeléséért is.

Bean: A Spring-ben minden Java-objektumot bean-nek hívunk, amit az alkalmazás elindításakor a Spring keretrendszer hozott létre.

Az ApplicationContext interfésznek több implementációja is létezik a Spring-ben, pl.: *ClassPathXmlApplicationContext*, *FileSystemXmlApplicationContext, WebApplicationContext.* Manuálisan például így lehet létrehozni egy konténert:

```
ApplicationContext context
  = new ClassPathXmlApplicationContext("applicationContext.xml");
```

Forrás: https://www.baeldung.com/inversion-control-and-dependency-injection-in-spring

A bean-ek összeállításához a konténer konfigurációs metaadatokat használ, amit meg lehet adni egy xml-konfiguráció vagy annotációk formájában is.

A Spring-ben a függőség-befecskendezés történhet konstruktorokon, setter-eken vagy mezőkön keresztül is.

1. **Konstruktor alapú DI:** Ebben az esetben a konténer meg fog hívni egy olyan konstruktort, aminek olyan argumentumai vannak, amik megfelelnek a megadni kívánt függőségekkel:

```
public class SimpleMovieLister {

  // the SimpleMovieLister has a dependency on a MovieFinder
  private MovieFinder movieFinder;

  // a constructor so that the Spring container can 'inject' a MovieFinder
  public SimpleMovieLister(MovieFinder movieFinder) {
      this.movieFinder = movieFinder;
  }

  // business logic that actually 'uses' the injected MovieFinder is omitted...
}
```

Forrás: https://docs.spring.io/spring-framework/docs/3.2.x/spring-framework-reference/html/beans.html

2. **Setter alapú DI:** Itt a konténer először a bean példányosításához meghív egy argumentum nélküli konstruktort, vagy egy argumentum nélküli statikus

```
public class SimpleMovieLister {

  // the SimpleMovieLister has a dependency on the MovieFinder
  private MovieFinder movieFinder;

  // a setter method so that the Spring container can 'inject' a MovieFinder
  public void setMovieFinder(MovieFinder movieFinder) {
      this.movieFinder = movieFinder;
  }

  // business logic that actually 'uses' the injected MovieFinder is omitted...
}
```

„factory” metódust, majd ezután meghívja az osztályunk setter metódusait a függőség befecskendezéshez:

Forrás: https://docs.spring.io/spring-framework/docs/3.2.x/spring-framework-reference/html/beans.html

3. **Mező alapú DI:** Itt a függőségeket befecskendezhetjük az osztály mezőibe úgy, hogy ellátjuk őket az @Autowired annotációval:

```
public class Store {
    @Autowired
    private Item item;
}
```

Forrás: https://www.baeldung.com/inversion-control-and-dependency-injection-in-spring

A Store típusú objektum létrehozásakor, ha nincs az Item bean befecskendezésére alkalmas konstruktor vagy setter metódus, a konténer

reflexió segítségével fogja befecskendezni az Item-et a Store-ba. Ugyanez elérhető XML-konfigurációval is.

Ez azonban nem az ajánlott módja a DI-nek, hiszen:

a) Reflexiókon keresztül történik a függőség befecskendezése, ami költségesebb, mint egy konstruktor vagy setter használata,

b) Final mezőkkel nem lehet használni,

c) Ezzel a módszerrel könnyűnek tűnhet több függőséget is megadni, de így egy osztálynak több felelőssége is lehet – több függőség, több felelősség - , ami megsérti az egyszeres felelősség elvét.

Bean-ek: Ahogy arról már szó esett fentebb, a Spring-ben azokat az objektumokat nevezzük bean-eknek, amik az alkalmazásunk gerincét alkotják, és a Spring IoC konténer kezeli őket. A bean egy olyan objektum, amit a Spring IoC konténer hoz létre, állít össze és kezel.

- **@Bean:** Ahhoz, hogy egy bean-t deklaráljunk, elég egy metódust ellátnunk a @Bean annotációval. Ezzel szétválasztható a bean létrehozása az osztálydeklarációtól. Ez az annotáció azt jelzi, hogy az ezzel megjelölt metódus egy olyan objektumot ad vissza, amit bean-ként kell regisztrálni a Spring application context-ben. Például:

```
package com.tutorialspoint;
import org.springframework.context.annotation.*;

@Configuration
public class HelloWorldConfig {
   @Bean
   public HelloWorld helloWorld(){
      return new HelloWorld();
   }
}
```

Forrás: https://www.tutorialspoint.com/spring/spring_java_based_configuration.htm

Ez a kód ekvivalens az alábbi xml-konfigurációval:

```
<beans>
   <bean id = "helloWorld" class = "com.tutorialspoint.HelloWorld" />
</beans>
```

Forrás: https://www.tutorialspoint.com/spring/spring_java_based_configuration.htm

- **@Configuration:** Az ezzel az annotációval ellátott osztályokat a Spring IoC konténer bean-definíciók forrásaként használhatja.

- **@ComponentScan:** A Spring képes egy csomagot automatikusan átkutatni, és megkeresni benne a bean-eket. A @ComponentScan annotációt a @Configuration annotációval együtt szokták használni, és megmondja, hogy melyik package-ben keressen a Spring bean-eket. Pl.:

```
@Configuration
@ComponentScan(basePackages = "com.baeldung.annotations")
@ComponentScan(basePackageClasses = VehicleFactoryConfig.class)
class VehicleFactoryConfig {}
```

Forrás: https://www.baeldung.com/spring-bean-annotations

Ugyanez megadható XML-konfigurációval is:

```
<context:component-scan base-package="com.baeldung" />
```

Forrás: https://www.baeldung.com/spring-bean-annotations

- **@Component:** Ez egy osztályszintű annotáció. A komponensszkennelés alatt a Spring keretrendszer automatikusan észreveszi a @Component annotációval ellátott osztályokat, pl.:

```
@Component
class CarUtility {
    // ...
}
```

Forrás: https://www.baeldung.com/spring-bean-annotations

Alapból ezen osztályok bean-példányainak ugyanaz a neve, mint az osztálynak, csak kis kezdőbetűvel. Ezen az annotáció opcionális *value* paraméterével változtathatunk, ahol megadhatunk egy nevet.

- **@Repository:** A DAO vagy Repository osztályok általában az adatbázis hozzáférési réteget képviselik egy alkalmazásban, és ezeket el kell látni a @Repository annotációval:

```
@Repository
class VehicleRepository {
    // ...
}
```

Forrás: https://www.baeldung.com/spring-bean-annotations

Ennél az annotációnál engedélyezve van az automatikus persistence exception translation, az perzisztenciával kapcsolatos kivételek „lefordítása”, ami azt jelenti, hogy az osztályon belül dobott alacsony szintű kivételekből magas szintű, a Spring DataAccessException osztályából származó alosztályok példányaivá konvertálja át őket.

- **@Service:** Egy alkalmazás üzleti logikája legtöbbször a szolgáltatás/service rétegben van, és a @Service annotáció való arra, hogy az ebbe a rétegbe tartozó osztályokat megjelöljük:

```
@Service
public class VehicleService {
    // ...
}
```

Forrás: https://www.baeldung.com/spring-bean-annotations

- **@Controller:** Ez egy osztályszintű annotáció, amivel a controller osztályt szokták megjelölni a Spring MVC-ben.

Spring bean életciklus: A Spring bean-eknek van egy életciklusa, amin végigmennek az „életük" során. Ennek sarkalatos pontjai a bean inicializálása, és megsemmisítése. A keretrendszer több módot is biztosít a bean-ek inicializálása után, és megsemmisítése előtt végrehajtandó metódusok megadására:

a) **@PostConstruct és @PreDestroy:** Ezekkel megjelölhetünk olyan metódusokat, amik a bean-ek létrejötte után, és megsemmisítése előtt lefutnak:

```
package com.journaldev.spring.service;

import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;

public class MyService {

	@PostConstruct
	public void init(){
		System.out.println("MyService init method called");
	}

	public MyService(){
		System.out.println("MyService no-args constructor called");
	}

	@PreDestroy
	public void destory(){
		System.out.println("MyService destroy method called");
	}
}
```

Forrás: https://www.digitalocean.com/community/tutorials/spring-bean-life-cycle

Ezek működéséhez úgy kell felkonfigurálni a Spring-alkalmazásunkat, hogy keressen annotációkat.

b) Implementálhatjuk az **InitializingBean** és **DisposableBean** interfészeket – mindkettő egy metódust deklarál, ahol inicializálhatunk/felszabadíthatunk erőforrásokat a bean-ben. A bean inicializálása utáni műveletekhez implementálni kell az InitializingBean interfész **afterPropertiesSet()** metódusát, a bean megsemmisítése előtti műveletek elvégzéséhez pedig a DisposableBean interfész **destroy()** metódusát. Ezt egyszerű használni, de nem

ajánlott, mert szoros csatoltságot fog eredményezni a Spring keretrendszer és a bean implementációnk között.

```
package com.journaldev.spring.service;

import org.springframework.beans.factory.DisposableBean;
import org.springframework.beans.factory.InitializingBean;

import com.journaldev.spring.bean.Employee;

public class EmployeeService implements InitializingBean, DisposableBean{

	private Employee employee;

	public Employee getEmployee() {
		return employee;
	}

	public void setEmployee(Employee employee) {
		this.employee = employee;
	}

	public EmployeeService(){
		System.out.println("EmployeeService no-args constructor called");
	}

	@Override
	public void destroy() throws Exception {
		System.out.println("EmployeeService Closing resources");
	}

	@Override
	public void afterPropertiesSet() throws Exception {
		System.out.println("EmployeeService initializing to dummy value");
		if(employee.getName() == null){
			employee.setName("Pankaj");
		}
	}
}
```

Forrás: https://www.digitalocean.com/community/tutorials/spring-bean-life-cycle

c) A Spring bean konfigurációs állományban megadhatunk értékeket az **init-method** és **destroy-method** attribútumoknak. Ez az ajánlott módja az életciklussal kapcsolatos műveletek megadásának. Ezeknek a metódusoknak nem lehetnek paramétereik, de dobhatnak kivételt.

```java
package com.journaldev.spring.service;

import com.journaldev.spring.bean.Employee;

public class MyEmployeeService{

	private Employee employee;

	public Employee getEmployee() {
		return employee;
	}

	public void setEmployee(Employee employee) {
		this.employee = employee;
	}

	public MyEmployeeService(){
		System.out.println("MyEmployeeService no-args constructor called");
	}

	//pre-destroy method
	public void destroy() throws Exception {
		System.out.println("MyEmployeeService Closing resources");
	}

	//post-init method
	public void init() throws Exception {
		System.out.println("MyEmployeeService initializing to dummy value");
		if(employee.getName() == null){
			employee.setName("Pankaj");
		}
	}
}
```

Forrás: https://www.digitalocean.com/community/tutorials/spring-bean-life-cycle

Egy példa, hogyan konfigurálhatók a bean-ek:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="https://www.springframework.org/schema/beans"
	xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance"
	xsi:schemaLocation="https://www.springframework.org/schema/beans https://www.springframe

<!-- Not initializing employee name variable-->
<bean name="employee" class="com.journaldev.spring.bean.Employee" />

<bean name="employeeService" class="com.journaldev.spring.service.EmployeeService">
	<property name="employee" ref="employee"></property>
</bean>

<bean name="myEmployeeService" class="com.journaldev.spring.service.MyEmployeeService"
		init-method="init" destroy-method="destroy">
	<property name="employee" ref="employee"></property>
</bean>

<!-- initializing CommonAnnotationBeanPostProcessor is same as context:annotation-config -->
<bean class="org.springframework.context.annotation.CommonAnnotationBeanPostProcessor" />
<bean name="myService" class="com.journaldev.spring.service.MyService" />
</beans>
```

Forrás: https://www.digitalocean.com/community/tutorials/spring-bean-life-cycle

Bean scope-ok: A bean scope-ok határozzák meg a bean-ek életciklusát és láthatóságát.

a) **Singleton:** A konténer csak egy példányt hoz létre ebből a bean-ből, és később minden bean-re vonatkozó kérésre ugyanazt az objektumokat adja majd vissza, ami gyorsítótárazva is van. Ez az alapértelmezett scope, ha nincs más scope megadva. Ezt a scope-ot általában stateless bean-eknél kell használni.

```
@Bean
@Scope("singleton")
public Person personSingleton() {
    return new Person();
}
```

Forrás: https://www.baeldung.com/spring-bean-scopes

b) **Prototype:** A konténertől érkező minden kérésre más példányt kerül visszaadásra az ilyen típusú bean-ből. Ezt a scope-ot általában stateful bean-eknél érdemes használni.

```
@Bean
@Scope("prototype")
public Person personPrototype() {
    return new Person();
}
```

Forrás: https://www.baeldung.com/spring-bean-scopes

A további négy scope ún. web aware scope, azaz csak web-aware alkalmazásokban elérhetőek.

c) **Request:** Ez a scope egy HTTP-kéréshez létrehoz egy bean-példányt:

```
@Bean
@Scope(value = WebApplicationContext.SCOPE_REQUEST, proxyMode = ScopedProxyMode.TARGET_CLASS)
public HelloMessageGenerator requestScopedBean() {
    return new HelloMessageGenerator();
}
```

Forrás: https://www.baeldung.com/spring-bean-scopes

Vagy:

```
@Bean
@RequestScope
public HelloMessageGenerator requestScopedBean() {
    return new HelloMessageGenerator();
}
```

Forrás: https://www.baeldung.com/spring-bean-scopes

d) **Session:** Ez a scope egy bean-példányt hoz létre egy HTTP-munkamenethez/session-höz:

```
@Bean
@Bean
@SessionScope
public HelloMessageGenerator sessionScopedBean() {
    return new HelloMessageGenerator();
}
```

Képek forrása: https://www.baeldung.com/spring-bean-scopes

e) **Application:** Ez a scope egy bean-példányt hoz létre a ServletContext életciklusa alatti időtartamra. Hasonló a singleton scope-hoz, de a bean-nek ugyanaz a példánya lesz megosztva az ugyanazon ServletContextben futó, több servlet alapú alkalmazás között, míg a singleton hatáskörű bean-ek csak egy application context-hez tartoznak.

```
@Bean
@Scope(
  value = WebApplicationContext.SCOPE_APPLICATION, proxyMode = ScopedProxyMode.TARGET_CLASS)
public HelloMessageGenerator applicationScopedBean() {
    return new HelloMessageGenerator();
}
```

```
@Bean
@ApplicationScope
public HelloMessageGenerator applicationScopedBean() {
    return new HelloMessageGenerator();
}
```

Képek forrása: https://www.baeldung.com/spring-bean-scopes

f) **WebSocket:** Első elérésnél ezek a bean-ek a websocket munkamenet attribútumokban kerülnek eltárolásra. Ezután a bean-nek ugyanaz a példánya kerül visszaadásra a teljes websocket munkamenet/session alatt, amikor valaki megkísérel hozzáférni:

```
@Bean
@Scope(scopeName = "websocket", proxyMode = ScopedProxyMode.TARGET_CLASS)
public HelloMessageGenerator websocketScopedBean() {
    return new HelloMessageGenerator();
}
```

Forrás: https://www.baeldung.com/spring-bean-scopes

Spring Data JPA

A Spring Data JPA a Spring Data család része. Megkönnyíti a JPA alapú - JPA = Java Persistence API, egy keretrendszer a Java programozási nyelvhez, melynek fő feladata a relációs adatok kezelése a Java Platform Standard és Enterprise Editiont használó alkalmazásokban. – forrás: https://hu.wikipedia.org/wiki/Java_Persistence_API - repository-k/tárolók implementálását.

A Spring Boot-os alkalmazásokban való felhasználáshoz szükséges az alábbi függőségek hozzáadása a projekthez:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter</artifactId>
    <version>2.2.6.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
    <version>2.2.6.RELEASE</version>
</dependency>
```

Forrás: https://www.baeldung.com/the-persistence-layer-with-spring-and-jpa

@Entity: Ez az annotáció azt jelenti, hogy egy osztály hozzátársítható egy adatbázisbeli táblához – egy adatbázisbeli objektumot jelöl.

@Id: Ez az annotáció jelöli az elsődleges kulcsát az entitásnak.

@GeneratedValue: Ezzel az annotációval lehet megadni az elsődleges kulcs generálásának módját, például hogy az adatbázis automatikusan generáljon értéket ennek a mezőnek. Ha ez a stratégia nincs megadva, az AUTO lesz használva alapból:

```java
// @Entity annotation defines that a
// class can be mapped to a table
@Entity
public class Employee {

    // @ID This annotation specifies
    // the primary key of the entity.
    @Id

    // @GeneratedValue This annotation
    // is used to specify the primary
    // key generation strategy to use
    @GeneratedValue(strategy = GenerationType.AUTO)
    private long id;
    private String name;
    private String city;

    public Employee() {
        super();
    }
    public Employee(String name, String city) {
        super();
        this.name = name;
        this.city = city;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getCity() {
        return city;
    }

    public void setCity(String city) {
        this.city = city;
    }

}
```

Forrás: https://www.geeksforgeeks.org/spring-boot-spring-data-jpa/

A Hibernate egy nyílt forráskódú objektum-relációs leképező keretrendszert nyújt a Java-hoz. A 3.2-es és az ennél későbbi verziók a Java Persistence API-hoz is implementációt adnak. (Forrás: https://hu.wikipedia.org/wiki/Java_Persistence_API)

A Spring Boot alapból a Hibernate-et használja a JPA implementáláshoz.

Tételezzük fel, hogy van egy Employee entitásunk, amihez szükséges egy séma, és példa adatok ahhoz, hogy inicializálni lehessen az adatbázisban:

```java
public class Employee {

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

    private String employeeName;
    private String salary;
    private Date createdAt;
    private Date updatedAt;
}
```

Forrás: https://www.javadevjournal.com/spring-boot/loading-initial-data-with-spring-boot/

Amikor futtatjuk a Spring Boot alkalmazásunkat, a keretrendszer létre fog hozni nekünk egy üres táblát, de nem fogja a fenti entitással kitölteni. Az entitásoknak lehet automatikusan sémákat generálni az application.properties konfigurációs állományban a spring.jpa.hibernate.ddl-auto beállításával. Ennek lehetséges értékei:

- **create:** A Hibernate először eldobja a meglévő táblákat, majd létrehozza az újakat
- **update:** Csak akkor frissíti a sémát, ha szükséges. Például, ha egy új mező lett hozzáadva egy entitáshoz, akkor egyszerűen meg fogja változtatni a táblát, és hozzáad egy új oszlopot a meglévő adatok babrálása nélkül
- **create-drop:** Létrehoz egy sémát indításkor, és megsemmisíti a sémát a context bezárásakor/megszűnésekor – a SessionFactory „bezárásakor”. Hasznos egységteszteknél.
- **validate:** Csak azt ellenőrzi le, hogy a séma passzol-e az entitásokhoz. Ha nem, akkor az alkalmazás elindítása sikertelen lesz. Az adatbázist nem módosítja.
- **none:** Nem történik adatbázisséma inicializáció.

DataSource: A DataSource a fizikai adatbázisokhoz való kapcsolatokat kialakító „kapcsolatgyár”, a DriverManager egy alternatívája. Egy DataSource egy url-t, és egy felhasználónév-jelszó párost használ az adatbázis-kapcsolat létrehozásához. Kétféleképpen is be lehet konfigurálni: Java-kóddal, vagy az application.properties állományban is. Most az utóbbit nézzük meg. Ennek a módszernek az előnye, hogy szétválasztja a konfigurálást a kódtól.

Az application.properties fájlban a DataSource beállítására a spring.datasource.* alakú tulajdonságok szolgálnak:

application.properties

```
# H2 DB
spring.datasource.url=jdbc:h2:file:C:/temp/test
spring.datasource.username=sa
spring.datasource.password=
spring.datasource.driverClassName=org.h2.Driver
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect

# MySQL
spring.datasource.url=jdbc:mysql://localhost:3306/test
spring.datasource.username=dbuser
spring.datasource.password=dbpass
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
spring.jpa.database-platform=org.hibernate.dialect.MySQL5InnoDBDialect

# Oracle
spring.datasource.url=jdbc:oracle:thin:@localhost:1521:orcl
spring.datasource.username=dbuser
spring.datasource.password=dbpass
spring.datasource.driver-class-name=oracle.jdbc.OracleDriver
spring.jpa.database-platform=org.hibernate.dialect.Oracle10gDialect

# SQL Server
spring.datasource.url=jdbc:sqlserver://localhost;databaseName=springbootdb
spring.datasource.username=dbuser
spring.datasource.password=dbpass
spring.datasource.driverClassName=com.microsoft.sqlserver.jdbc.SQLServerDriver
spring.jpa.hibernate.dialect=org.hibernate.dialect.SQLServer2012Dialect
```

Forrás:
https://howtodoinjava.com/spring-boot2/datasource-configuration/

1:1, 1:N, N:1, N:M kapcsolatok: A Hibernate-nél is van lehetőség az ilyen kapcsolatok kezelésére.

a) **1:1 kapcsolat:** Két entitás közötti kapcsolat, ahogy az egyik tábla 1 sorához a másik táblának 1 sora tartozik. Ezt a @OneToOne annotációval lehet elérni:

Person.java

```
1.  @Getter
2.  @Setter
3.  @Builder
4.  @Entity
5.  @Table(name = "persons")
6.  public class Person {
7.
8.      @Id
9.      @GeneratedValue(strategy = GenerationType.IDENTITY)
10.     private Long id;
11.     private String name;
12.     private String email;
13.     private String password;
14.     @OneToOne(cascade = CascadeType.ALL)
15.     @JoinColumn(name = "address_id")
16.     private Address address;
17. }
```

Forrás: https://springframework.guru/one-to-one-relationship-in-jpa/

A @Table(name=”persons”) annotáció azt mondja meg, hogy a Person entitás példányát a „persons” nevű táblába mentse el a rendszer. A @OneToOne annotáció mondja meg, hogy két entitás között 1:1 kapcsolat áll fenn. A cascade paramétere a szülőentitás frissítésekor és törlésekor megtörténő kaszkádolási műveletek viselkedését írja le. Ezen kívül meg lehet még adni a

fetch-elés módját is. Ezt a „fetch=FetchType.LAZY/EAGER”-rel adhatjuk meg. Ez azt mondja meg a Hibernate-nek, hogy mikor szerezze meg a kapcsolódó entitásokat az adatbázisból. 2 fajtája van:

a) **fetch=FetchType.EAGER:** Ez a „Fetcheld, és rendelkezésedre fog állni, ha majd kelleni fog” hozzáállást képviseli. Ennél az opciónál a Hibernate a gyökérentitás kiválasztásánál a kapcsolat minden elemét megszerzi. Az 1:1 és N:1 kapcsolatoknál ez az alapértelmezett. SQL-lel valahogy így nézne ki: SELECT * FROM EMPL … JOIN DEPT. Például, ha van egy N:1 kapcsolat az OrderItem és a Product között, és ha fetch-elünk egy OrderItem-et, a hozzá tartozó Product entitást is lekéri a Hibernate.

b) **fetch=FetchType.LAZY:** Ez a „Fetcheld, amikor szükséged van rá” hozzáállást képviseli. A kapcsolat használatakor a Hibernate csak a kapcsolódó entitásokat kéri le az adatbázisból. SQL-lel valahogy így nézne ki: SELECT * FROM EMPL.

A @JoinColumn annotáció a külső kulcshoz tartozó oszlopot elnevezi address_id-nak.

b) **1:N kapcsolat:** Egy táblázat 1 sorához egy másik táblázat több sora is tartozhat. Ezt a @OneToMany annotációval jelezhetjük:

```
public class Cart {

    //...

    @OneToMany(mappedBy="cart")
    private Set<Item> items;

    //...
}
```

Forrás: https://www.baeldung.com/hibernate-one-to-many

c) **N:1 kapcsolat:** Egy táblázat több sorához egy másik tábla 1 sora tartozhat. Ezt a @ManyToOne annotációval jelezhetjük:

```java
@Entity(name = "PostComment")
@Table(name = "post_comment")
public class PostComment {

    @Id
    @GeneratedValue
    private Long id;

    private String review;

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "post_id")
    private Post post;

    //Getters and setters omitted for brevity
}
```

Forrás: https://vladmihalcea.com/manytoone-jpa-hibernate/

d) **N:M kapcsolat:** Egy táblázat több sorához egy másik tábla több sora tartozhat. Ezt a @ManyToMany annotációval jelezhetjük:

```java
@Entity
class Student {

    @Id
    Long id;

    @ManyToMany
    Set<Course> likedCourses;

    // additional properties
    // standard constructors, getters, and setters
}

@Entity
class Course {

    @Id
    Long id;

    @ManyToMany
    Set<Student> likes;

    // additional properties
    // standard constructors, getters, and setters
}
```

Forrás: https://baeldung.com/jpa-many-to-many

@Transactional: Ez az annotáció használható metódusokon és osztályokon is. Az ezzel az annotációval megjelölt dolgok műveleteit tranzakciókba kell raknia a rendszernek, így az adatbázis nem kerül inkonzisztens állapotba.

@Repository: A Spring Data-ban a tárolók absztrakciójának a központi interfésze a Repository interfész. Ezt az interfészt terjeszti ki a CrudRepository interfész, ami CRUD-műveletekhez – Create, Read, Update, Delete – ad támogatást a kezelés alatt álló entitásosztályokhoz:

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

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

    T findOne(ID primaryKey);                            ❷

    Iterable<T> findAll();                               ❸

    Long count();                                        ❹

    void delete(T entity);                               ❺

    boolean exists(ID primaryKey);                       ❻

    // … more functionality omitted.
}
```

Forrás: https://docs.spring.io/spring-data/data-commons/docs/1.6.1.RELEASE/reference/html/repositories.html

A CrudRepository interfészt kiterjeszti a PagingAndSortingRepository interfész, ami az entitásokhoz való „lapszámozásos”/oldalakba szedett (paginated) hozzáférést ad:

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

  Iterable<T> findAll(Sort sort);

  Page<T> findAll(Pageable pageable);
}
```

Forrás: https://docs.spring.io/spring-data/data-commons/docs/1.6.1.RELEASE/reference/html/repositories.html

Ha saját repository-t akarunk definiálni, akkor egy olyan interfészt kell definiálunk, ami kiterjeszti a Repository vagy a CrudRepository interfészt:

```
interface MyBaseRepository<T, ID extends Serializable> extends Repository<T, ID> {
  T findOne(ID id);
  T save(T entity);
}

interface UserRepository extends MyBaseRepository<User, Long> {

  User findByEmailAddress(EmailAddress emailAddress);
}
```

Forrás: https://docs.spring.io/spring-data/data-commons/docs/1.6.1.RELEASE/reference/html/repositories.html

Ők fognak felelni az SQL-utasítások kiadásáért: a függvénynevek át lesznek „alakítva” SQL-lekérdezésekké, pl: findAll() -> SELECT * FROM EMPL.

Java EE Security

A Java EE 8 tartalmaz egy Security API-t, ami hordozható, „beépülő” (plug-in) interfészeket határoz meg a hitelesítéshez és az identitástárolókhoz, valamint egy injektálható típusú SecurityContext interfészt, ami a programozott biztonsághoz ad egy hozzáférési pontot/API-t.

JASPIC: Java Authentication Service Provider Interface for Containers. A Java EE 6-ba került bele. A JSR 196 – Java Specification Request – egy szabványos szolgáltatói interfészt határoz meg, és szabványosítja azt, hogy egy hitelesítési modul hogyan integrálódjon egy Java EE konténerbe.

JSR-375: Egy könnyen használható, egységes API-t ad a hitelesítés és meghatalmazás (authentication and authorization) megvalósításához.

Célok:

a) modernizálás

- Context Dependency Injection (CDI)
- Expression Language (EL)

b) fejlesztőbaráttá tétel

A referencia implementációja a Soteria.

Tulajdonságai:

1. **HTTP-s hitelesítés**
    a. **Basic HTTP hitelesítés**: Ehhez szükséges a @BasicAuthenticationMechanismDefinition annotáció használata:

```
@BasicAuthenticationMechanismDefinition(
  realmName = "userRealm")
@ApplicationScoped
public class AppConfig{}
```

Forrás: https://www.baeldung.com/java-ee-8-security

Ilyenkor a Servlet konténer megkeresi, és példányosítja az HttpAuthenticationMechanism interfész megadott implementációját. Ha a konténer kap egy „unauthorized” kérést, akkor megkéri a klienst egy „WWW-Authenticate” fejlécmező elküldésével, hogy adja meg a hitelesítési adatait:

```
WWW-Authenticate: Basic realm="userRealm"
```

Forrás: https://www.baeldung.com/java-ee-8-security

Erre válaszul a kliens megfogja a felhasználónevét és a jelszavát, egymás mellé rakja őket, közéjük pakol egy :-ot, Base64-gyel titkosítja, és elküldi a konténernek az „Authorization” fejlécmezőben:

```
//user=baeldung, password=baeldung
Authorization: Basic YmFlbGR1bmc6YmFlbGR1bmc=
```

Forrás: https://www.baeldung.com/java-ee-8-security

b. **Form alapú hitelesítés**: Ezt a @FormAuthenticationMechanismDefinition annotációval lehet elérni. Az annotációval megadhatjuk a bejelentkezési, és hibaoldalt is:

```
@FormAuthenticationMechanismDefinition(
  loginToContinue = @LoginToContinue(
    loginPage = "/login.html",
    errorPage = "/login-error.html"))
@ApplicationScoped
public class AppConfig{}
```

Forrás: https://www.baeldung.com/java-ee-8-security

A bejelentkezési oldal „meghívásának” eredményeképpen a szerver el fogja küldeni a form-ot a kliensnek:

```
<form action="j_security_check" method="post">
    <input name="j_username" type="text"/>
    <input name="j_password" type="password"/>
    <input type="submit">
</form>
```

Forrás: https://www.baeldung.com/java-ee-8-security

A kliensnek ezután el kellene küldenie a form-ot egy, a konténer által biztosított, előre megadott hitelesítési folyamatnak.

c. **<u>Saját form alapú hitelesítés</u>**: Ezt a @CustomFormAuthenticationMechanismDefinition annotációval érhetjük el:

```
@CustomFormAuthenticationMechanismDefinition(
  loginToContinue = @LoginToContinue(loginPage = "/login.xhtml"))
@ApplicationScoped
public class AppConfig {
}
```

Forrás: https://www.baeldung.com/java-ee-8-security

De a sima form alapú hitelesítéssel szemben itt egy saját bejelentkezési oldalt adunk meg, és a SecurityContext.authenticate() metódust hívjuk meg, mint hitelesítő folyamatot:

```
@Named
@RequestScoped
public class LoginBean {

    @Inject
    private SecurityContext securityContext;

    @NotNull private String username;

    @NotNull private String password;

    public void login() {
        Credential credential = new UsernamePasswordCredential(
          username, new Password(password));
        AuthenticationStatus status = securityContext
          .authenticate(
            getHttpRequestFromFacesContext(),
            getHttpResponseFromFacesContext(),
            withParams().credential(credential));
        // ...
    }

    // ...
}
```

Forrás: https://www.baeldung.com/java-ee-8-security

A saját bejelentkezési oldalunk „meghívása” hatására a kliens elküldi a megkapott form-ot a loginBean login() metódusának (fenti példa):

```
//...
<input type="submit" value="Login" jsf:action="#{loginBean.login}"/>
```

Forrás: https://www.baeldung.com/java-ee-8-security

2. **Identity Store:** Ezt a bejelentkezési adatok validálására, csoportok és csoportjogok kezelésére, más szóval hitelesítésre, engedélyezésre (authentication, authorization) vagy akár mindkettőre is használják. Ennek a központi elemei az IdentityStore interfész, és a CredentialValidationResult objektum. Egy alkalmazás vagy saját implementációt csinál az IdentityStore interfészhez, vagy a 2 beépített implementáció egyikét használja: a Database-t, vagy az LDAP-ot.

    - A Database-es implementációnak inicializáláskor át kell adni a konfigurációs adatokat a @DatabaseIdentityStoreDefinition annotációba:

```
@DatabaseIdentityStoreDefinition(
  dataSourceLookup = "java:comp/env/jdbc/securityDS",
  callerQuery = "select password from users where username = ?",
  groupsQuery = "select GROUPNAME from groups where username = ?",
  priority=30)
@ApplicationScoped
public class AppConfig {
}
```

      Forrás: https://www.baeldung.com/java-ee-8-security

    - Az LDAP-os IdentityStore implementációnak is át kell adni inicializáláskor konfigurációs adatokat az @LdapIdentityStoreDefinition annotációban:

```
@LdapIdentityStoreDefinition(
  url = "ldap://localhost:10389",
  callerBaseDn = "ou=caller,dc=baeldung,dc=com",
  groupSearchBase = "ou=group,dc=baeldung,dc=com",
  groupSearchFilter = "(&(member=%s)(objectClass=groupOfNames))")
@ApplicationScoped
public class AppConfig {
}
```

      Forrás: https://www.baeldung.com/java-ee-8-security

    - Lehet saját IdentityStore implementációt is megadni. Ebben az interfészben – IdentityStore – alapból 4 metódus van:

```
default CredentialValidationResult validate(
  Credential credential)
default Set<String> getCallerGroups(
  CredentialValidationResult validationResult)
default int priority()
default Set<ValidationType> validationTypes()
```

Forrás: https://www.baeldung.com/java-ee-8-security

Egy példa implementáció:

```
@ApplicationScoped
public class InMemoryIdentityStore implements IdentityStore {
    // init from a file or harcoded
    private Map<String, UserDetails> users = new HashMap<>();

    @Override
    public int priority() {
        return 70;
    }

    @Override
    public Set<ValidationType> validationTypes() {
        return EnumSet.of(ValidationType.VALIDATE);
    }

    public CredentialValidationResult validate(
      UsernamePasswordCredential credential) {

        UserDetails user = users.get(credential.getCaller());
        if (credential.compareTo(user.getLogin(), user.getPassword())) {
            return new CredentialValidationResult(user.getLogin());
        }
        return INVALID_RESULT;
    }
}
```

Forrás: https://www.baeldung.com/java-ee-8-security

3. **Security Context API**: A SecurityContext interfészen keresztül hozzáférési pontot ad a programozott biztonsághoz. Futásidőben meg kell adni a SecurityContext-nek egy alapértelmezett implementációját egy CDI bean formájában, amit befecskendezéssel adunk meg:

```
@Inject
SecurityContext securityContext;
```

Forrás: https://www.baeldung.com/java-ee-8-security

Az interfész getCallerPrincipal(), isCallerInRole(String role) és getPrincipalsByType(Class<T> type) metódusai arra szolgálnak, hogy

megállapítható legyen velük egy adott erőforrás meghívója, felhasználója. A hasAccessToWebResource() metódus megmondja, hogy egy felhasználó meg tud-e hívni egy adott HTTP-metódust egy Servlet-en. Az authenticate() metódussal meg programkódból tudjuk megkezdeni, kezdeményezni a hitelesítési folyamatot.

A Java EE 8-ból Jakarta EE 8 lett, és a javax.* alakú importokból jakarta.* alakú importok lettek.

Webszolgáltatások

„**A webszolgáltatás (angolul webservice) alkalmazások közötti adatcserére szolgáló protokollok és szabványok gyűjteménye. Különböző programnyelveken írt és különböző platformokon futó szoftveralkalmazások számítógép-hálózatokon (mint az internet) keresztül történő adatcserére használják a webszolgáltatásokat**, az egy gépes folyamatközi kommunikációhoz (IPC) hasonlóan. Ezen interoperabilitás (például Java és Python, illetve Windows és Linux között) a nyílt szabványok használatának eredménye.” (forrás: https://hu.wikipedia.org/wiki/Webszolg%C3%A1ltat%C3%A1s)

Felhasznált szabványok:

1. **XML**: Minden kicserélendő adat XML címkékkel van formázva. Ezen kódolás SOAP vagy XML-RPC formában hajtható végre. Az ipari szabványok (biztonság, együttműködés…) a SOAP-ra alapulnak.
2. **Gyakori protokollok**: az XML adat mozgatása az alkalmazások között az olyan gyakran alkalmazott protokollokkal történik, mint a HTTP, az FTP, az SMTP és az XMPP.
3. **WSDL**: a webszolgáltatás nyilvános felülete a Webszolgáltatás leíró nyelvvel (Web Services Description Language, WSDL) van leírva. Ez egy XML alapú leírása a webszolgáltatásokkal történő kommunikációnak.
4. **UDDI**: **A webszolgáltatásokat ezen protokoll segítségével teszik elérhetővé.** Lehetővé teszi, hogy információt keressünk webszolgáltatásokról, így segítve a döntést, hogy felhasználjuk-e őket.
5. **WS-Security**: A Web Services Security (Webszolgáltatás biztonság) protokollt az OASIS szabványként fogadta el. Lehetővé teszi a felek azonosítását, valamint az üzenetek titkosítását.

SOA = Services-Oriented Architecture. „A szolgáltatásorientált architektúra (angolul Service-Oriented Architecture, röviden SOA) egy programozási módszer, aminek lényege, hogy az egyes szolgáltatások egy hálózatban helyezkednek el, és

egy protokoll által meghatározott módon kommunikálnak. Alapvető elvei függetlenek gyártóktól, termékektől, terjesztőktől és technológiáktól. A szolgáltatások különálló egységek, amik távolról is elérhetők, egymástól függetlenül elérhetők, használhatók, frissíthetők, újra kombinálhatók a folyamatok folytonos változásának, megújulásának megfelelően." (Forrás: https://hu.wikipedia.org/wiki/Szolg%C3%A1ltat%C3%A1sorient%C3%A1lt_architekt%C3%BAra)

Egy szolgáltatásnak a következőket kell teljesítenie:

- Logikailag reprezentál egy üzleti aktivitást előre meghatározott kimenettel.
- Önálló, független egység (self-contained).
- Feketedoboz a kliensei számára, azaz a kliensnek nem kell ismernie a szolgáltatás belső működését.
- Tartalmazhat több részszolgáltatást.

A részrendszerek az alábbi csoportokba oszthatók szét:

1. **Szolgáltató: Létrehozza a webszolgáltatást, és információt közvetít a névszolgáltatónak.** Azt is leírja, hogy milyen szolgáltatást nyújt, mi a fontosabb: a könnyű elérhetőség vagy a biztonság, mennyiért lehet igénybe venni, és további információkat is adhat, amik bekerülnek a névszolgáltatóhoz.
2. **Névszolgáltató: Egy névszolgáltató feladata az, hogy információt nyújtson az elérhető szolgáltatásokról.** Megvalósítójuk döntésétől függően lehetnek nyilvánosak vagy privátok, ez utóbbiakat nem mindenki veheti igénybe. Az UDDI egy korai kísérlet volt a webszolgáltatások felderítésére.
3. **Szolgáltatásigénylő**: **Az általa igényelt szolgáltatás elérhetőségét névszolgáltatónál keresi, majd a kellő információk birtokában hozzákapcsolódik a szolgáltatóhoz, és meghívja szolgáltatásait.** Egy résztvevő lehet egyszer szolgáltató, másszor szolgáltatásigénylő. A szolgáltatók is névszolgáltatónál keresik a számukra fontos adatokat. (forrás: https://hu.wikipedia.org/wiki/Szolg%C3%A1ltat%C3%A1sorient%C3%A1lt_architekt%C3%BAra)

A SOA-t egy szabványos felszínen, a webszolgáltatás platformon valósítják meg. Ezt a Web szolgáltatás szabványt egyformán támogatja a .NET és a Java. A SOA által használt szolgáltatásorientált környezet legfontosabb szabványai a következők:

- SOAP, egy XML (WSDL) alapú kiterjesztett üzenetformátum (boríték).
- WSDL, webszolgáltatás leíró nyelv, (Web Services Description Language)
- UDDI, általános kereső és integrációs leírásokat tároló eszköz (Universal Description, Discovery and Integration).
- RESTful HTTP, a REST saját megkötés alapú architekturális stílusával

SOAP: „A számítástechnikában **a SOAP (Simple Object Access Protocol) egy protokoll, amelyet webszolgáltatások hálózaton keresztüli kommunikációjához terveztek. Az üzenetek XML-alapúak és általában más, alkalmazás rétegbeli protokollokra támaszkodva továbbítódnak**, például Távoli Eljáráshívás (Remote Procedure Call, RPC) vagy a Hypertext Transport Protocoll (HTTP) protokollokra.” (forrás: https://hu.wikipedia.org/wiki/SOAP)

Egy példa arra, hogy működnek a SOAP eljárások: egy SOAP üzenetet küldhetünk egy webszolgáltatással rendelkező weboldalnak, pl. egy ingatlanügynökségnek, melyben megadjuk a keresési paramétereket. Az oldal egy XML dokumentumot küld vissza a kért adatokkal, pl. az ingatlanok árai, helye, tulajdonságai. Mivel ez az adat a gépek számára könnyen feldolgozható formátumban van, így egyszerűen integrálható egy weboldalba vagy alkalmazásba.

WSDL: „A Webszolgáltatás leíró nyelv (angolul Web Services Description Language, röviden WSDL) webszolgáltatások leírására szolgáló XML formátum. A WSDL a webszolgáltatás nyilvános felületét írja le, beleértve a használható üzenetek formátumát. A támogatott műveletek és üzenetek először absztrakt módon vannak definiálva, majd ezek a definíciók vannak hozzákötve a tényleges hálózati protokollhoz és üzenetformátumhoz. A WSDL-t általában SOAP-pal XSD-vel együtt használják, hogy webszolgáltatást nyújtsanak az interneten. Egy webszolgáltatáshoz kapcsolódó kliens-program (általában a szolgáltatás tényleges használatától függetlenül) le tudja kérni WSDL-t, hogy feltérképezze a rendelkezésre álló funkciókat a szerveren.” (forrás: https://hu.wikipedia.org/wiki/Webszolg%C3%A1ltat%C3%A1s-le%C3%ADr%C3%B3_nyelv)

UDDI: Az UDDI a Universal Description, Discovery, and Integration, azaz univerzális leírás, felfedezés és integrálás rövidítése – egy platformfüggetlen, XML-alapú nyilvántartó rendszer (regiszter), mely lehetőséget biztosít a vállalatok számára, hogy bekerüljenek egy internetes listába és közzétegyék webszolgáltatásaikat. A rendszer segítségével kereshetünk a publikált webszolgáltatások között és további információkat tudhatunk meg azokról. Ezek

között szerepel az is, hogy az adott szolgáltatást hogyan lehet használni. (forrás: https://hu.wikipedia.org/wiki/UDDI)

A rendszerben található információk három fő csoportba sorolhatóak:

- Fehér oldalak (White Pages) – a szolgáltatást nyújtó vállalat neve, leírása, elérhetősége
- Sárga oldalak (Yellow Pages) – a vállalatok vagy szolgáltatások rendszerezése ágazat alapján (mint Magyarországon a TEÁOR)
- Zöld oldalak (Green Pages) – információk a nyújtott szolgáltatás interfészéről (cím, paraméterek), vagy hivatkozás az interfész leírására

Az UDDI csomópontok olyan szerverek, amelyek támogatják az UDDI szabványt és egy UDDI regiszterhez tartoznak. Egy UDDI regiszter egy vagy több csomópontból áll. A szolgáltatás nyújtója WSDL formátumban jegyzi be a szolgáltatást a regiszterbe. A felhasználó SOAP segítségével tud csatlakozni hozzá.

RESTful HTTP: A REST (Representational State Transfer) egy szoftverarchitektúra típus, elosztott kapcsolat (loose coupling), nagy, internet alapú rendszerek számára, amilyen például a világháló. Azokat a rendszereket, amelyek eleget tesznek a REST megszorításainak, "RESTful"-nak nevezik. A legnagyobb olyan rendszer, amely eleget tesz a REST szoftverarchitektúra típus követelményeinek a világháló. A REST szemlélteti a világháló architektúráját azzal, hogy leírja és megköti a világháló négy komponensének (kiszolgálók, átjárók, proxyk és kliensek) magas szintű kölcsönhatásait.

Egy REST típusú architektúra kliensekből és szerverekből áll. A kliensek kéréseket indítanak a szerverek felé; a szerverek kéréseket dolgoznak fel és a megfelelő választ küldik vissza. A kérések és a válaszok erőforrás-reprezentációk szállítása köré épülnek. Az erőforrás lényegében bármilyen koherens és értelmesen címezhető koncepció lehet. Egy erőforrás-reprezentáció általában egy dokumentum, mely rögzíti az erőforrás jelenlegi vagy kívánt állapotát.

Bármely adott pillanatban egy kliens vagy állapotok közötti átmenetben van, vagy "nyugalmi" állapotban. A nyugalmi állapotban lévő kliens képes interakcióra a felhasználójával, de nem hoz létre terhelést és nem fogyaszt tárolót a szervereken vagy a hálózaton.

Ha a kliens készen áll az átmenetre egy új állapotba, akkor elkezdi küldeni a kéréseit a szerverekhez. Míg legalább egy olyan kérés van, amelyre nem érkezett válasz, a kliens átmeneti állapotban marad. Egyes erőforrás-reprezentációk

hivatkozásokat tartalmaznak további erőforrásokra, amelyeket a kliens felhasználhat új állapotba történő átmenetkor.

A HTTP nagyon gazdag szókinccsel rendelkezik igék (vagy "metódusok"), URI-k, média típusok, kérés- és feleletkódok stb. szempontjából. Egy REST alkalmazás a HTTP protokoll meglévő tulajdonságait használja, és így lehetővé teszi a proxyknak és az átjáróknak, hogy együttműködjenek az alkalmazással (például gyorsítótárazás vagy biztonsági funkciók formájában). (Forrás: https://hu.wikipedia.org/wiki/REST)

Egy REST architektúra a következő hat megszorításnak kell megfeleljen, miközben az egyes komponensek implementációit hagyja szabadon tervezni – nem soroljuk most fel mindet:

- Közös interfész
    - Minden leírható erőforrásként
    - URI: Uniform Resource Identifier
    - HTTP metódusok az erőforrások kezelésére
        - GET – letöltés
        - POST – feltöltés
        - PUT – módosítás
        - DELETE – törlés
- Állapotmentesség
    - A szerver nem tárol semmilyen információt a kliens aktuális állapotáról
    - A kliensnek kell tárolnia mindent, ami a szolgáltatás állapotával kapcsolatos
- Gyorsítótárazhatóság
    - A kliens ezen a módon tárolja az információkat (pl. egy webshopban a kosár tartalmát)
- Rétegelt felépítés
    - A felek különböző szinteken kommunikálnak. Pl. a kommunikáció során Proxy szervereket használunk az adatok betöltésére egy alsóbb rétegben, míg magasabb szinten megtörténik a cachelés.
- Igényelt kód / Code on demand

- Nem szükséges a teljes kód letöltése, csak akkor ha éppen a kliensnek szüksége is van rá.

Forrás:
https://arato.inf.unideb.hu/kocsis.gergely/soft_dev/hu/lab11/Webszolg%C3%A1ltat%C3%A1sok.pptx

Spring Security

A Spring Security egy hatékony, és nagymértékben testreszabható hitelesítési (authentication) és „hozzáférés-felügyeleti/hozzáférési jogosultság ellenőrzését lehetővé tevő" (access control) keretrendszer. Ez a de facto szabványa a Spring alapú alkalmazások biztonságossá tételéhez.

A Java alkalmazásoknak a Spring Security lehetővé teszi a felhasználók hitelesítését – authentication - , és az authorizációt (az alkalmazásnak a felhasználó engedélyével történő felhatalmazása bizonyos műveletek végrehajtására, vagy bizonyos adatokhoz való hozzáférésre) – authorization.

Tulajdonságai:

- Átfogó és bővíthető támogatás a hitelesítéshez és az engedélyezéshez is (authentication and authorization)
- Védelem az olyan támadások ellen, mint a session fixation, clickjacking, cross site request forgery, stb.
- Servlet API integráció
- igény szerinti integráció a Spring Web MVC-vel
- …

Forrás: https://spring.io/projects/spring-security#overview

A Spring Security alkotóelemei:

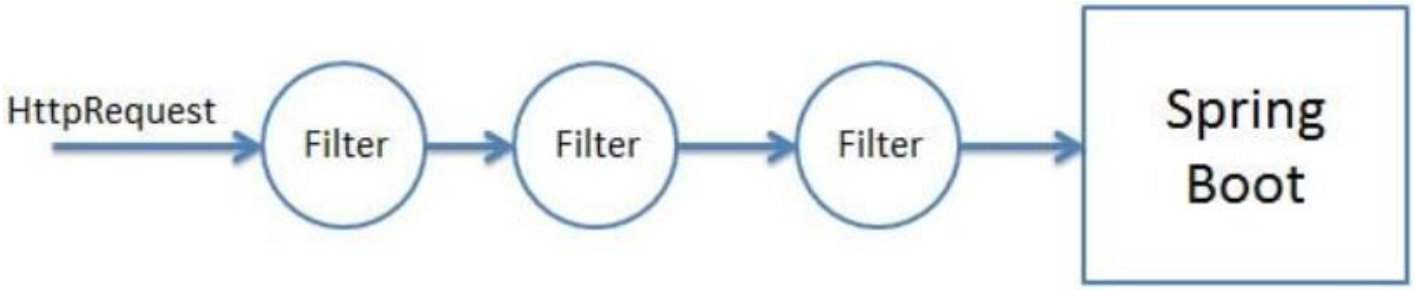
- **Servlet filterek**: Mielőtt belemennénk, hogy mik is azok a filterek, nézzük meg, mi is az az **Interceptor**! **Az Interceptor** a Spring-ben egy olyan osztály, ami **implementálja a HandlerInterceptor interfészt**, és elkapja a HTTP-kéréseket, mielőtt azok elérnék a controller-t. A filter-ekre/szűrőkre úgy is gondolhatunk, mint egy speciális Interceptor-ra.

  Amikor egy kérés beérkezik a Tomcat-hez, vagy bármelyik más webszerverhez, de még nem érte el a servlet-et, lefut egy szűrőmetódus. Ez lehetővé teszi a szűrőknek, hogy blokkolják a kérést, így azok el sem érik a

servlet-et. Innen ered a nevük is. Rengeteg Filter interfész van a Spring Security-ben, amit implementálhatunk egy saját filterosztály létrehozásához. Ezután pedig megmondhatjuk a Spring-nek, hogy adja hozzá ezeket a filter-eket a filter chain-hez, azaz a szűrőlánchoz.

Forrás: https://auth0.com/blog/spring-security-overview/

Mielőtt a HTTP-kérések elérnék a servlet-eket, például a DispatcherServlet-et, először elfogja ezt szűrőkből álló lánc, az ún. filter chain:



Forrás: https://www.javainuse.com/webseries/spring-security-jwt/chap3

Minden bejövő kérés ezeken a szűrőkön fog keresztülmenni, és itt történik meg a kérések hitelesítése – authentication – és a felhatalmazás is – authorization - , ami megakadályozza a hitelesítés nélküli és kártékony kérések beérkezését, továbbhaladását.

A bejövő kérés fajtájától függően többféle filter létezik:

- **BasicAuthenticationFilter**: A bejövő HTTP-kérésben megpróbálja megkeresni a Basic Authentication-ös fejlécet, és az abban talált felhasználónév-jelszó párossal hitelesíteni a felhasználót.
- **UsernamePasswordAuthenticationFilter**: Megpróbál felhasználónevet/jelszót tartalmazó kérésparamétert/POST üzenettörzset keresni, és az abban talált adatokkal hitelesíteni a felhasználót.
- **DefaultLoginPageGeneratingFilter**: Létrehoz egy bejelentkezési oldalt, hacsak ezt a funkciót külön nem kapcsoljuk ki. Emiatt a szűrő miatt kapunk egy alap bejelentkező oldalt, amikor engedélyezzük a Spring Security.
- **DefaultLogoutPageGeneratingFilter**: Létrehoz neked egy kijelentkező oldalt, hacsak nem kapcsolod ki ezt a funkciót.
- **FilterSecurityInterceptor**: Ez végzi az authorization-t, azaz a meghatalmazást.

Igazából ezek, és más szűrők teszik ki a Spring Security nagyját, ők végzik el a piszkos munkát.

- **Matcher-ek**: A segítségükkel megmondhatjuk, milyen URL-eket kell szűrni. Reguláris kifejezéshez hasonló módon történik ezek megadása:

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests()
            .antMatchers(HttpMethod.GET, "/public/**").permitAll()
            .anyRequest().authenticated()
            .and()
            .addFilter(new CustomFilter(authenticationManager())
            .addFilter(new JWTAuthorizationFilter(authenticationManager()))
}
```

Forrás: https://auth0.com/blog/spring-security-overview/

Ez a kódrészlet megengedi minden „/public/” kezdetű URL-re irányuló GET-kérésnek, hogy átmenjen a filtereken. Minden más URL-re érvényesek lesznek a szűrők.

- **User Roles – Role Based Authorization**: A Spring Security-ban a felhasználóknak ki lehet osztani szerepköröket, és ezen szerepek alapján el lehet végezni a szűrést. Ez lehetővé teszi, hogy pl. egy oldalt csak adminok érjenek el. Ezt is a Matcher-ek segítségével érhetjük el:

```
@Override
    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests()
            .antMatchers("/").permitAll
            .antMatchers("/new-blog-post").hasAnyAuthority("ADMIN", "AUTH0 EMPLOYEE", "GUEST_WRITE
            .antMatchers("/edit/**").hasAnyAuthority("ADMIN", "EDITOR")
            .antMatchers("/delete/**").hasAuthority("ADMIN")
            .and()
            .formLogin().permitAll()
            .and()
            .logout().permitAll();
    }
```

Forrás: https://auth0.com/blog/spring-security-overview/

Hogyan konfiguráljuk be a Spring Security-t?

Kell egy osztály, ami:

- El van látva az @EnableWebSecurity annotációval
- Kiterjeszti a WebSecurityConfigurerAdapter osztályt. Az ebben lévő metódusokkal megmondhatod, hogy az alkalmazásodnak milyen URI-kat kell védenie, vagy hogy milyen exploit-ok elleni védelel legyen ki- vagy bekapcsolva.

Így néz ki általában egy ilyen osztály:

```
@Configuration
@EnableWebSecurity // (1)
public class WebSecurityConfig extends WebSecurityConfigurerAdapter { // (1)

  @Override
  protected void configure(HttpSecurity http) throws Exception {  // (2)
      http
        .authorizeRequests()
          .antMatchers("/", "/home").permitAll() // (3)
          .anyRequest().authenticated() // (4)
          .and()
       .formLogin() // (5)
          .loginPage("/login") // (5)
          .permitAll()
          .and()
      .logout() // (6)
        .permitAll()
        .and()
      .httpBasic(); // (7)
  }
}
```

1. Egy normális @Configuration-ös osztály, amin van egy **@EnableWebSecurity** annotáció is, és kiterjeszti a **WebSecurityConfigurerAdapter** absztrakt osztályt.
2. Azzal, hogy felülírjuk az adapter configure(HttpSecurity http) metódusát, kapunk egy szép DSL-t – Domain Specific Language - , amivel be lehet konfigurálni a szűrőláncot.

3. Minden „/”-re és „/home”-ra menő kérés engedélyezve van, a felhasználónak ezekhez nem kell hitelesítenie magát. Ezt antMatcher-rel oldjuk meg.
4. Minden más kérésnél a felhasználónak hitelesítenie kell magát, pl. be kell jelentkeznie.
5. Engedélyezed az űrlapos bejelentkezést egy felhasználónév-jelszó párossal, egy saját bejelentkezőoldalon – azaz pl. nem a Spring Security által automatikusan generált oldalon. Mindenkinek hozzá kell tudni férnie előzetes bejelentkezés nélkül.
6. Ugyanez igaz a kijelentkezőoldalra is.
7. Ezen kívül engedélyezzük a Basic HTTP-s hitelesítést is.

Ebben a configure(HttpSecurity http) metódusban adjuk meg az alábbiakat:

- Milyen URL-eket kell megvédeni (authenticated()), és milyen URL-ek vannak engedélyezve (permitAll())
- Milyen hitelesítési módok engedélyezettek (formLogin(), httpBasic()), és ezek hogyan vannak felkonfigurálva
- Röviden: Az alkalmazásod teljes biztonsági konfigurációját

Felhasználók hitelesítése – Authentication

Alapból három eset van a Spring Security-ben a felhasználók hitelesítésénél:

1. Hozzáférsz a felhasználó – hash-elt – jelszavához, mert pl. az adatait egy adatbázisban tárolod – ez az alap
2. Kevésbé gyakori: Nem tudsz hozzáférni a felhasználó – hash-elt – jelszavához. Ez akkor áll fenn, ha a felhasználók adatait valahol máshol tároljuk, például egy harmadik féltől származó identity management product-ban, ami a felhasználók hitelesítéséhez REST-szolgáltatásokat nyújt. Ilyen például az Atlassian Crowd.
3. Szintén népszerű: OAuth2-t, vagy OpenID-t (Google-lel/Twitter-rel, stb.-vel történő belépés) akarunk használni, általában JWT-vel – JSON Web Tokens - együtt.

Nézzük meg az első 2 esetet:

1. UserDetailsService: Hozzáférünk a felhasználó jelszavához

Képzeljük el, hogy van egy adatbázistáblánk, amiben a felhasználók adatait tároljuk. Ebben vannak oszlopok, köztük a felhasználóneveket, és a hash-elt jelszavakat tartalmazó oszlop.

Ebben az esetben a Spring Securitynek szüksége van 2 bean-re ahhoz, hogy a hitelesítés működhessen:

a) UserDetailsService

b) PasswordEncoder

A UserDetailsService megadása így zajlik:

```
@Bean
public UserDetailsService userDetailsService() {
    return new MyDatabaseUserDetailsService(); // (1)
}
```

Forrás: https://www.marcobehler.com/guides/spring-security?fbclid=IwAR05uQQxBrEU4j3agMv-TIQ3EXatbBmVR_-uKiTsWQ3t4vuDDaZqeGaFfp8

A MyDatabaseUserDetailsService implementálja a UserDetailsService interfészt, ami egy metódust tartalmaz, ami egy UserDetails típusú objektumot ad vissza:

```
public class MyDatabaseUserDetailsService implements UserDetailsService {

        UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
         // 1. Load the user from the users table by username. If not found, throw Usernam
         // 2. Convert/wrap the user to a UserDetails object and return it.
        return someUserDetails;
    }
}

public interface UserDetails extends Serializable { // (2)

    String getUsername();

    String getPassword();

    // <3> more methods:
    // isAccountNonExpired,isAccountNonLocked,
    // isCredentialsNonExpired,isEnabled
}
```

Forrás: https://www.marcobehler.com/guides/spring-security?fbclid=IwAR05uQQxBrEU4j3agMv-TIQ3EXatbBmVR_-uKiTsWQ3t4vuDDaZqeGaFfp8

A UserDetailsService a UserDetails-t a felhasználó neve alapján tölti be, amint azt a paramétere is mutatja a metódusnak. A UserDetails interfésznek 2 metódusa van: az egyikkel a felhasználónevet, a másikkal a hash-elt(!) jelszót kapjuk meg.

A UserDetails interfésznek vannak további metódusai is, amikkel a fiók aktív/blokkolt állapotát, vagy a hitelesítő adatok lejártságát tudjuk ellenőrizni.

A UserDetailsService és UserDetails interfészeket bármikor implementálhatjuk mi magunk is! De a Spring Security-nak is vannak kész implementációi is:

a) **JdbcUserDetailsManager:** Ez egy JDBC (adatbázis) alapú UserDetailsService. BE lehet konfigurálni úgy, hogy a felhasználókat tartalmazó tábla felépítéséhez.

b) **InMemoryUserDetailsManager:** Ez minden UserDetails-t a memóriában tart, így remekül lehet vele tesztelni.

c) **org.springframework.security.core.userdetail.User:** Ez egy alapértelmezett UserDetails implementáció, amit nyugodtan használhatunk mi is.

A UserDetails működése HTTP Basic Authentication-nel bejelentkezés során:

a) Kiszedi a felhasználónév-jelszó párost a HTTP Basic Auth-os fejlécből, és átadja egy szűrőnek. Ez magától megtörténik.

b) Ezután meg kell hívnunk a MyDatabaseUserDetailsService-t, hogy töltse be a megfelelő felhasználót az adatbázisból egy UserDetails objektum formájában, amin keresztül elérjük a hash-elt jelszavát a felhasználónak.

c) Ezután vennünk kell a fejlécből kiszedett jelszót, automatikusan hash-elni kell, és összehasonlítani a UserDetails objektumból jövő hash-elt jelszóval. Ha megegyeznek, a felhasználó sikeresen hitelesítette magát.

Ez mind szép és jó, de hogyan hash-eli a Spring a kliens jelszavát?

Itt jönnek képbe a PasswordEncoder-ek.

PasswordEncoders: Ehhez meg kell adnunk egy PasswordEncoder típusú bean-t, amit el kell látnunk a @Bean annotációval. Ha mondjuk BCrypt hash-

elő algoritmust akarunk használni MINDEN jelszóhoz, akkor ezt a bean-t kéne definiálni a SecurityConfig-ban:

```
@Bean
public BCryptPasswordEncoder bCryptPasswordEncoder() {
    return new BCryptPasswordEncoder();
}
```

Forrás: https://www.marcobehler.com/guides/spring-security?fbclid=IwAR05uQQxBrEU4j3agMv-TIQ3EXatbBmVR_-uKiTsWQ3t4vuDDaZqeGaFfp8

Mi van akkor, ha több hash-elő algoritmusunk van? Akkor ezt használjuk:

```
@Bean
public PasswordEncoder passwordEncoder() {
    return PasswordEncoderFactories.createDelegatingPasswordEncoder();
}
```

Forrás: https://www.marcobehler.com/guides/spring-security?fbclid=IwAR05uQQxBrEU4j3agMv-TIQ3EXatbBmVR_-uKiTsWQ3t4vuDDaZqeGaFfp8

Ez meg fogja nézni a UserDetails objektum hash-elt jelszavát – ami valószínűleg egy adatbázisból jön majd - , aminek most már egy {prefix}-szel kell kezdődnie. Ez a prefix fogja megmondani a használandó hash-algoritmust:

username	password
john@doe.com	{bcrypt}$2y$12$6t86Rpr3llMANhCUt26oUen2WhvXr/A89Xo9zJion8W7gWgZ/zA0C
my@user.com	{sha256}5ffa39f5757a0dad5dfada519d02c6b71b61ab1df51b4ed1f3beed6abe0ff5f6

Table 1. Users Table

Forrás: https://www.marcobehler.com/guides/spring-security?fbclid=IwAR05uQQxBrEU4j3agMv-TIQ3EXatbBmVR_-uKiTsWQ3t4vuDDaZqeGaFfp8

A Spring Security mit fog itt csinálni?

1. Beolvassa a jelszavakat és lenyesi az elejéről a prefixet.
2. A prefix alapján felhasználja a megfelelő PasswordEncodert – BcryptEncoder, SHA256Encoder, …

3. Ezután hash-eli a kapott, nyers jelszót, ezzel a PasswordEncoder-rel, és összehasonlítja a tárolt hash-sel.

Gyorstalpaló: Ha Spring Security-t használsz, és hozzáférsz a jelszavakhoz, akkor:

1. Adj meg egy UserDetailsService-t: vagy saját implementációt, vagy egy Spring Security által biztosítottat.
2. Adj meg egy PasswordEncoder-t.

2. **AuthenticationProvider: nem férünk hozzá a jelszavakhoz**

Képzeljük el, hogy most az Atlassian Crowd-ot használjuk az adatok tárolására, azaz minden felhasználónk neve és jelszava az Atlassian Crowd-ban vannak eltárolva, és nem a mi adatbázisunkban.

Ez két dolgot jelent:

a) Többé NEM állnak rendelkezésünkre a jelszavak az alkalmazásunkban, a Crowd meg nem fogja csak úgy ideadni nekünk ingyen.

b) Azonban van egy REST API-nk, aminek a segítségével bejelentkezhetünk – egy POST kérés küldésével.

Ebben az esetben nem használhatjuk többet a UserDetailsService-t, helyette ehelyett implementálnunk kell, és meg kell adnunk egy **AuthenticationProvider** bean-t, és kell látni a @Bean annotációval:

```
@Bean
public AuthenticationProvider authenticationProvider() {
    return new AtlassianCrowdAuthenticationProvider();
}
```

Forrás: https://www.marcobehler.com/guides/spring-security?fbclid=IwAR05uQQxBrEU4j3agMv-TIQ3EXatbBmVR_-uKiTsWQ3t4vuDDaZqeGaFfp8

Az AuthenticationProvider interfészben mindössze 1 metódus van, aminek egy naiv implementációja valahogy így nézne ki:

```
public class AtlassianCrowdAuthenticationProvider implements AuthenticationProvider {

        Authentication authenticate(Authentication authentication)  // (1)
                throws AuthenticationException {
            String username = authentication.getPrincipal().toString(); // (1)
            String password = authentication.getCredentials().toString(); // (1)

            User user = callAtlassianCrowdRestService(username, password); // (2)
            if (user == null) {                                          // (3)
                throw new AuthenticationException("could not login");
            }
            return new UserNamePasswordAuthenticationToken(user.getUsername(), user.getPas
        }
            // other method ignored
}
```

Forrás: https://www.marcobehler.com/guides/spring-security?fbclid=IwAR05uQQxBrEU4j3agMv-TIQ3EXatbBmVR_-uKiTsWQ3t4vuDDaZqeGaFfp8

1. A UserDetails load() metódusával szemben, ahol csak a felhasználónévhez fértünk hozzá, itt most hozzáférünk a teljes hitelesítési kísérlethez, ami ÁLTALÁBAN tartalmazza a felhasználónevet ÉS a jelszót is.
2. Itt kb. akármit csinálhatunk a felhasználó hitelesítéséhez, pl. meghívhatunk egy REST-szolgáltatást.
3. Ha a hitelesítés nem sikerült, dobni kell egy kivételt.
4. Ha a hitelesítés sikeres volt, akkor vissza kell adnunk egy teljesen inicializált UsernamePasswordAuthenticationToken-t. Ez az Authentication interfész egy implementációja, ahol az authenticated nevű mezőt true-ra kell állítani – amit a fentebb használt konstruktor automatikusan megtesz.

Az AuthenticationProvider működése HTTP Basic Authentication-nel történő bejelentkezésnél:

a) Kiszedi a felhasználónév-jelszó párost a HTTP Basic Auth-os fejlécből, és átadja egy szűrőnek. Ez magától megtörténik.

b) Meghívja az AuthenticationProvider-ünket, pl. az AtlassianCrowdAuthenticationProvider-t, és azzal a felhasználónévvel és jelszóval megcsinálhatjuk pl. egy REST-hívással a hitelesítést.

Itt nincs semmiféle jelszó hash-elés, mivel gyakorlatilag egy harmadik félre bízzuk a felhasználói adatok ellenőrzését.

Gyorstalpaló: Ha Spring Security-t használsz, és NEM férsz hozzá a felhasználó jelszavához, akkor implementálj és adj meg egy AuthenticationProvider @Bean-t.

Authorization a Spring Security-vel:

Itt két nagyon fontos fogalom az **Authority** és a **Role**, amik arra szolgálnak, hogy különböző felhasználóknak más mértékű, másféle hozzáférést akarunk adni a rendszerhez, amik az **Authority**-nek és a **Role**-oknak a függvényei.

Mi az az Authority?

Egyszerűen szólva egy String, ami bármi lehet: user, ADMIN, ROLE_ADMIN, vagy 53cr37_r0l3

Mi az a Role?

A Role egy Authority egy *ROLE_* prefixszel. Tehát egy **ADMIN** Role ugyanaz, mint a **ROLE_ADMIN** Authority.

A kettő közötti megkülönböztetés kicsit rejtélyes, oka nem teljesen tiszta.

A Spring Security természetesen nem fogja megengedni, hogy csak úgy Authority-ként lehessen kezelni a sima String-eket. Van egy Java-osztály, ami egy String Authority-t jelképez, egy ilyen népszerű osztály a **SimpleGrantedAuthority**:

```
public final class SimpleGrantedAuthority implements GrantedAuthority {

        private final String role;

    @Override
        public String getAuthority() {
                return role;
        }
}
```

Forrás: https://www.marcobehler.com/guides/spring-security?fbclid=IwAR05uQQxBrEU4j3agMv-TIQ3EXatbBmVR_-uKiTsWQ3t4vuDDaZqeGaFfp8

Vannak más Authority-t reprezentáló osztályok is, de azokat itt most békén hagyjuk.

1. **UserDetailsService**

Ha az adatokat a saját alkalmazásunkban tároljuk – gondoljunk a UserDetailsService-re - , akkor lesz egy Felhasználók (Users) nevű táblánk.

Ezután csak simán hozzáadhatunk egy „Authorities” című oszlopot a táblához. Ezen kívül nyilván léteznek alternatív megoldásai is ennek az adatbázis részéről nézve.

Az Authority-ket le fogjuk menteni az adtatbázisba, és ezek az Authority-k egy „ROLE_” prefixszel fognak kezdődni, így a Spring Security szakszavaival élve ezek is Role-ok lesznek.

username	password	authorities
john@doe.com	{bcrypt}...	ROLE_ADMIN
my@callcenter.com	{sha256}...	ROLE_CALLCENTER

Forrás: https://www.marcobehler.com/guides/spring-security?fbclid=IwAR05uQQxBrEU4j3agMv-TIQ3EXatbBmVR_-uKiTsWQ3t4vuDDaZqeGaFfp8

Ezután már csak ki kell egészíteni a UserDetailsService-ünket úgy, hogy ezt az új Authorities oszlopot is beolvassa:

```
public class MyDatabaseUserDetailsService implements UserDetailsService {

  UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
     User user = userDao.findByUsername(username);
     List<SimpleGrantedAuthority> grantedAuthorities = user.getAuthorities().map(authority
     return new org.springframework.security.core.userdetails.User(user.getUsername(), use
  }

}
```

Forrás: https://www.marcobehler.com/guides/spring-security?fbclid=IwAR05uQQxBrEU4j3agMv-TIQ3EXatbBmVR_-uKiTsWQ3t4vuDDaZqeGaFfp8

1. Akármi is van az új oszlopunkban, azt belenyessük egy SimpleGrantedAuthority-ket tartalmazó listába. Ennyi.

2. Itt megint csak a Spring Security-s alap UserDetails implementációt használjuk. Ehelyett csinálhatnánk egy saját implementációt, és akkor lehet még a map-elést is megspórolnánk.

2. AuthenticationManager: Hol tároljuk, és honnan szerezzük meg az Authority-ket?

Ha a felhasználók egy harmadik féltől származó alkalmazást használnak, pl. Atlassian Cloud-ot, akkor rá kell jönnünk, hogy ott milyen dolgok azok, amik támogatják az Authority-ket. Az Atlassian Crowd-nál pl. ott voltak a role-ok, de aztán ez később elavult lett, és átálltak a „group”-ra.

Szóval a használt szoftvertől függően ezt a cuccot át kell alakítanunk az AuthenticationProvider-ünkben egy Spring Security-s Authority-vé:

```
public class AtlassianCrowdAuthenticationProvider implements AuthenticationProvider {

    Authentication authenticate(Authentication authentication)
            throws AuthenticationException {
        String username = authentication.getPrincipal().toString();
        String password = authentication.getCredentials().toString();

        atlassian.crowd.User user = callAtlassianCrowdRestService(username, password); // (1
        if (user == null) {
            throw new AuthenticationException("could not login");
        }
        return new UserNamePasswordAuthenticationToken(user.getUsername(), user.getPassword(
    }
            // other method ignored
}
```

Forrás: https://www.marcobehler.com/guides/spring-security?fbclid=IwAR05uQQxBrEU4j3agMv-TIQ3EXatbBmVR_-uKiTsWQ3t4vuDDaZqeGaFfp8

1. Ez nem tényleges Atlassian Crowd-os kód, de a célnak megfelel. Egy REST-szolgáltatással hitelesítjük magunkat, és visszakapunk egy JSON User objektumot, amit aztán a program átkonvertál atlassian.crowd.User típusú objektummá.
2. A felhasználó tagja lehet akárhány group-nak, amik itt csak String-ek lesznek. Ezeket a group-okat egyszerűen leképezzük SimpleGrantedAuthority-re a képről lenyesett mapToAuthorities(user.getGroups())-szal.

Épp eleget beszéltünk az Authority-k megszerzéséről és tárolásáról. De hogyan védjük meg a különböző Authority-kkel rendelkező URL-eket?

Hát így:

```
@Configuration
@EnableWebSecurity
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {

        @Override
        protected void configure(HttpSecurity http) throws Exception {
        http
          .authorizeRequests()
            .antMatchers("/admin").hasAuthority("ROLE_ADMIN") // (1)
            .antMatchers("/callcenter").hasAnyAuthority("ROLE_ADMIN", "ROLE_CALLCENTER") //
            .anyRequest().authenticated() // (3)
            .and()
         .formLogin()
           .and()
         .httpBasic();
        }
}
```

Forrás: https://www.marcobehler.com/guides/spring-security?fbclid=IwAR05uQQxBrEU4j3agMv-TIQ3EXatbBmVR_-uKiTsWQ3t4vuDDaZqeGaFfp8

1. Ahhoz, hogy hozzáférjünk a /admin-hoz, hitelesítenünk kell magunkat, ÉS rendelkeznünk kell a ROLE_ADMIN Authority-vel.
2. A callCenter-hez való hozzáféréshez hitelesíteni kell magunkat, ÉS rendelkeznünk kell VAGY a ROLE_ADMIN, vagy a ROLE_CALLCENTER Authority-vel.
3. Bármilyen más kéréshez nem szükséges semmilyen Role sem, csak hitelesíteni kell magunkat.

A fentebbi (1, 2) ugyanaz, mintha ezt írtuk volna Role-okkal:

```
http
  .authorizeRequests()
    .antMatchers("/admin").hasRole("ADMIN") // (1)
    .antMatchers("/callcenter").hasAnyRole("ADMIN", "CALLCENTER") // (2)
```

Forrás: https://www.marcobehler.com/guides/spring-security?fbclid=IwAR05uQQxBrEU4j3agMv-TIQ3EXatbBmVR_-uKiTsWQ3t4vuDDaZqeGaFfp8

1. A hasAuthority() helyett használhatjuk a hasRole() metódust is. Ebben az esetben a Spring Security egy ROLE_ADMIN nevű Authority-t fog keresni a felhasználónál.

2. A hasAnyAuthority() metódus meghívása helyett bevethető a hasAnyRole() metódushívás is. Itt a Spring Security egy ROLE_ADMIN vagy ROLE_CALLCENTER nevű Authority-t fog keresni a felhasználónál.

És a végére maradt a legerősebb módja az authorizáció konfigulásának, és ez pedig az access() metódus. Ezzel gyakorlatilag bármilyen érvényes SpEL - Spring Expression Language - kifejezést definiálhatunk:

```
http
   .authorizeRequests()
    .antMatchers("/admin").access("hasRole('admin') and hasIpAddress('192.168.1.0/24') and
@myCustomBean.checkAccess(authentication,request)") // (1)
```

Forrás: https://www.marcobehler.com/guides/spring-security?fbclid=IwAR05uQQxBrEU4j3agMv-TIQ3EXatbBmVR_-uKiTsWQ3t4vuDDaZqeGaFfp8

Itt lellenőrizzük az 1.-nél, hogy a felhasználónak van-e ROLE_ADMIN-ja, ez bizonyos IP-címe, és egy bean check-et is tolunk a végére.