Webes architektúrák, Spring MVC

Imre Gábor

Q.B224

gabor@aut.bme.hu

BME AUT
Automatizálási és
Alkalmazott
Informatikai Tanszék

Webes architektúrák

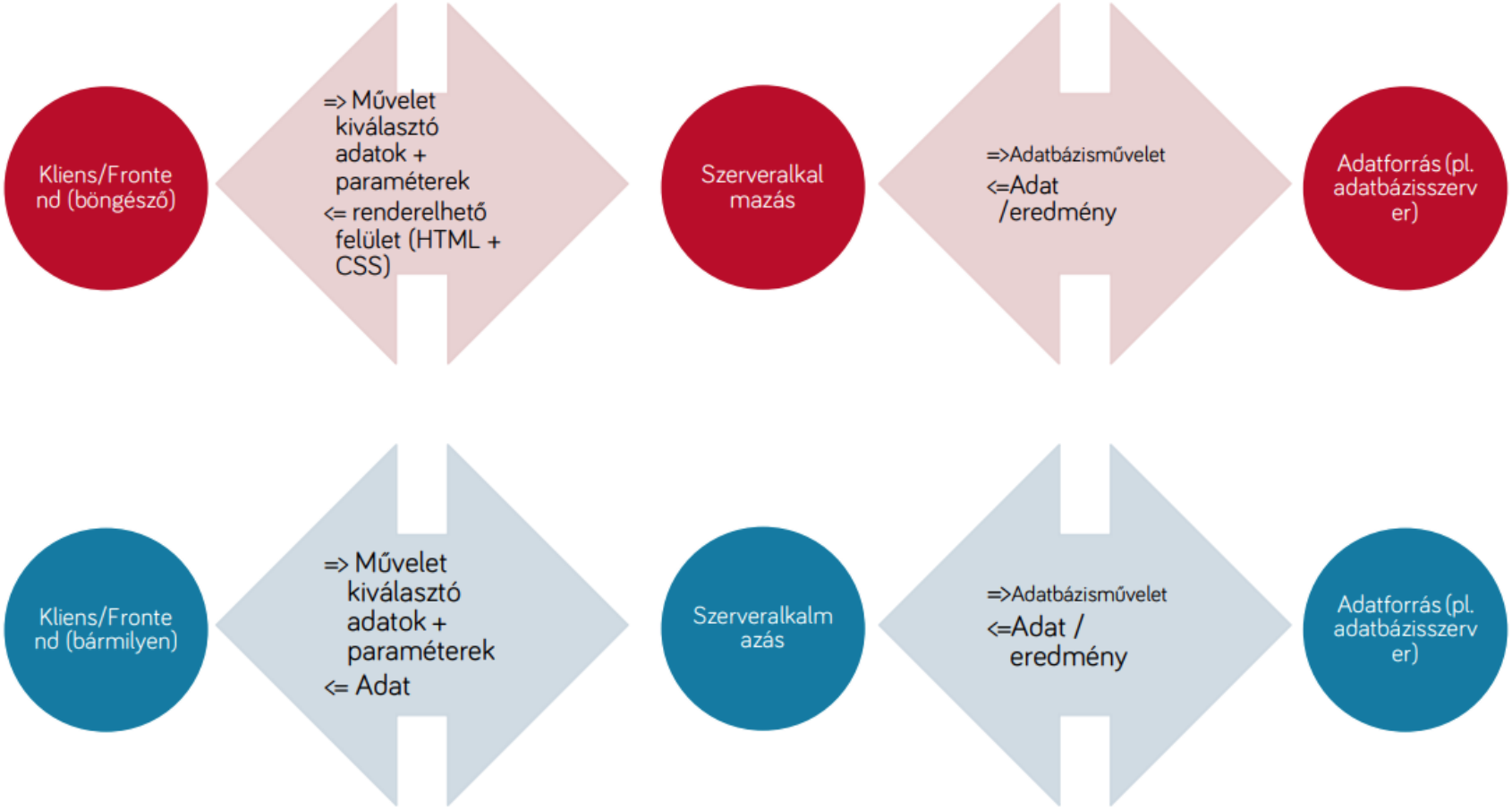
Web alapok

- Lásd a Kliensalkalmazások tantárgy webes bevezető előadását
  - HTTP alapok
  - HTTP kliens (pl. böngésző) és webszerver működése
  - Webalkalmazások
  - Statikus és dinamikus tartalom/kiszolgálás
  - REST

Szerver vs kliens oldali renderelés

- Szerver renderelt
  - A szerveralkalmazás felületleírót (HTML+CSS) ad válaszul a kliensnek
  - Csak böngészős frontend
  - A felület szerkezetének előállításhoz szükséges logika a szerveren fut
- Kliens renderelt
  - A szerveralkalmazás a felület előállításához szükséges adatot adja
  - A felület szerkezetének előállításhoz szükséges logika a kliensen fut
  - Gyakorlatilag bármilyen klienstechnológiához illeszkedik
    - Feltétel: adatot tudjon küldeni/fogadni hálózaton keresztül

Szerver vs kliens oldali renderelés



A renderelés helye szerinti előnyök

Kliens

- Teljes szeparáció a UI és backend csapatok között
  - Más programozói ismereteket igényelnek
- Jellemzően jóval gazdagabb felhasználói élmény
- A kliensalkalmazás használható lehet offline módban (PWA)
- Jellemzően kevesebb adat utazik az első betöltés után, a frissítés gyorsabb
- Szükség szerint egyszerűbben cserélhető a teljes UI
- Kíméli a szerver erőforrásait

Szerver

- Gyors prototipizálás, potenciálisan gyorsabb fejlesztési ciklusok
- Kevesebb kliensoldali ismeret szükséges
- Egyszerűbben generálható kód az üzleti modellek alapján (használható reflexió)
- Kevésbé törékeny a felhasználói felület változására
- Jellemzően kevesebb adat utazik az első betöltéskor, a betöltés gyorsabb
- Nem szükséges komplex kommunikációs réteg karbantartása
- Régi böngészőket is egyszerűen támogat
- Kíméli a kliens erőforrásait

Szerver renderelt

- Ha nincs JavaScript is mellé, akkor elég korlátozott felhasználói élmény
  - Minden felületi változáshoz kommunikáni kellene a szerverrel
  - Ma már a klienseszközök nagyon erősek, nem szükséges minden felületi logikának a szerveren futnia
- Vegytisztán ma már nem alkalmazzuk, többkevesebb JavaScript logikát is mellékel a szerver, amit a böngésző futtat
  - Pl. jQuery – egyszerűbb felületi módosítások a böngészőn belül is végrehajtódnak
  - Egyes műveletek kliens renderelt módon működhetnek, pl. a JS kód adatot kérdez le a szerverről, ami alapján frissíti a felületet

Kommunikáció

- Kliens-szerveralkalmazás között
  - Szerver renderelt esetben: HTTP alapú protokoll(ok)
  - Kliens renderelt: amit a kliens és a szerver is támogat
    - Többféle kliens is lehet, a közös nevező tipikusan a HTTP
    - Az üzenetek formátuma régebben sokszor XML, ma gyakrabban JSON
    - Teljesítménykritikus esetben bináris formátumok, pl. protobuf, Apache Avro, ill. ráépülő RPC protokollok: gRPC (protobuf-ra épít), Apache Thrift (saját bináris formátum)
- Szerveralkalmazás – adatbázis között
  - Relációs DB: saját protokoll, a DB driver/provider intézi
  - NoSQL DB: HTTP alapú protokollok

Spring MVC

Servlet, JSP

Java (Jakarta) EE webes komponensek

- A J2EE kezdetben két egyszerű webes technológiát definiált
- Szervlet
  - Kérés-válasz alapú protokollok szerver oldalát megvalósító Java osztály
    - Speciális esete a web szervlet, amikor a protokoll a HTTP, a továbbiakban ezt értjük szervlet alatt
  - Ún. **webkonténer**ben fut, ez hasznos szolgáltatásokat ad a szervletnek
    - A Java EE alkalmazásszervereknek kötelezően része, de önállóan is elérhetők (pl. Apache Tomcat, Jetty)

- JSP – JavaServer Pages
  - „Olyan mint a PHP, ASP”
  - HTML kód, JSP vezérlő kódok és Java kódok
  - Bővíthető tag library
  - „kifordított szervlet”
  - Szintén webkonténerben fut, végül szervletté fordul

Szervlet-JSP példa

```
@WebServlet("/hello")

public class HelloServlet extends
   HttpServlet {

   public void doGet(

      HttpServletRequest request,
   HttpServletResponse response) throws
   ServletException, IOException {

        String userName =

           request.getParameter("userName");


   response.setContentType("text/html");

        PrintWriter out =
   response.getWriter();

        out.println("<html><head>Sample
   servlet</head><body>");

        out.println("<h1>Hello " + userName
   + "</h1>");

        out.println("</body></html>");

   }

}
```

```
<html>

 <head>Sample JSP</head>

 <body>

  <h1>Hello ${param.userName}</h1>

 </body>

</html>
```

Szervlet és JSP

- Szervlet
  - HTML kód Java-n belül
  - Bármilyen adatfeldolgozás
  - Egyszerű a webes kérésekhez
  - Nem az egyszerűen karbantartható kimenetet szolgálja
- JSP
  - Java-kód a HTML-ben
  - Egyszerűbb a formázás, bonyolultabb a feldolgozás
  - Bináris tartalmat nem tud generálni
  - A kód szervletté fordul
- Általában együtt használjuk őket (Model-View-Controller szemlélet)
  - Szervlet: controller
  - JSP: view
- Vagy nem is használjuk őket közvetlenül → webalkalmazás keretrendszer
  - Tipikusan webkonténert igényelnek, mert a Servlet API-ra épülnek
  - A JSP-t vagy felhasználják, vagy más view technológiát használnak
  - Számos hasznos feature, pl. kész GUI komponensek, AJAX, eseménykezelés, validáció, navigációs szabályok
  - Néhány példa: Spring MVC, JavaServer Faces (a Java EE része), Struts, Tapestry, Wicket, GWT, Vaadin

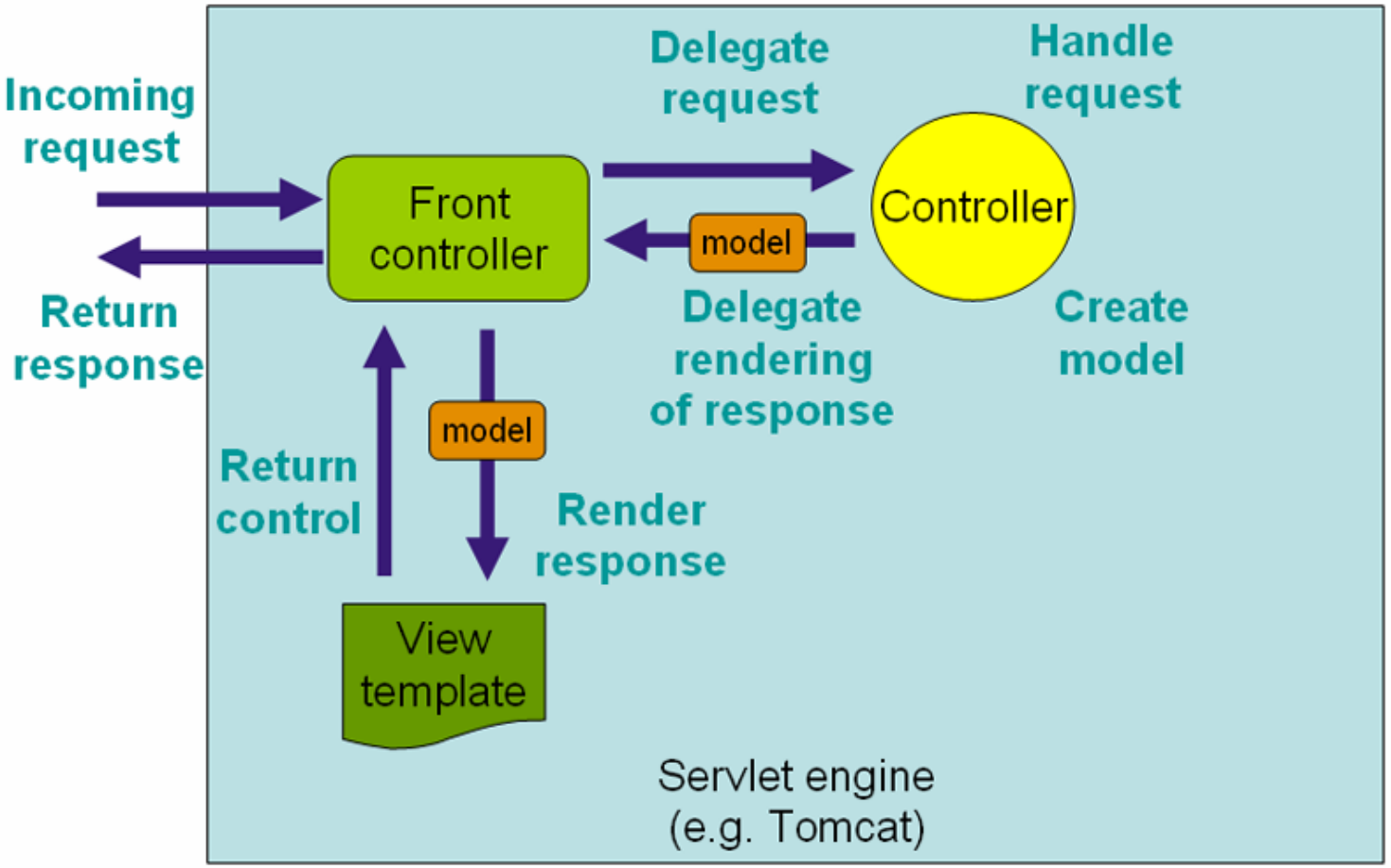
Spring MVC

Spring MVC alapok

Spring MVC: általános jellemzők

- Az objektumok felelősségeinek jó szétválasztása (MVC, validátorok, stb.)
- Rugalmasan konfigurálható
- Nincsenek előírt ősosztályok, interfészek
- Annotációk támogatása
- Testre szabható validáció
- Több megjelenítési technológia támogatása (JSP, Thymeleaf, Velocity, FreeMarker)
- Nincs UI komponens modellje

Spring MVC: működés



Hello World példa

- Egy URL paramétert kap, és azt a személyt köszönti, pl.:

http://localhost:8080/hello?user=Peter

- Eredménye:

Hello Peter

HelloWorldController.java

- A kérések feldolgozását egy controller osztály végzi

```
@Controller
public class HelloWorldController {

   @GetMapping("/hello")
   public String helloWorld(
      @RequestParam("user") String user,
      Map<String, Object> model) {
         model.put("welcome", "Hello " + user);
         return "index";
   }
}
```

index.html

Szerver oldali renderelés a Thymeleaf template engine segítségével

```
<!DOCTYPE HTML>
<html
  xmlns:th="http://www.thymeleaf.org">
    <head>
      <title>Welcome</title>
    </head>
    <body>
      <h1 th:text="${welcome}">Some
  welcome</h1>
    </body>
</html>
```

Spring MVC konfigurációja

- Spring Boot nélkül XML és/vagy JavaConfigra lenne szükség a Spring MVC, és még a Thymeleaf bekonfigurálásához
- Spring Boot esetén ez nem szükséges, csak
  - két függőség:

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

    - és

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

  - és egy szokásos @SpringBootApplication osztály
  - A controllerek elhelyezése: a "root" package valamelyik alpackage-ébe
  - Thymeleaf-es html fájlok az src\main\resources\templates alá

Spring Boot webalkalmazás futtatása

- Ha a spring-boot-starter-webtől függünk, beágyazott webkonténert indít, by default Tomcat-et
  - Jetty vagy Undertow is kérhető, a spring-boot-starter-tomcat függőség kizárásával, és a …-jetty vagy …-undertow hozzáadásával
  - →nem szükséges külön webkonténer, fejlesztés során pedig a deploy kimarad
  - By default a 8080-as porton, a root contexten érhető el az alkalmazás, vagyis pl. http://localhost:8080/
    - Felüldefiniálható propertykkel: server.port, server.context-path
- Futtatható jar készítése:
  - A pom-ban a packaging legyen jar
  - spring-boot plugin a pom-ban → függőségeket is belecsomagolja a jar-ba és a spring boot loader-t, ami képes betölteni ezeket a függőségeket
  - mvn package → target mappában kész jar
  - java –jar myapp-0.1.1-SNAPSHOT.jar
- Hagyományos, különálló webkonténerre telepíthető war is készíthető (kicsit más pom + egy ősosztály)

Spring MVC

Handler metódusok

Controller osztály: URL leképezés

- Tetszőleges számú handler metódusa lehet, amelyek más-más URL-t kezelnek
- Maga a controller osztály is kaphat `@RequestMapping` annotációt, ilyenkor ehhez relatívan értelmeződnek a handler metódusokon lévő URL minták
- Példák URL mintára:

```
@RequestMapping(value="/owners/{ownerId}", method=RequestMethod.GET)
public String findOwner(@PathVariable("ownerId") String ownerId){...}

@RequestMapping(value="/owners/{ownerId}/pets/{petId}")
public String findPet(
  @PathVariable String ownerId,
  @PathVariable int petId) {...}
```

- Szűrés header vagy paraméter alapján:

```
@RequestMapping(value = "/pets/{petId}", params="myPar=myVal")
@RequestMapping(value = "/pets", headers="content-type=text/*")
```

- Szintén érvényes: /myPath/*.do, /owners/*/pets/{petId}
- Spring MVC 4.3 óta @RequestMapping helyett @GetMapping, @PostMapping, ...

Controller osztály: handler metódusok

- Rugalmas argumentumlista, pl.
  - `ServletRequest`, `ServletResponse`, `HttpSession`
  - `@PathVariable`, `@RequestParam`
  - `@RequestHeader`, `@CookieValue`
  - `@RequestBody`
    - típusához `HttpMessageConverter` kell
    - A `HttpEntity<?>` nemcsak a törzshöz, hanem a headerhez is hozzáférést ad
  - Tetszőleges saját, ún. command objektumok, melyek a formokban lévő adatokat fogják össze
  - `java.util.Map` vagy `ModelMap` vagy `Model`, ami az aktuális modellt tartalmazza
  - `@ModelAttribute`-tal annotált modell-beli objektumok
    - A modellből kinyeri az adott nevű objektumot
  - `Errors`/`BindingResults` (az előtte lévő `@ModelAttribute`-hoz tartozó konverziós/validációs hibákat reprezentálja)

Controller osztály: handler metódusok

- Rugalmas visszatérési típus:
  - `String`, ami a `View` nevét jelenti
  - void: ha magunk generáljuk a választ bemenő argumentumként lapott `ServletResponse`-on keresztül, vagy `RequestToViewNameTranslator` határozza meg a következő view-t
  - tetszőleges típus, `@ResponseBody`-s handler metódus esetén (`HttpMessageConverter`rel fog szerializálódni a válaszba)
  - `HttpEntity<?>`, szintén `HttpMessageConverter`rel
  - `ResponseEntity<?>`, ha a válasz státusz kódját is állítani akarjuk

HttpMessageConverter

- HTTP kérés törzsét lehet leképezni vele objektummá (`@RequestBody` és `@ResponseBody` annotációk)
- Beépített converterek:
  - `ByteArrayHttpMessageConverter`
  - `StringHttpMessageConverter`
  - `FormHttpMessageConverter`
  - `SourceHttpMessageConverter`
  - `MarshallingHttpMessageConverter`
  - `MappingJackson2HttpMessageConverter`
    - a spring-boot-starter-web a Jackson nevű JSON libraryt is hozza
- Nagy a jelentősége REST stílusú webalkalmazások esetén

Spring MVC

RESTful webszolgáltatások

RESTful webszolgáltatások

- RESTful szolgáltatás (REST API): HTTP fölött, URI alapon egyszerűen címezhető szolgáltatás, melyhez meg kell adni
  - a szolgáltatás URI-jét
  - a szolgáltatás által támogatott adatformátumot
  - a szolgáltatás által támogatott HTTP metódusokat (általában GET, POST, PUT, DELETE)
- Az interfész formális leírására az OpenAPI (Swagger) használható
- A leggyakoribb módja egy szerver oldali funkció publikálásának, kliens oldalra, de backend-backend közti hívások esetében is
- Az üzenetformátum nem kötött, nagyon gyakran JSON

Egy tipikus RESTful webszolgáltatás

- http://example.com/books/
  - GET: könyvek listáját adja vissza
  - POST: új könyvet szúr a listába, az azonosítót a szerver adja
  - DELETE: törli a teljes listát
  - PUT: lecseréli a teljes listát
- http://example.com/books/xyz
  - GET: egy konkrét könyvet ad vissza, melynek xyz az azonosítója
  - POST: az xyz elemet listának tekinti, és alárendel egy új elemet (pl. fejezetet a könyvön belül)
  - DELETE: törli az xyz könyvet
  - PUT: módosítja az xyz könyvet, vagy újat hoz létre xyz azonosítóval

REST API fejlesztése Spring MVC-vel

- A Spring MVC kezdettől fogva alkalmas REST API fejlesztésére
  - A controller handler metódusra @ResponseBody annotáció
    → a metódus visszatérési értéke a válaszba megy bele
  - Handler metódus bemenő paraméterre @RequestBody
    → a kérés törzse injektálódik
  - A törzs (de)szerializálásához, különböző formátumokhoz HttpMessageConverter-ek használhatók (amelyek közül a tipikus JSON, XML készen áll)
  - A formátumok között a @RequestMapping produces és consumes paraméterével választhatunk, ha nem választunk, application/json a default
- Kis egyszerűsítés a Spring 4-ben: @RestController az osztályra → @ResponseBody elhagyható

Data Transfer Objects

- A REST API-n, tipikusan JSON-ben megjelenő adat típusok több problémát is felvetnek
  - Körkörös hivatkozás tagváltozókon keresztül → végtelen ciklus
  - Az adatbázisban tárolt típusok (entitások) és a REST API-n utazó típusok kapcsolata
    - Adott kérésnél fölösleges vagy nem publikus mezők kihagyása a JSON-ből
    - Több entitás adatainak összefogása egy REST végpont kérésébe/válaszába
    - REST interfész és adatmodell szoros/laza csatolása
- Problémák kezelésére két megközelítés
  - A REST API-n megjelenő típusok közvetlenül a DB-ben tárolt entitások legyenek
  - A REST API-n ne az entitások utazzanak, hanem új, erre dedikált típusok = DTO (Data Transfer Object)

DTO-mentes megoldás

- Előnyök:
  - Nem kell új (az entitásokhoz nagyon hasonló) osztályokat bevezetni
  - Nem kell megírni az oda-vissza leképezést (konvertálást) az entitások és DTO-k között
  - A leképezés futási idejű overheadje nem jelentkezik
- Hátrányok:
  - Az adatmodell és a REST API szoros csatolása (egyik módosítása maga után vonja a másik módosítását)
  - A speciális eseteket (pl. adott mező ne látszódjon JSON-ben, körök kezelése) jellemzően a JSON leképezést végző library (by default Jackson) annotációival oldjuk meg → annotációkkal telepakolt entitások

DTO-k alkalmazása

- Előnyök:
  - Az entitások kódja nem duzzad fel annotációkkal
  - Az adatmodell és a REST API szétcsatolt
- Hátrányok:
  - Sok plusz osztály, amelyek gyakran az entitások (részleges) másolatai → nehéz karbantartani
  - A DTO és entitás közti konverzió futási idejű overhead-et jelent
  - A DTO és entitás közti konverziót le kell fejleszteni
    - Library-k segíthetnek, pl. MapStruct, ModelMapper

Spring MVC: hibakezelés

- Legegyszerűbb módszer:
  - ResponseEntity-vel tér vissza a metódusunk, hiba esetén a HTTP státusz kóddal jelzünk vissza

```
@GetMapping("/{id}")
public ResponseEntity<AirportDto> findById(@PathVariable long id)
{

  Airport airport = airportService.findById(id);
  if(airport == null) {
    ResponseEntity.status(HttpStatus.NOT_FOUND).build();
  }
  return ResponseEntity.ok(airportMapper.airportToDto(airport));

}
```

Spring MVC: hibakezelés

- Alternatíva:
  - ResponseStatusException dobása → nem kell a választ ResponseEntity-be csomagolni

```
@GetMapping("/{id}")
public AirportDto findById(@PathVariable long id) {
  Airport airport = airportService.findById(id);
  if(airport == null) {
    throw new
      ResponseStatusException(HttpStatus.NOT_FOUND);
  }
  return airportMapper.airportToDto(airport);
}
```

Spring MVC: hibakezelés

- További gyakori igények:
  - Minden handler metódusra, akár külön controllerekben lévőkre egységes hibakezelés → @(Rest)ControllerAdvice, @ExceptionHandler
    - Tipikusan le is naplózzuk a hibát
  - A státusz kódon túl több részlet visszaadása
    - Spring Boot by default JSON válaszban ad infókat (időbélyeg, exception message)
    - Ha jobban testre akarjuk szabni → saját hibaválasz

Spring MVC: lapozás

- A Spring Data által definiált Pageable típust közvetlenül felvehetjük controller metódus argumentumként, by default page, size és sort nevű query paraméterekből tölti ki nekünk, pl.

```
@GetMapping

public List<EmployeeDto> findAll(@RequestParam Integer minSalary,
@SortDefault("id") Pageable pageable) {

  Page<Employee> employeePage =
employeeRepository.findBySalaryGreaterThan(minSalary, pageable);

  employees = employeePage.getContent();

  System.out.println(employeePage.getTotalElements());

  System.out.println(employeePage.isFirst());

  System.out.println(employeePage.isLast());

  System.out.println(employeePage.hasPrevious());

  System.out.println(employeePage.hasNext());


  return employeeMapper.employeesToDtos(employees);

}
```

Spring MVC

Thymeleaf

Thymeleaf: Általános jellemzők

- Java library, XML/HTML/HTML5 template fájlok kezelésére
- Rugalmasan bővíthető dialektusokkal, pl.
  - Standard Dialect
  - SpringStandard Dialect (OGNL helyett SpringEL az objektumok elérésekor)
- A thymleaf template szabványos XML/HTML, csak th: névtérbe (xmlns:th="http://www.thymeleaf.org") tett *attribútumokkal* bővítve → a statikus fájl template egyszerűen designolható
  - A designolás elvégezhető a template fájlon, nem kell futtatni a szervert, az alkalmazást feltelepíteni, stb.
  - A designer és fejlesztő ugyanazon a fájlon dolgozhat
- Belső működés:
  - Saját DOM implementáció
  - A parszolt template-eket cache-eli
- spring-boot-starter-thymeleaf autokonfigolja

Változó kifejezések

- ${} között, SpringEL kifejezések, leggyakrabban modell objektumok
- Pl. egy handler metódus a modellbe tesz egy person objektumot, és a person.html-re navigál:

```
public String findPerson(
   @RequestParam("personId") long id,
   Map model){
     model.put("person", personService.findById(id));
     return "person";
}
```

- Akkor a person.html template-ben így írjuk ki a nevét (t.f.h. a Person-nak van firstName propertyje):

```
<span th:text="${person.firstName}">Gábor</span>
```

- Ha nyersen nézi valaki (designer) a HTML-t, a Gábor nevet látja
- Ha a szerveren fut és végrehajtódik a template, akkor a dinamikus érték helyettesítődik be

Selection kifejezések

- Szintaxis: *{}
- Egy gyökér objektumhoz relatív property-k elérésére, pl.

```
<div th:object="${book}">
   ...
   <span th:text="*{title}">...</span>
   ...
</div>
```

I18N kifejezések

- #{} között, properties fájlban lévő kulcsot adhatunk meg, pl.

```
<h2 th:text="#{welcome}">Welcome</h2>
```

- A Spring szinten bekonfigolt properties fájlból fogja feloldani, pl.

```
@Bean
public MessageSource messageSource() {

  ReloadableResourceBundleMessageSource messageSource =
        new ReloadableResourceBundleMessageSource();
  messageSource.setBasename("classpath:Messages");
  return messageSource;
}
```

- classpath-on Messages_en.properties, benne

```
welcome=Hello
```

- Messages_hu.properties, benne

```
welcome=Üdvözöljük
```

URL kifejezések

- Szintaxis: @{}
- Oldal-relatív: `@{img/bg.png}` → `img/bg.png`
- Context relatív: `@{/img/bg.png}` → `/myapp/img/bg.png`
- Szerver relatív: `@{~/otherapp/index}` → `/otherapp/index`
- Protokoll relatív: `@{//code.jquery.com/jquery-2.0.1.min.js}`
- Abszolút: `@{http://xy.com/index}`
- Paraméterezés ()-jel, több elválasztása ,-vel:

```
<a href="details.html"
 th:href="@{/order/details(orderId=${o.id})}">View</a>
```

vagy

```
<a href="details.html"
 th:href="@{/order/details/{orderId}(orderId=${o.id})}">
View</a>
```

Form adatok modellhez kötése

- th:action, th:object, th:field

```
<form action="saveCustomer.html" th:action="@{/saveCustomer}"
th:object="${customer}" method="post">

        <input type="hidden" th:field="*{id}" />


        <label for="firstName">First name:</label>

        <input type="text" th:field="*{firstName}" value="John" />


        <label for="lastName">Last name:</label>

        <input type="text" th:field="*{lastName}" value="Wayne" />


        <label for="balance">Balance (dollars):</label>

        <input type="text" th:field="*{balance}" size="10" value="2500" />


        <input type="submit" />

</form>
```

Iteráció

- th:each

```
<table>
      <tr>
        <th>NAME</th>
        <th>PRICE</th>
      </tr>
      <tr th:each="prod : ${prods}">
        <td th:text="${prod.name}">Onions</td>
        <td th:text="${prod.price}">2.41</td>
      </tr>
</table>
```

- Az iteráció álllapotára implicit változó: prodStat, rajta keresztül elérhető:
  - index (0-tól), count (1-től), size,
  - boolean propertyk: even, odd, first, last

Feltételes kifejezések

- th:if:
  - ha false-ra értékelődik ki, nem jelenik meg a tag

```
<span th:if="${prod.price lt 100}"
   class="offer">Special offer!</span>
```

- Negálás:
  - not vagy ! a kifejezés elé
  - th:unless: true esetén nem jelenik meg
- Egyszerűsítések
  - A null false-t jelent
  - true-nak minősül:
    - Szám vagy karakter, ha nem 0
    - String, ezeket kivéve: "false", "off", "no"
    - bármilyen nem-null objektum, ami nem boolean, szám, char vagy String

Feltételes kifejezések

- th:switch, th:case

```
<div th:switch="${user.role}">
  <p th:case="'admin'">User is an administrator</p>
  <p th:case="#{roles.manager}">User is a manager</p>
  <p th:case="*">User is some other thing</p>
</div>
```

- Más th attribútumok értéke is tartalmazhat feltételes kifejezést

```
<tr th:class="${row.even}? 'even' : 'odd'">
```

- : elhagyásával null lesz false esetén

```
<tr th:class="${row.even}? 'alt'">
```

- Default kifejezés (ha null az eredeti) ?:

```
<p>Age: <span th:text="*{age}?:
        '(no age specified)'">27</span>.</p>
```

  - Ennek a rövidítése

```
<p>Age: <span th:text="*{age != null}? *{age} : '(no age
specified)'">27</span>.</p>
```