

Let's win the race together!



Enterprise webalkalmazások Springgel

Spring Web, REST, Validation, Mapper-DTO

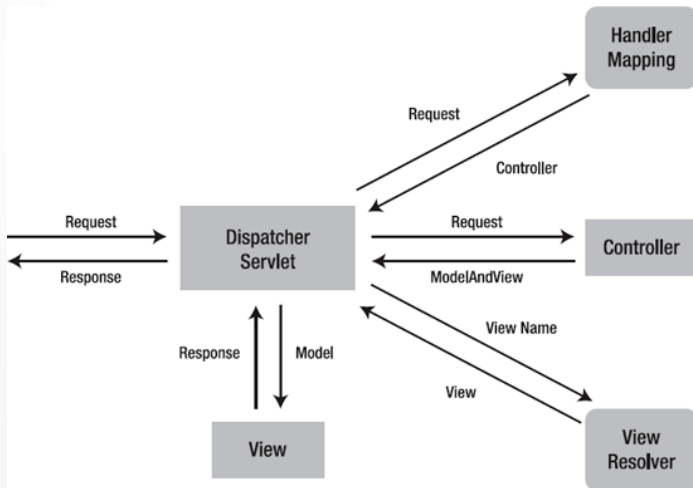
Sulyok Csaba
csaba.sulyok@gmail.com

1. rész

Spring Web

- ▶ A Spring Web és Spring MVC modulok segítenek webalkalmazások készítésében.
- ▶ A **spring-boot-starter-web** függőség segítségével megkapjuk a teljes funkcionalitást.
- ▶ A Spring MVC thin kliensek kiszolgálására alkalmas - sablonmotor segítségével dinamikus tartalmat készít, általában a Thymeleaf motorral.
- ▶ A Spring Web használható az MVC modul nélkül is, amely esetben kiszolgálhatunk HTTP hívásokat sablonok nélkül - RESTful API-k készítéséhez elegendő.
- ▶ Mindkét megközelítés **Servlet**-alapú - a Spring karbantart egy darab **DispatcherServlet** példányt, amely el irányítja a hívásokat a megfelelő beanekhez.
- ▶ Korábbihoz hasonló **war** állományokat is készíthetünk, vagy alternatívan embedded webkonténer futtatását is támogatja a Spring (alapértelmezetten Tomcat). Az **autoconfigure** mechanizmus elindítja a Tomcatet, hogyha a megfelelő webhez tartozó függőségeket classpath-en látja.

Spring DispatcherServlet működése



forrás: <https://howtodoinjava.com/wp-content/uploads/2015/02/Spring-dispatcher-servlet.png>

- ▶ A HTTP hívásokat feldolgozó beaneket nevezzük *Controller*eknek, így annotáljuk őket a szemantikus **@Controller** annotációval. Az annotáció maga nem tesz semmit, de jelzi az osztály szerepét.
- ▶ Ahhoz, hogy a **DispatcherServlet** bizonyos URL-ekhez érkezett hívásokat ide route-oljon, használjuk a következő Spring által nyújtott annotációkat:
 - ▶ **@RequestMapping**
 - ▶ jelzi, hogy HTTP hívások érkezhetnek ide
 - ▶ megadható a HTTP metódus, URL pattern
 - ▶ osztályra vagy metódusra alkalmazható
 - ▶ **@GetMapping**, **@PostMapping** - shorthand a **@RequestMapping**-re már beállított HTTP metódusokkal
- ▶ Ezek tartalmazhatnak dinamikus path paramétereket is a **@PathVariable** változó segítségével

```
@GetMapping("/blogs/{blogId}/comments")
public ResponseObject method(@PathVariable("blogId") Integer blogId) {
    // GET /blogs/42/comments esetén blogId == 42
}
```

- ▶ Egy HTTP lekezelő controller metódus a következő módon téríthet vissza információt a kliensnek:
 - ▶ Egy **ResponseEntity** típusú objektumot térít vissza, ez tartalmazza a válasz teljes információját (státuszkód, headerek, body) - buildert használunk az elkészítésére
 - ▶ Annotáljuk **@ResponseBody**-val a metódust, így a visszatérítési értéke lesz a HTTP válasz body-ja - JSON szerializáció történik alapértelmezetten. A státuszkódot a **@ResponseStatus** annotációval állíthatjuk be.
 - ▶ A **Servletek** által definiált **HttpServletRequest** és **HttpServletResponse** itt is alkalmazhatóak–bármely feldolgozó metódusnak megadhatunk ilyen típusú paramétereket.
 - ▶ Különben egy visszatérített String egy view nevét jelzi, ahova szeretnénk hogy továbbítson a Spring MVC.
- ▶ A HTTP hívás paramétereit vagy a body-ja tartalmát ugyancsak lekérhetjük egy megfelelő típusú paraméterrel, amelyet annotálunk **@RequestParam** vagy **@RequestBody**-val.

```
/**
 * HTTP kérések lekezelésére alkalmas Spring kontroller bean.
 * Minden /general-ra érkező HTTP hívás ide lesz irányítva.
 * Lehet injektálni külső függőségeket bele.
 */
@Controller
@RequestMapping("/withannotations")
public class ControllerWithAnnotations {
    private static final Logger LOG = LoggerFactory.getLogger(ControllerWithAnnotations.class);

    @PostConstruct
    public void postConstruct() {
        LOG.info("Initializing ControllerWithAnnotations");
    }

    // ...
}
```

- ▶ adat visszatérítése `ResponseEntity`-vel, paraméter kibányászása `@RequestParam`-mal

```
@GetMapping
public ResponseEntity handleGet(@RequestParam(value = "id", required = false) String id) {
    LOG.info("Handling GET request, request parameter is {}", id);
    return ResponseEntity.ok("Hello from GeneralSpringController, your id is " + id);
}
```

- ▶ metódus visszatérési értékének body-ban való szerializációja `@ResponseBody`-val

```
@GetMapping
@ResponseBody
public String handleGet(@RequestParam(value = "id", required = false) String id) {
    LOG.info("Handling GET request, request parameter is {}", id);
    return "Hello from GeneralSpringController, your id is " + id;
}
```


▶ adat küldése kérés-body-ban

```
@PostMapping
public ResponseEntity handlePost(@RequestBody String body) {
    LOG.info("Received POST with body {}", body);
    return ResponseEntity.ok().build();
}
```

▶ státusz beállítása @ResponseStatus-szal

```
@PostMapping
@ResponseStatus(HttpStatus.NO_CONTENT)
public void handlePost(@RequestBody String body) {
    LOG.info("Received POST with body {}", body);
}
```

- ▶ hibák fellépése esetén:
 - ▶ visszatéríthetünk különböző státuszú **ResponseEntity** válaszokat
 - ▶ nem engedi általánosítani a visszatérési értéket
 - ▶ dobhatunk egy **kivételt**, majd általános módon kezeljük le a kivételtípust
 - ▶ a kontroller metódus csak sikeres futás esetén térít vissza megadott típusú választ
 - ▶ a Spring Webbel kivételeket kezelhetünk általánosan
- ▶ általános hibakezelés
 1. Egy kontrollerben egy lekezelő metódust **@ExceptionHandler**-rel annotálunk, s megadjuk a kivétel típusát mint paraméter, így abban a kontrollerben minden ilyen típusú kivételt lekezelhetünk.
 2. Ha **több** kontrollerben általánosan szeretnénk hibákat kezelni, a **@ControllerAdvice** komponenstípus minden kontrollerben újrhasználható elemeket definiál.

- ▶ **ControllerWithExceptionHandler** - Ha az **id** paraméter string hosszabb mint 16 karakter, dobunk egy sajátos kivételt.

```
@GetMapping
@ResponseBody
public String handleGet(@RequestParam(value = "id", required = false) String id) {
    if (id != null && id.length() > 16) {
        throw new BadRequestException("ID string is too long");
    }
    return "The provided ID is good";
}
```

- ▶ **BadRequestException**

```
@ResponseStatus(HttpStatus.BAD_REQUEST)
public class BadRequestException extends RuntimeException {
    public BadRequestException(String message) {
        super(message);
    }
}
```

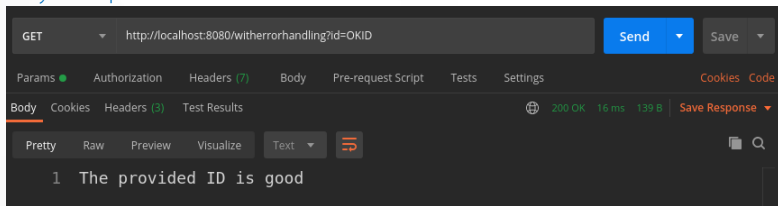
▶ @ControllerAdvice

- ▶ komponenstípus, amely több kontrollerben újrahasználható elemeket definiál.
- ▶ a `handleBadRequest` általánosan lekezeli a bármely kontrollerben fellépő `BadRequestException` típusú kivételeket.
- ▶ alternatívaként használható arra, hogy egyenesen annotáljuk a kivételosztályt

```
@ControllerAdvice
public class GeneralExceptionHandler {

    /**
     * Általános hibakezelő
     * Minden BadRequestException típusú kivételhez közös lekezelőt rendelünk.
     */
    @ExceptionHandler
    @ResponseStatus(HttpStatus.BAD_REQUEST)
    @ResponseBody
    public String handleBadRequest(BadRequestException e) {
        return e.getMessage();
    }
}
```

▶ helyes `id` paraméter

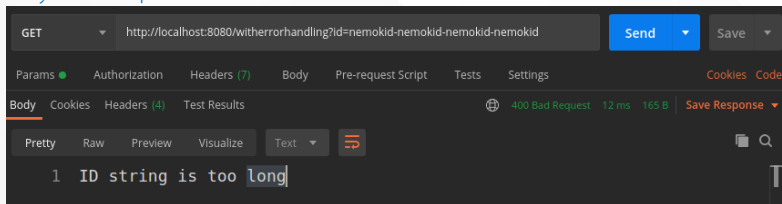


The screenshot shows a web client interface with the following details:

- Method: GET
- URL: `http://localhost:8080/witherrorhandling?id=OKID`
- Status: 200 OK
- Time: 16 ms
- Size: 139 B
- Response body (Pretty):

```
1 The provided ID is good
```

▶ helytelen `id` paraméter



The screenshot shows a web client interface with the following details:

- Method: GET
- URL: `http://localhost:8080/witherrorhandling?id=nemokid-nemokid-nemokid-nemokid`
- Status: 400 Bad Request
- Time: 12 ms
- Size: 165 B
- Response body (Pretty):

```
1 ID string is too long
```

2. rész

REST

- ▶ **REpresentational State Transfer** - Roy Fielding, 2000
- ▶ Egységes body-reprezentáció (általában JSON) mind kérésekben, mind válaszokban
 - ▶ segéd-annotáció: **@RestController** - egy osztályra ráhelyezni egyenértékű azzal, hogy **@Controller**-t használunk, s **minden lekezelő metóduson @ResponseBody**-t
- ▶ Állapotmentes, cache-elhető
- ▶ URL-konvenciók: kollekciók többesszámban, igék nélkül
- ▶ CRUD módszerek HTTP metódusokra térképezve:
 - ▶ **GET** - lekérés
 - ▶ **POST** - hozzáadás (nem idempotens)
 - ▶ **PUT/PATCH** - teljes vagy részleges módosítás (idempotens)
 - ▶ **DELETE** - törlés

GET `http://myhost/api/books`

- ▶ összes bejegyzés lekérése
- ▶ **Figyelem:** az URI az entitás neve többesszámban (nem `/book`, nem `/getBooks`, stb.)
- ▶ konvencionális tömböt térít vissza erőforrásleírásokkal
- ▶ query paraméterekben adhatunk meg szűrési információkat (adattag szerinti szűrés, pagination információ)

```
@RestController
@RequestMapping(value = "/api/books")
public class BookController {

    @Autowired
    private BookService bookService;

    // ...

    @GetMapping
    public Collection<Book> findAllBooks() {
        // hibakezelés
        return bookService.findAllBooks();
    }
}
```


GET <http://myhost/api/books/2>

- ▶ egyedi azonosító esetén nem query paramétert használunk (**nem** `/books?id=2`)
- ▶ nem talált erőforrás esetén **404-es státuszkód** elvárt

```
@GetMapping("/{bookId}")  
public Book findBookById(@PathVariable("bookId") Long bookId) {  
    Book book = service.findBookById(bookId);  
    if (book == null) {  
        throw new NotFoundException(Book.class, bookId);  
    }  
    return book;  
}
```

GET <http://myhost/api/books/2>

- ▶ egyedi azonosító esetén nem query paramétert használunk (**nem** `/books?id=2`)
- ▶ nem talált erőforrás esetén **404-es státuszkód** elvárt
- ▶ általános hibakezelés

```
@ControllerAdvice
public class GeneralExceptionHandler {

    @ExceptionHandler(NotFoundException.class)
    @ResponseStatus(HttpStatus.NOT_FOUND)
    @ResponseBody
    public final void handleNotFound(NotFoundException e) {
    }

    ...
}
```

GET <http://myhost/api/books/2/comments>

- ▶ A 2-es azonosítójú könyvhoz tartozó összes komment
- ▶ Alkalmazható egy-a-többhöz kapcsolatok esetén
- ▶ Nem intuitív, mivel a blog posztok kollekciójából kérünk le külső kulcs szerint. Így lehetne `/comments?bookId=2` is, de ez REST szervereknél nem szokás, mert:
 - ▶ Az endpoint jogosultságok nem változnak query paraméterek alapján (pl. mert a 2-es ID-jú könyv kommentjeit láthatjuk, még nem jelenti azt, hogy mindegyiket láthatjuk)
 - ▶ További egymásbaágyazást akadályoz meg - pl.

GET <http://myhost/api/books/2/comments/4>

```
@GetMapping(value =("/{bookId}/comments")
public Collection<Comment> findCommentsByBookId(@PathVariable("bookId") Long bookId) {
    Book book = bookService.findBookById(bookId);
    if (book == null) {
        throw new NotFoundException(Book.class, bookId);
    }
    return book.getComments();
}
```

POST `http://myhost/api/books`

- ▶ az URL megegyezik a lekérő URL-lel
- ▶ a felhasználó paraméterei a body-ban lesznek ugyanazon formátumban, mint ahogy a **GET** visszaadta
- ▶ egyedi azonosítót **nem** adunk, azt mindig a szerver generálja
- ▶ hibás bemenet esetén megfelelő státuszkód-pl. **400 Bad Request**
- ▶ a válasz státuszkódja konvencionálisan **201 Created** (válaszkor már biztosan létrejött az erőforrás) vagy **202 Accepted** (válaszkor még nem jött létre, de a várakozási sorba bekerült a kérés)
- ▶ a **Location** válaszfejlécben visszatérítjük az URI-t, ahol az újonnan létrehozott erőforrás elérhető (`/api/users/<new_id>`)

`@PostMapping`

```
public ResponseEntity<Book> createBook(@RequestBody Book book) {  
    book = bookService.createBook(book);  
    URI createUri = URI.create("/api/books/" + book.getId());  
    return ResponseEntity.created(createUri).body(book);  
}
```

DELETE `http://myhost/api/books/2`

- ▶ Az egyedi azonosítót használó URI-ra küldjük a **DELETE** parancsot.
- ▶ Hiányzó erőforrás esetén téríthetünk vissza **404**-et, ez nem töri meg az idempotenciát, mivel a rendszer állapota nem változik többszörös hívások esetén, csak a státuszkód.
- ▶ Sikeres törlés esetén visszatéríthető **200** vagy **204 No Content**.

```
@DeleteMapping("/{bookId}")
@ResponseStatus(HttpStatus.NO_CONTENT)
public void deleteBook(@PathVariable("bookId") Long bookId) {
    if (!service.bookExists(bookId)) {
        throw new NotFoundException(Book.class, bookId);
    }
    service.deleteBook(bookId);
}
```

PUT `http://myhost/api/books/2`

- ▶ A **PUT** módszer elmenti a body-ban küldött információt a megadott URI-ra. Ezáltal, ha nem létezik erőforrás még ott, automatikusan létrejön.
- ▶ Emiatt a kérés kötelezően kell tartalmazza a *teljes információt*, ami körülírja az entitást. Megfeleltethetjük ezt a típusú módosítást egy törlés és létrehozás sorozatának.
- ▶ A válasz kezelése ezáltal hasonlít a **POST**-hoz, de mivel itt nem jön létre generált ID (ugyanazon URI definiálja az erőforrást ahova a **PUT**-ot küldjük), a **Location** fejléc nem szükséges, ill. a **204** egy elfogadott helyes státusz.

PATCH `http://myhost/api/books/2`

- ▶ A **PATCH** módszer kiegészíti az URI-n található erőforrást a body-ban megadott információval. Ezáltal a megadott URI-n **kell létezzen** már egy entitás, ellenkező esetben **404**-et küldünk vissza.
- ▶ A kérés tartalmazhat részleges információt, mely esetben csak a megadott kulcsok lesznek frissítve.
- ▶ A válasz lehet **204** üres testtel, vagy **200**-as az erőforrás teljes újdonsült teljes állapotával.

1. Alakítsuk a projektünket Spring webalkalmazássá a web starter segítségével.
2. Készítsük el a **BlogPost** entitás RESTful endpointjait: **findAll**, **findById**, **create**.
3. **Otthoni:** Készítsük el a **BlogPost** teljes CRUD endpointjait, valamint a többi entitását is. Ahol kapcsolat van 2 entitás között, lehessen egyiket elérni a másikon keresztül, pl. `/api/users/42/blogPosts` - a 42-es felhasználóhoz tartozó BlogPostok.

3. rész

A Mapper/DTO minta

- ▶ A korábbi REST endpointok esetén a **Book** modellosztályt alkalmaztuk a REST endpointokon történő (de)szerializáció esetén is.
- ▶ Ez a következő esetekben jelenthet gondot:
 - ▶ Egyes modell mezőket nem szeretnénk feltárni a kliensnek. Pl. a felhasználó **jelszava**.
 - ▶ Egyes mezőket más formátumban küldünk hálózaton, mint ahogyan tároljuk az adatbázisban. Pl. képeket tárolhatunk **byte[]** formájában, de hálózaton base 64 Stringként küldünk.
 - ▶ A modellünk több variánsát tárnánk fel különböző endpointokon. Pl. a **GET /api/books** visszatérítheti az összes könyvet leírás nélkül, míg a **GET /api/books/42** visszatéríti a leírást is.
 - ▶ A bemeneti modellünk (**create**-ben használt) különböző mezőket tartalmaz, mint a visszatérített elemek. Pl. **id**-t a szerverünk generál, ezért **create**-kor nem kellene lehessen elküldjük.
 - ▶ Egyes mezők nem a body-ból, hanem kontextusból következtethetőek. Pl. a bejelentkezett felhasználó hozzárendelése, mint az entitás tulajdonosa.

- ▶ Ezeknek kiküszöbölésére alkalmazzuk a **Data Transfer Object** (DTO) tervezési mintát.
- ▶ A minta szerint létrehozunk különböző Java Beaneket, amelyek különböző helyzetekben reprezentálják a modell entitásunkat (pl. létrehozás, egyszerű lekérés, részletes lekérés).
- ▶ A modell és DTO beanek közötti átalakításhoz alkalmazunk **Mappereket** (vagy **Assembler**eket), amelyek egy Java Bean reprezentációból másikba végeznek átalakítást.
- ▶ Megírhatjuk a saját mapperjeinket (dedikált csomagban), vagy alkalmazhatunk külső library-t, pl.:
 - ▶ `mapstruct`
 - ▶ `modelmapper`
- ▶ Ezek reflection és/vagy annotation parsing segítségével generálják le az átalakítási logikát DTO és mapper között.

A **Book entitás** mellett definiálunk 2 kimenő és 2 bejövő DTO-t:

1. Egyszerű kimenő - **findAll**-ban használatos, nem tartalmazza a leírást.
2. Teljes kimenő - **findById**-ban használatos, tartalmaz mindent.
3. Létrehozás bejövő - tartalmazza a létrehozáshoz szükséges/lehetséges kulcsokat.
4. Módosítás bejövő - tartalmazza a módosításhoz lehetséges kulcsokat.

```
@Data
@Entity
public class Book extends BaseEntity {

    String title;
    String author;
    Integer releaseYear;

    @Column(length = 16)
    String isbn;

    @Lob
    String description;

    @Lob
    byte[] coverImage;
}
```

A **Book** entitás mellett definiálunk 2 kimenő és 2 bejövő DTO-t:

1. **Egyszerű kimenő** - **findAll**-ban használatos, nem tartalmazza a leírást.
2. Teljes kimenő - **findById**-ban használatos, tartalmaz mindent.
3. Létrehozás bejövő - tartalmazza a létrehozáshoz szükséges/lehetséges kulcsokat.
4. Módosítás bejövő - tartalmazza a módosításhoz lehetséges kulcsokat.

```
@Data
@NoArgsConstructor
public class BookReducedDto implements Serializable {
    Long id;
    String title;
    String author;
    Integer releaseYear;
    String isbn;
    String coverImage;
}
```

A **Book** entitás mellett definiálunk 2 kimenő és 2 bejövő DTO-t:

1. Egyszerű kimenő - **findAll**-ban használatos, nem tartalmazza a leírást.
2. **Teljes kimenő** - **findById**-ban használatos, tartalmaz mindent.
3. Létrehozás bejövő - tartalmazza a létrehozáshoz szükséges/lehetséges kulcsokat.
4. Módosítás bejövő - tartalmazza a módosításhoz lehetséges kulcsokat.

```
@Data
public class BookDetailsDto extends BookReducedDto {
    String description;
}
```

A **Book** entitás mellett definiálunk 2 kimenő és 2 bejövő DTO-t:

1. Egyszerű kimenő - **findAll**-ban használatos, nem tartalmazza a leírást.
2. Teljes kimenő - **findById**-ban használatos, tartalmaz mindent.
3. **Létrehozás bejövő** - tartalmazza a létrehozáshoz szükséges/lehetséges kulcsokat.
4. **Módosítás bejövő** - tartalmazza a módosításhoz lehetséges kulcsokat.

```
@Data
public class BookCreationDto implements Serializable {
    String title;
    String author;
    Integer releaseYear;
    String isbn;
    String coverImage;
    String description;
}
```

A **Book** entitás mellett definiálunk 2 kimenő és 2 bejövő DTO-t:

1. Egyszerű kimenő - **findAll**-ban használatos, nem tartalmazza a leírást.
2. Teljes kimenő - **findById**-ban használatos, tartalmaz mindent.
3. Létrehozás bejövő - tartalmazza a létrehozáshoz szükséges/lehetséges kulcsokat.
4. **Módosítás bejövő** - tartalmazza a módosításhoz lehetséges kulcsokat.

```
@Data
public class BookUpdateDto implements Serializable {
    String title;
    String author;
    Integer releaseYear;
    String coverImage;
    String description;
}
```

- ▶ A **mapstruct** megfelelően annotált absztrakt metódusoknak generál testet kompilálási időben.
- ▶ A névből, paraméterekből és annotációban megadott metainformációk alapján generál mappert.
- ▶ A spring **componentModel** megjelöléssel a generált osztály injektálható lesz.
- ▶ A Lombokhoz hasonlóan 2 különböző hatókörű függőségre van szükségünk:

```
dependencies {  
    // ...  
    implementation group: 'org.mapstruct', name: 'mapstruct', version: '1.3.1.Final'  
    annotationProcessor group: 'org.mapstruct', name: 'mapstruct-processor', version: '1.3.1.Final'  
}  
  
@Mapper(componentModel = "spring")  
public abstract class BookMapper {  
  
    @IterableMapping(elementTargetType = BookReducedDto.class)  
    public abstract Collection<BookReducedDto> modelsToReducedDtos(Iterable<Book> models);  
  
    public abstract BookDetailsDto modelToDetailsDto(Book model);  
}
```


DTO mapstruct példa: a generált osztály egy része

```
@Generated(  
    value = "org.mapstruct.ap.MappingProcessor",  
    date = "2020-12-15T17:33:49+0200",  
    comments = "version: 1.3.1.Final, compiler: javac, environment: Java 1.8.0_272 (Private Build)"  
)  
@Component  
public class BookMapperImpl extends BookMapper {  
    // ...  
  
    @Override  
    public BookDetailsDto modelToDetailsDto(Book model) {  
        if ( model == null ) {  
            return null;  
        }  
  
        BookDetailsDto bookDetailsDto = new BookDetailsDto();  
  
        bookDetailsDto.setTitle( model.getTitle() );  
        bookDetailsDto.setAuthor( model.getAuthor() );  
        bookDetailsDto.setReleaseYear( model.getReleaseYear() );  
        bookDetailsDto.setDescription( model.getDescription() );  
  
        return bookDetailsDto;  
    }  
}
```

```
@RestController
@RequestMapping("/api/books")
public class BookController {

    @Autowired
    BookMapper bookMapper;

    @GetMapping
    public Collection<BookReducedDto> findAllBooks() {
        // Összes book megkeresése repository segítségével
        List<Book> books = Arrays.asList(
            new Book("My title", "My author", 1942, "The description is a long text"));
        // Átalakítás DTO-kká (nem tartalmaz leírást)
        return bookMapper.modelsToReducedDtos(books);
    }

    @GetMapping("/{bookId}")
    public BookDetailsDto findBookById(@PathVariable("bookId") Long bookId) {
        // Book megkeresése repository segítségével
        Book book = new Book("My title", "My author", 1942, "The description is a long text");
        // Átalakítás DTO-vá (tartalmaz leírást is)
        return bookMapper.modelToDetailsDto(book);
    }
}
```

4. rész

Bean Validation API

- ▶ A Jakarta Bean Validation API egy Java specifikáció, amely:
 - ▶ lehetőséget ad modellekkel kapcsolatos korlátozások kifejezésére
 - ▶ lehetővé teszi sajátos validátorok írását
 - ▶ API-kat biztosít az objektumok (pl. modell vagy DTO) validálásához
 - ▶ API-kat biztosít metódusok és konstruktorok paramétereinek és visszatérési értékeinek érvényesítésére
 - ▶ a validációs hibákat megjeleníti kivételek formájában
- ▶ Beanek adattagjait (kiterjeszthető) validációs annotációkkal láthatjuk el, pl.:
 - ▶ `@NotNull`, `@Null` - adattagok beállítottsága
 - ▶ `@NotEmpty`, `@Size` - karakterláncok hosszúsága
 - ▶ `@Pattern` - karakterláncok tartalma (regex)
 - ▶ `@Min`, `@Max` - számok
 - ▶ `@Positive`, `@PositiveOrZero`, `@Negative`, `@NegativeOrZero` - számok
 - ▶ `@Future`, `@FutureOrPresent`, `@Past`, `@PastOrPresent` - dátumok
 - ▶ `@Valid` - komplex beanekre helyezhetjük, így rekurzívan a beanben levő adattagok annotációit validálja
- ▶ A Spring Web starter régebbi verziójaival járt tranzitív függőség a csomagra, újabb verzióknál szükséges megadni a *validation startert* (`spring-boot-starter-validation`).

- ▶ A webalkalmazások esetén a bejövő DTO-k (létrehozás/módosítás) validálására helyezzünk hangsúlyt.

```
@Data
public class BookCreationDto implements Serializable {

    @NotEmpty
    @Size(max = 256)
    String title;

    @NotEmpty
    @Size(max = 256)
    String author;

    @Positive
    Integer releaseYear;

    @Pattern(regexp = "[0-9- ]+", message = "Not a valid ISBN number")
    @Size(min = 10, max = 16)
    String isbn;

    @Size(max = 8192)
    String description;
}
```

- ▶ A webalkalmazások esetén a bejövő DTO-k (létrehozás/módosítás) validálására helyezünk hangsúlyt.

```
@Data
public class BookUpdateDto implements Serializable {

    @Size(max = 256)
    String title;

    @Size(max = 256)
    String author;

    @Positive
    Integer releaseYear;

    @Size(max = 8192)
    String description;
}
```

- ▶ A Spring natívan támogatja a validációs annotációkat a kontrollált metódusainak paramétereiben.

```
@GetMapping("/validatePathVariable/{id}")  
public ResponseEntity<String> validatePathVariable(@PathVariable("id") @Min(5) int id) {  
    return ResponseEntity.ok("valid");  
}
```

- ▶ Bejövő DTO validálása:

```
@PostMapping  
@ResponseStatus(code = HttpStatus.CREATED)  
public BookDetailsDto createBook(@RequestBody @Valid BookCreationDto bookDto) {  
    // ha ide elértünk, a body biztosan helyes  
    // ...  
}
```

- ▶ Validációs hibák esetén kivételeket kapunk, melyeket ajánlott megfelelően lekezelni (kliensoldali bemeneti hiba = 400 bad request).

@ControllerAdvice

```
public class ValidationErrorHandler {
```

```
    @ExceptionHandler({ConstraintViolationException.class})
```

```
    @ResponseStatus(HttpStatus.BAD_REQUEST)
```

```
    @ResponseBody
```

```
    public final Iterable<String> handleConstraintViolation(ConstraintViolationException e) {
```

```
        return e.getConstraintViolations().stream()
```

```
            .map(it -> it.getPropertyPath().toString() + " " + it.getMessage())
```

```
            .collect(Collectors.toList());
```

```
    }
```

```
    @ExceptionHandler({MethodArgumentNotValidException.class})
```

```
    @ResponseStatus(HttpStatus.BAD_REQUEST)
```

```
    @ResponseBody
```

```
    public final Iterable<String> handleMethodArgumentNotValid(MethodArgumentNotValidException e) {
```

```
        return e.getBindingResult().getFieldErrors().stream()
```

```
            .map(it -> it.getField() + " " + it.getDefaultMessage())
```

```
            .collect(Collectors.toList());
```

```
    }
```

```
}
```


4. Vezessük be a **mapstruct** függőséget a projektünkbe, majd készítsünk megfelelő DTO-kat a REST API-nk kommunikációjának.
5. Vezessük be a bejövő DTO-k validálását a Jakarta Validation API segítségével.
6. **Otthoni:** Végezzük el a fenti feladatokat a többi entitásra is.
7. **Otthoni:** Böngésszük tovább a full-stack Bibliospring példát:
<https://git.edu.codespring.ro/training-assets/Bibliospring>