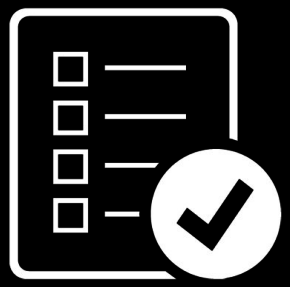
Programming 5

Web API - Implementation

KdG Karel de Grote Hogeschool



- The Java / Spring backend
- The JavaScript frontend
- Model mapping
- Validation
- Serialization
- Error handling

Controllers

- Use `@RestController` instead of `@Controller`

  Types that carry this annotation are treated as controllers where @RequestMapping methods assume @ResponseBody semantics by default.

- Naming convention: suffix ‘Controller’
- Methods:
  - Ideally returns ResponseEntity<>
  - Parameters are mapped from the HTTP req.

`@Controller + @ResponseBody == @RestController`

Per method

Example

As opposed to **@Controller** …

Implicit **@Autowired**

Method has implicit **@ResponseBody** annotation.

```
@RestController
@RequestMapping("/api/books")
public class BooksController {
    private final BookService bookService;

    public BooksController(BookService bookService) {
        this.bookService = bookService;
    }

    @GetMapping("{id}")
    public ResponseEntity<BookDto> getSingleBook(@PathVariable("id") long bookId) {
        var book = bookService.getBook(bookId);
        if (book != null) {
            return ResponseEntity.ok(new BookDto(book.getId(), book.getTitle(),
                    book.getGenre(), book.getRating(), book.getPages()));
        } else {
            return new ResponseEntity<>(HttpStatus.NOT_FOUND);
        }
    }
}
```

Example

URL is **/api/books/{id}**
For example, **/api/books/5**

```
@RestController
@RequestMapping("/api/books")
public class BooksController {
    private final BookService bookService;

    public BooksController(BookService bookService) {
        this.bookService = bookService;
    }

    @GetMapping("{id}")
    public ResponseEntity<BookDto> getSingleBook(@PathVariable("id") long bookId) {
        var book = bookService.getBook(bookId);
        if (book != null) {
            return ResponseEntity.ok(new BookDto(book.getId(), book.getTitle(),
                    book.getGenre(), book.getRating(), book.getPages()));
        } else {
            return new ResponseEntity<>(HttpStatus.NOT_FOUND);
        }
    }
}
```

Path variables

`http://www.domain.tld/api/books/5`

- Part of the path, so must be specified in the path of `@GetMapping`, `@PostMapping`, etc.

  Examples:

  - `@GetMapping("{id}")`
  - `@DeleteMapping("/books/{id}")`

- Place `@PathVariable` with the method parameter.

  Examples:

  - `@PathVariable("id") long bookId`
  - `@PathVariable long id`

Controller methods

- Mapping methods to HTTP verbs can be done using the same annotations as with MVC:
  - `@GetMapping`, `@PostMapping`, `@PutMapping`, `@PatchMapping`, and `@DeleteMapping`

Not part of the HTML standard for **form** submission.

- The '`/api`' prefix can be added using a controller-level `@RequestMapping`
- Multiple methods need to have either …
  - … different paths (`@GetMapping("/unique/path")`)
  - … or different verbs (`@GetMapping`, `@PostMapping`, …)
  - … or different parameters (you may have to name them explicitly)

Example

The name in curly braces **{}** must match either:

- The parameter name
- Or the name of the PathVariable

(Just like with **@RequestParam**)

```
@RestController
@RequestMapping("/api/books")
public class BooksController {
    private final BookService bookService;

    public BooksController(BookService bookService) {
        this.bookService = bookService;
    }

    @GetMapping("{id}")
    public ResponseEntity<BookDto> getSingleBook(@PathVariable("id") long bookId) {
        var book = bookService.getBook(bookId);
        if (book != null) {
            return ResponseEntity.ok(new BookDto(book.getId(), book.getTitle(),
                    book.getGenre(), book.getRating(), book.getPages()));
        } else {
            return new ResponseEntity<>(HttpStatus.NOT_FOUND);
        }
    }
}
```

ResponseEntity

- A returned value is mapped to the body of the returned HTTP message (body of the response)
  - Thanks to `@ResponseBody` (`@RestController`)
- ResponseEntity essentially adds an **HTTP status code** to the response
- Use a **DTO**!
- Examples:

```
return ResponseEntity.ok(book);

return new ResponseEntity<>(HttpStatus.NOT_FOUND);

return new ResponseEntity<>(book, HttpStatus.CREATED);
```

Status code

Body

HTTP request with data in the body

- In combination with `@PostMapping`, `@PatchMapping`, and `@PutMapping`
- The client wants to add or update a record
- Add `@RequestBody` to the parameter
- Use a **DTO**!
- Example:

```
@PostMapping
public ResponseEntity<BookDto> createNewBook(
                        @RequestBody NewBookDto bookDto) {
```

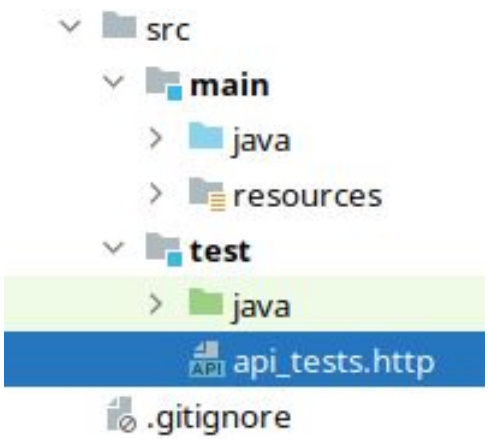
Example

```
@RestController
@RequestMapping("/api/books")
public class BooksController {
    private final BookService bookService;

    public BooksController(BookService bookService) {
        this.bookService = bookService;
    }

    @PostMapping
    public ResponseEntity<BookDto> createNewBook(
                                @RequestBody @Valid NewBookDto bookDto) {
        var newBook = bookService.addBook(bookDto.getTitle(), bookDto.getGenre(),
                                          bookDto.getPages());
        return new ResponseEntity<>(
                new BookDto(newBook.getId(), newBook.getTitle(), newBook.getGenre(),
                    newBook.getRating(), newBook.getPages()),
                HttpStatus.CREATED);
    }
}
```

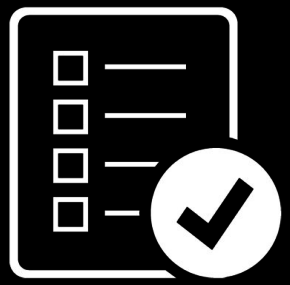
Two different DTO types. Why? 🤔

Exceptions should be handled using controller advice.

REST and HTTP in IntelliJ



- Excellent tool for manual testing
  - Very close to the HTTP protocol
  - (protocol version can be omitted)
- Use the **`.http`** file extension
- File can be added to the git repository
- Evaluation: you must be able to read and write HTTP messages
  - Include **`Accept`** and **`Content-Type`** whenever appropriate!
  - You can ignore other headers for now.



- The Java / Spring backend
- The JavaScript frontend
- Model mapping
- Validation
- Serialization
- Error handling

JavaScript files

- Place JavaScript files in **`resources/static/js`**

  (… for now)

  Never write your JS code inline!

- Reference the script *only* on those pages where it needs to be executed
  - Use **`defer`** if the DOM needs to be loaded before execution of the script.

fetch


```
try {
    const resp = await fetch(`/api/books/${getBookId()}/authors`,
        {
            headers: {
                Accept: "application/json"
            }
        });

    if (resp.status !== 200) {
        // Handle error
    } else {
        const authors = await resp.json();
        showAuthors(authors);
    }
} catch (exc) {
    // Handle error
}
```

Don't forget the header(s)

Can only await in an **async** function.

Handle additional status codes if applicable

fetch - POST

This object contains all of the request options.

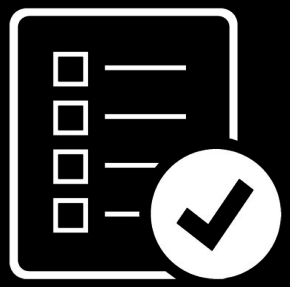
```
await fetch('/api/publishers', {
    method: 'POST',
    headers: {
        "Accept": "application/json",
        "Content-Type": "application/json"
    },
    body: JSON.stringify({
        name,
        yearFounded
    })
})
```

Don't forget the headers! "Content-Type" *must* be quoted since a dash is not allowed for an identifier.

This object contains the payload (=HTTP message body).
This is shorthand notation for `{ name: name, ... }`

Async

- JS functions can be declared as async
- It makes them return a Promise
- Example (check the console inside codepen)
- You can use await in an async function or at the top level of a module



- The Java / Spring backend
- The JavaScript frontend
- **Model mapping**
- **Validation**
- **Serialization**
- **Error handling**

Model mapping

- DTO = Data Transfer Object (REST API)
- VM = View Model (MVC)
- Use a library for automatic mapping
  - MapStruct (https://mapstruct.org/)
  - **Recommendation:** Use MapStruct for trivial mappings. Write custom mapping logic yourself for more complex mappings.

```
implementation("org.mapstruct:mapstruct:1.6.3")
annotationProcessor("org.mapstruct:mapstruct-processor:1.6.3")
```

Model mapping

```
@Mapper(componentModel =
                MappingConstants.ComponentModel.SPRING)
public interface IssueMapper {
    IssueDto toIssueDto(Issue issue);
    List<IssueDto> toIssueDtoList(List<Issue> issues);
}
```

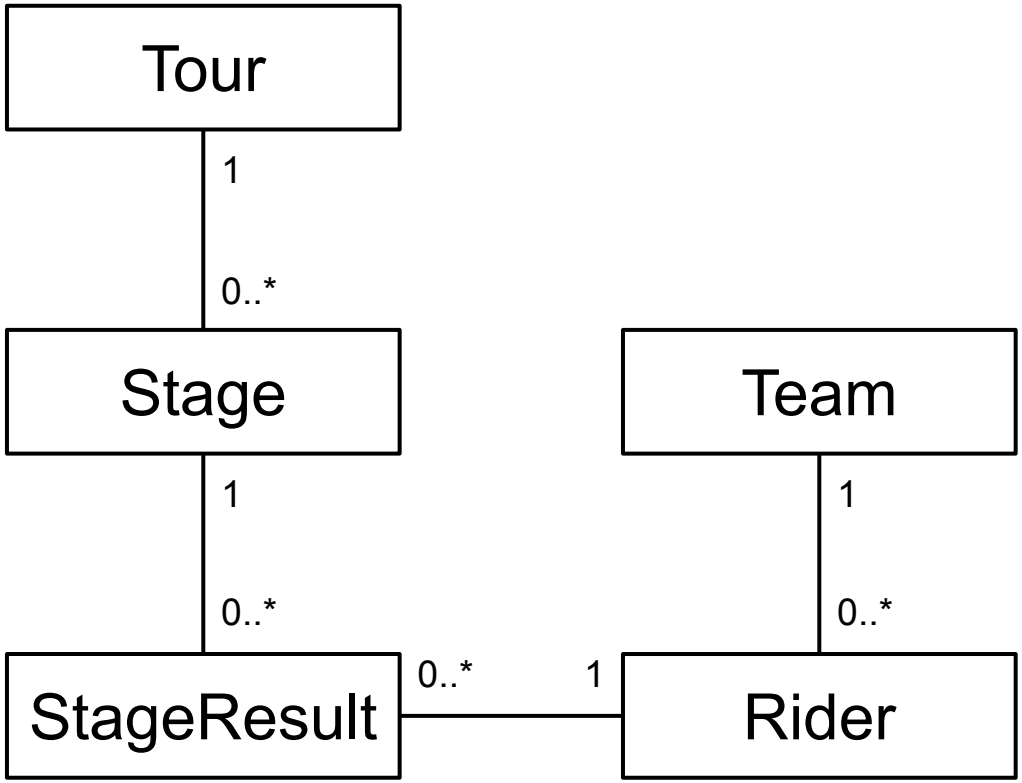
Enables Spring dependency injection for this mapper / interface.

```
return ResponseEntity.ok(issueMapper.toIssueDto(issue));
```

In this example, consider **`issueMapper`** to be an "autowired" dependency.

View models: example

- Domain:



View models: example

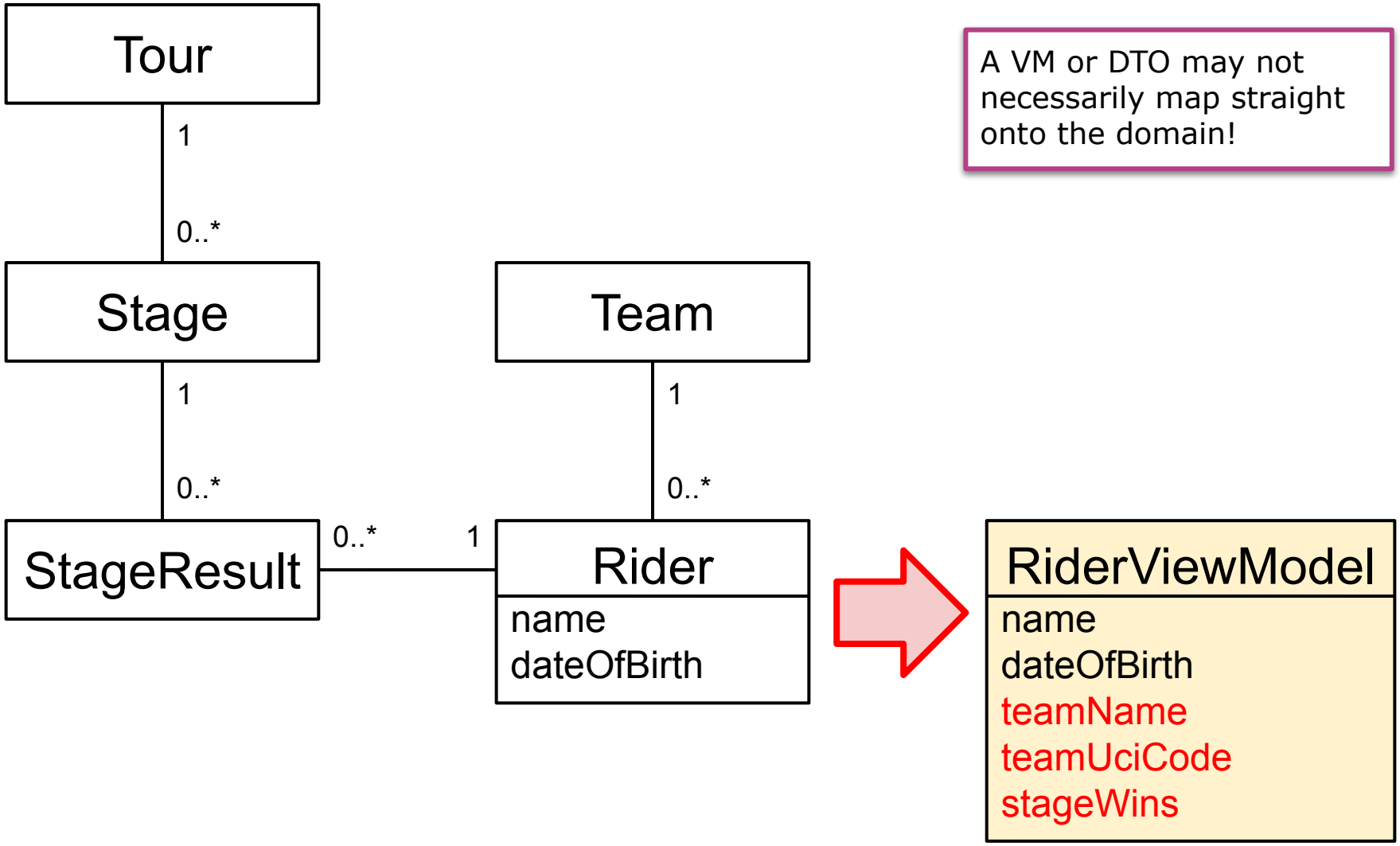
Name	Date of birth	Team Code	Team Name	Stage Wins
Thomas De Gendt	1986-11-06	LTS	Lotto–Soudal	1
Jelle Vanendert	1985-02-19	LTS	Lotto–Soudal	0
Tim Wellens	1991-05-10	LTS	Lotto–Soudal	0
Marcel Kittel	1988-05-11	EQS	Etixx–Quick-Step	1
Tony Martin	1985-04-23	EQS	Etixx–Quick-Step	1
Zdeněk Štybar	1985-12-11	EQS	Etixx–Quick-Step	1

Rider

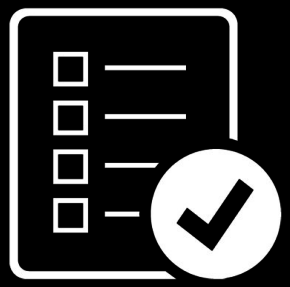
Team

???

View models: example



A VM or DTO may not necessarily map straight onto the domain!



- The Java / Spring backend
- The JavaScript frontend
- Model mapping
- **Validation**
- **Serialization**
- **Error handling**

Validation

- We'll continue to use Hibernate Validator
- We can use the annotations of the Bean Validation Framework:
  - Package: **`jakarta.validation.constraints`**
  - Add **`@Valid`** to the controller method's parameter(s)
  - Add validation annotations to the attributes of the VMs and DTOs:

    `@NotNull`, **`@Size`**, **`@Positive`**, …
- Will trigger status code 400 (Bad Request)

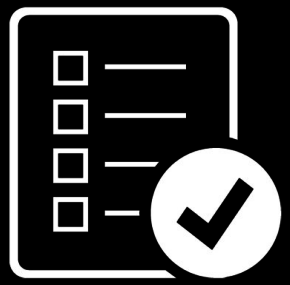
```
implementation "org.springframework.boot:spring-boot-starter-validation"
```

gradle

Validation

```
@PostMapping
public ResponseEntity<BookDto> createNewBook(
                        @RequestBody @Valid NewBookDto bookDto) {
    // ...


public class NewBookDto {
    @NotNull
    @Size(min = 3, max = 50)
    private String title;

    @NotNull
    private Genre genre;

    @Positive
    private int pages;

    // ...
```

https://www.baeldung.com/spring-boot-bean-validation



- The Java / Spring backend
- The JavaScript frontend
- Model mapping
- Validation
- **Serialization**
- **Error handling**

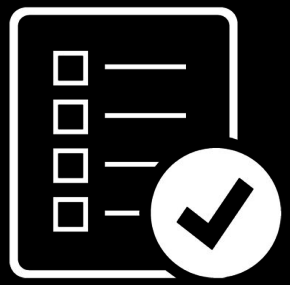
Content negotiation

- By default, Spring Boot uses Jackson for (de)serialization
  - JSON support is included
  - XML support is *not* included
  - To include support for XML:

```
implementation 'com.fasterxml.jackson.dataformat:jackson-dataformat-xml'
```

gradle

Baeldung



- The Java / Spring backend
- The JavaScript frontend
- Model mapping
- Validation
- Serialization
- **Error handling**

Error handling

- Use **@ControllerAdvice** to handle exceptions
- Differentiate between MVC and Web API requests

```
@ControllerAdvice
public class Error Handling {
   @ExceptionHandler(Exception.class)
   public Object onError(Exception e, HttpServletRequest request) {
       if (request.getRequestURI().startsWith("/api")) {
           return ResponseEntity
                   .status(HttpStatus.INTERNAL_SERVER_ERROR)
                   .body(new ErrorDto(e.getMessage()));
       }
       final ModelAndView modelAndView = new ModelAndView(
               "error",
               HttpStatus.INTERNAL_SERVER_ERROR
       );
       modelAndView.addObject("message", e.getMessage());
       return modelAndView;
   }
}
```

It's recommended to use a more specific type (possibly a custom exception).

If you use more specific exception types, you can use more specific status codes as well.