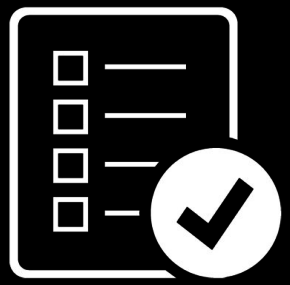
Programming 2.3

Web API - week 1

KdG Karel de Grote Hogeschool



- Introduction to 2.3
- Recap Programming 2.1
- Web API, AJAX, and REST
- The Java / Spring backend
- The JavaScript frontend

Programming 2.3

- ECTS

  https://bamaflexweb.kdg.be/BMFUIDetailxOLOD.aspx?a=117566&b=1&c=2

- 6 ECTS credits
- Lars Willemsens

  lars.willemsens@kdg.be and

- Application of software development principles, practices and APIs in a project
  - Spring (REST / Security / Testing)
  - Research

Programming 2.3

- Lectures:
  - Introduction and background on new concepts
  - Live demonstrations and application of these concepts
- Canvas:
  - Slides
  - Assignments
  - Links to sample code

Programming 2.3

- Evaluation:
  - Project
  - Examination Interview on the submitted project
- **Low-stakes (2 x 20%)**
  - Evaluation of the project code halfway through the 3rd and 4th terms (after 3rd and 9th week)
- **High-stakes (2 x 30%)**
  - Evaluation of the project code and an Examination Interview after both the **3rd and 4th terms** (after 6th and 12th week)

Week 1	
Week 2	
Week 3	
Spring break	
Week 4	Low-stakes assessment (20%)
Week 5	
Week 6	
Exam	**High-stakes assessment (30%)**
Easter holidays	
Easter holidays	
Week 7	(no class on Easter Monday)
Week 8	
Week 9	
Week 10	Low-stakes assessment (20%)
Week 11	
Week 12	
Study week	
Exam	**High-stakes assessment (30%)**

Programming 2.3

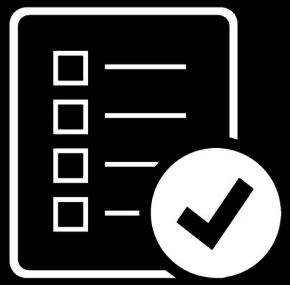
- **Low-stakes (2 x 20%)**
  - Your grade will **not** be taken into account if the grade for the **subsequent** high-stake is higher
  - Also called “nothing to lose”
  - Skipping a low-stakes assessment means discarding valuable feedback
  - Participating in a low-stakes assessment means securing your grade and taking pressure off the weeks ahead
- **High-stakes (2 x 30%)**
  - A high-stakes assessment can effectively overrule a poor or missing low-stakes assessment … and can then count for 50%!

Schedule for term 3

Week 1	<u>Web API</u> ● AJAX ● REST ● Implementation using Spring framework ● Frontend: npm, webpack, fetch
Week 2	
Week 3	
Week 4	<u>Security</u> ● Form login, cookies, … ● Implementation using Spring framework ● Users & Roles
Week 5	
Week 6	

Schedule for term 4

Week 7	<u>Backend Testing</u> ● Spring Testing (Unit testing, MVC, Web API, Security) ● Mocking
Week 8	
Week 9	<u>Research</u> ● GraphQL ● Android ● Websockets ● …
Week 10	
Week 11	
Week 12	



- Introduction to 2.3
- Recap Programming 2.1
- Web API, AJAX, and REST
- The Java / Spring backend
- The JavaScript frontend

Recap Programming 2.1

streams → lambda's
Gradle, Generics, JSON (Gson),

SPRING
DI, Annotations, Thymeleaf,
3-layered arch., Hibernate

Recap Programming 2.1

SPRING

DI container

@ComponentScan

@Service
@Repository
@Controller
@Configuration

} @Component
does exist.

↳ @Bean method exist.

Recap Programming 2.1

SPRING MVC

@Controller class annot.

@ExceptionHandler @ControllerAdvice

@GetMapping @PostMapping

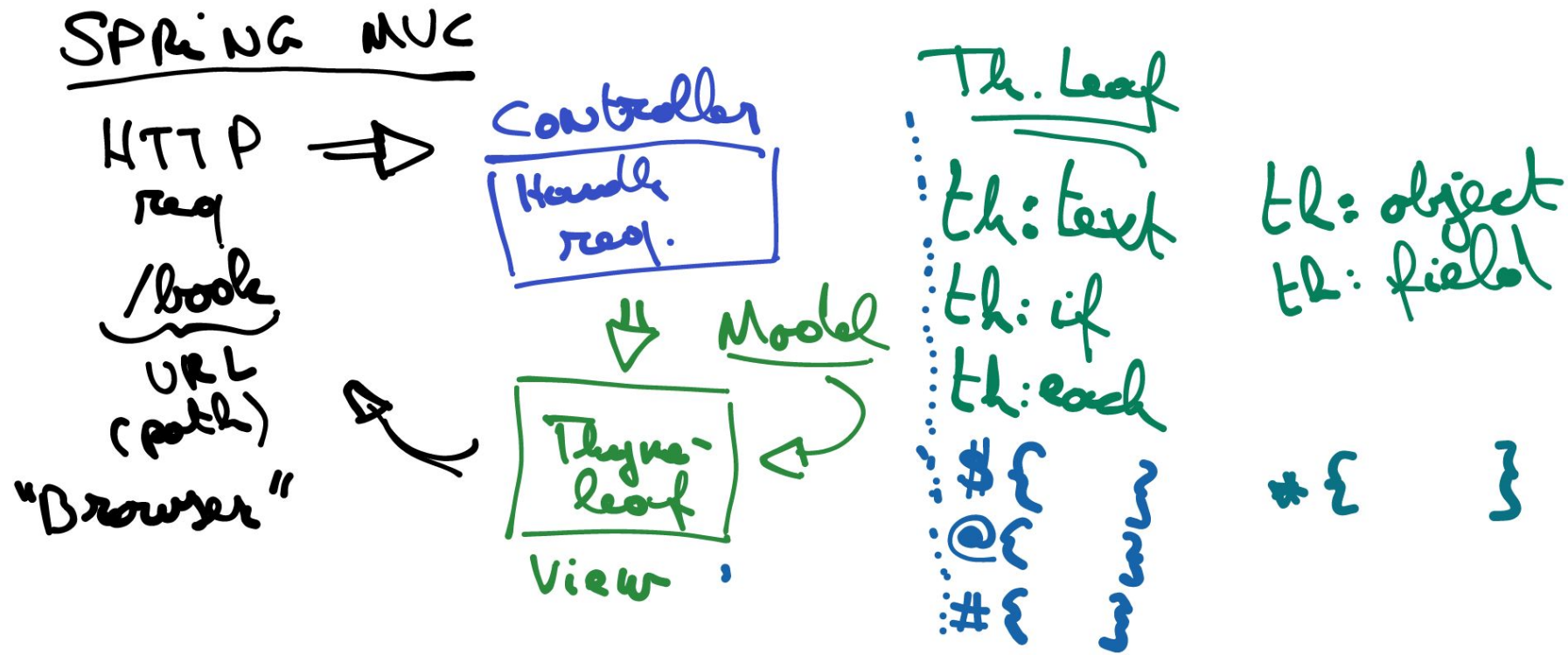
@RequestMapping

@PathVariable Form or ?id=5 ⇒ @RequestParam

@Min @Max @NotNull @NotEmpty

@PastOrPresent + custom valid.

Recap Programming 2.1



Recap Programming 2.1

Hibernate

@Entity
@Table

@Column
@Id
@Generated

@Query HQL

- Eager / Lazy loading

Spring profiles

application.properties
@Profile

@OneToMany
@ManyToOne
@OneToOne @ManyToMany

Publisher :: books
Book :: publisher

Recap Programming 2.1

Bootstrap

Container

row → 12-column

col-6 col-6

p-3 pt-3 pg-3

↳ t + b

ps-3 } start + end
e

↳ not sure!!

table

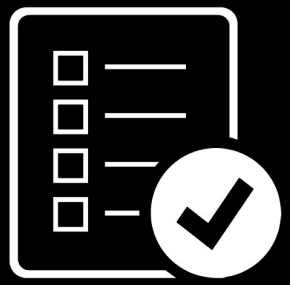
row ....

col-lg-X

md

sm

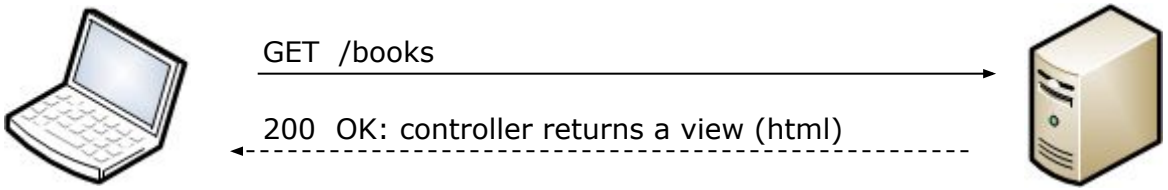
xl



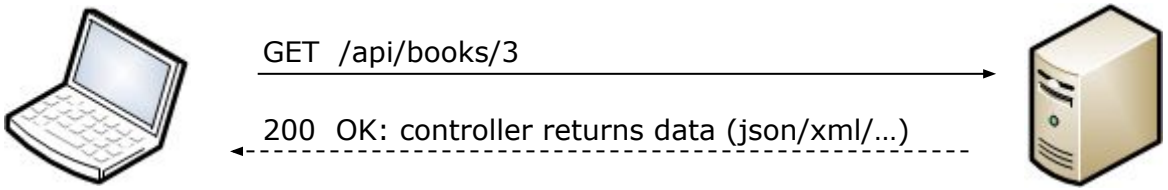
- Introduction to 2.3
- Recap Programming 2.1
- **Web API, AJAX, and REST**
- **The Java / Spring backend**
- **The JavaScript frontend**

MVC vs Web API

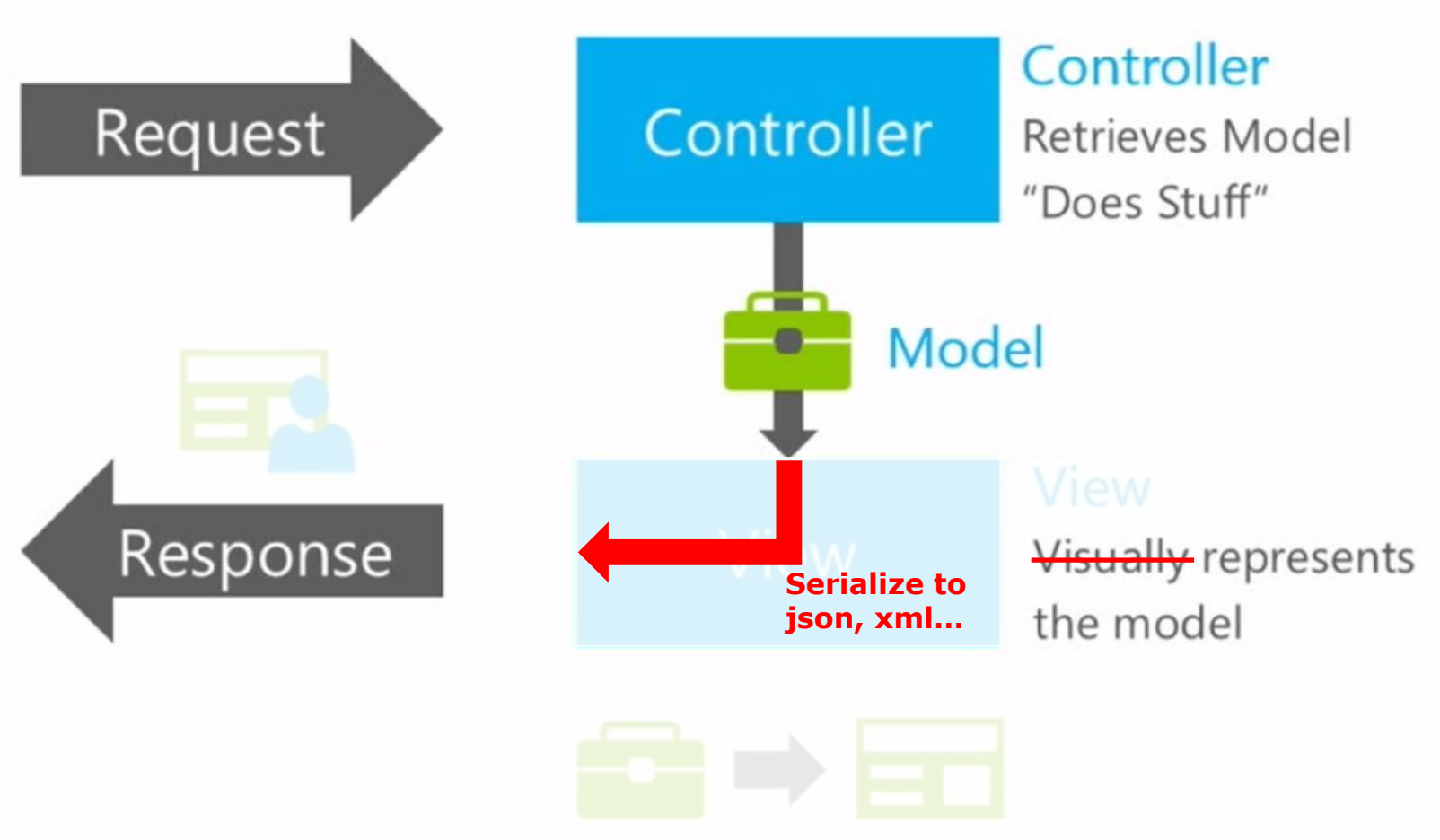
MVC: working with web pages



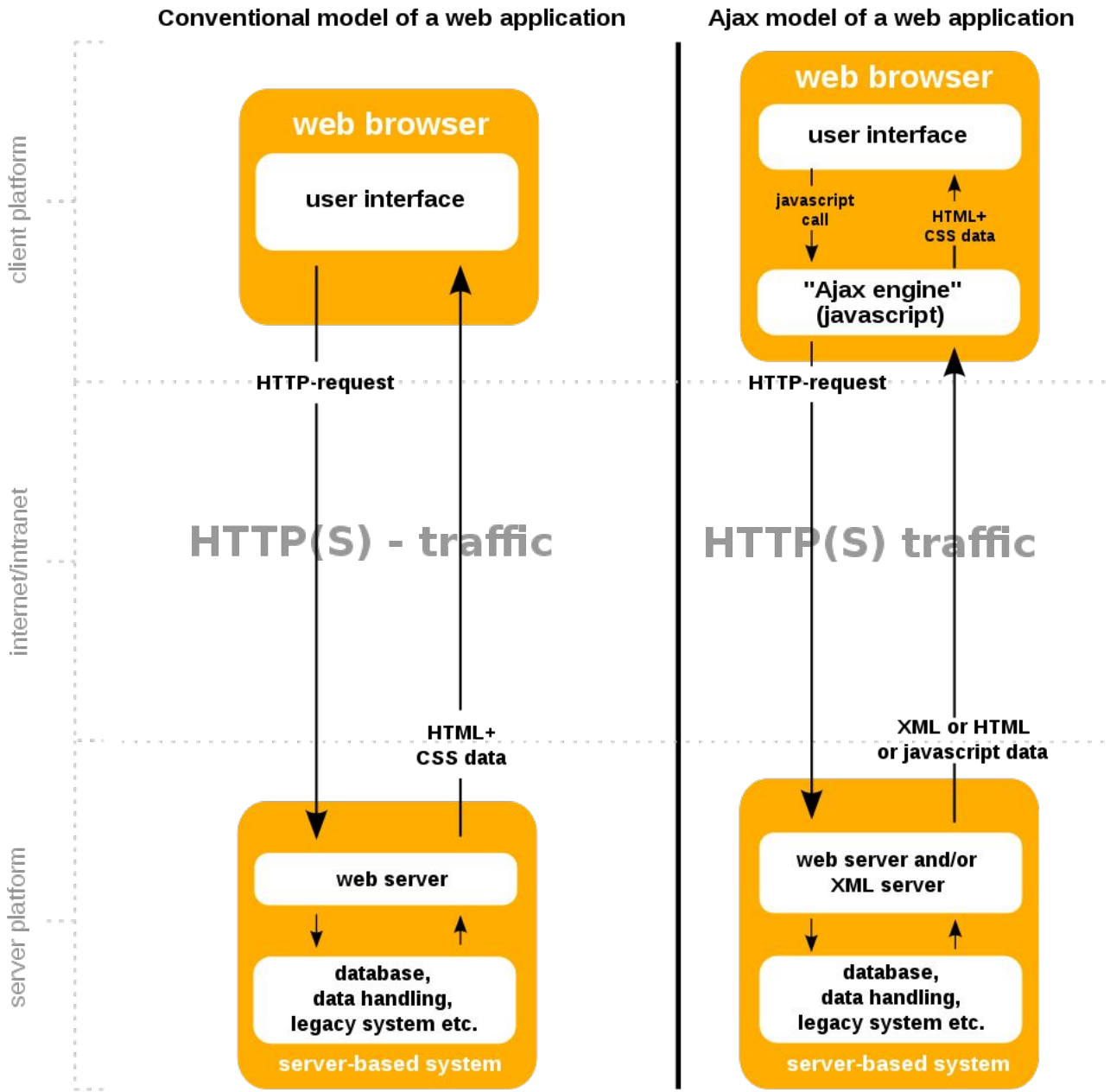
Web API: working with data



MVC vs Web API



AJAX


Conventional model of a web application
Ajax model of a web application
client platform
internet/intranet
server platform
web browser
user interface
HTTP-request
HTTP(S) - traffic
HTML+
CSS data
web server
database,
data handling,
legacy system etc.
server-based system
web browser
user interface
javascript
call
HTML+
CSS data
"Ajax engine"
(javascript)
HTTP-request
HTTP(S) traffic
XML or HTML
or javascript data
web server and/or
XML server
database,
data handling,
legacy system etc.
server-based system

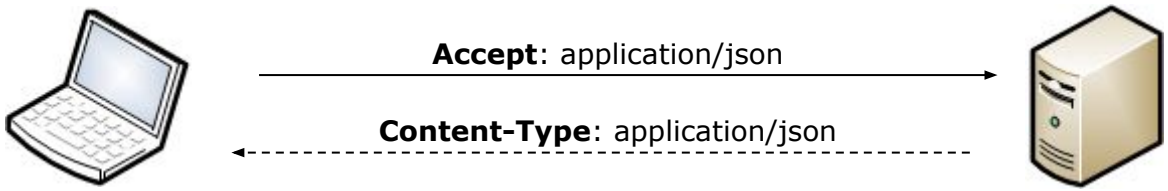
AJAX

- Asynchronous JavaScript and XML
- XML / JSON / …
- Technique enabled by web browsers (and, of course, the servers they're connecting to)

REST

- Architecture / set of conventions and best practices
- CRUD-actions
- Tied to the **HTTP** protocol
- **Content negotiation**
  - HTTP request header
    - **`Accept`**
  - HTTP response header
    - **`Content-Type`**

These are two headers that you should include whenever applicable.



REST

- Representational state transfer
  http://stackoverflow.com/questions/671118/what-exactly-is-restful-programming

Nouns	Verbs			
HTTP	**GET (*)**	**POST**	**PUT(*)**	**DELETE(*)**
Collection i.e.: /books/	Retrieve a list of all elements	Create a new element		
Element i.e.: /books/123	Retrieve a specific element		Replace a specific element	Delete a specific element
CRUD	**Read**	**Create**	**Update**	**Remove**

(*) Idempotent: https://en.wikipedia.org/wiki/Idempotence

REST

- Use of **HTTP status codes**:

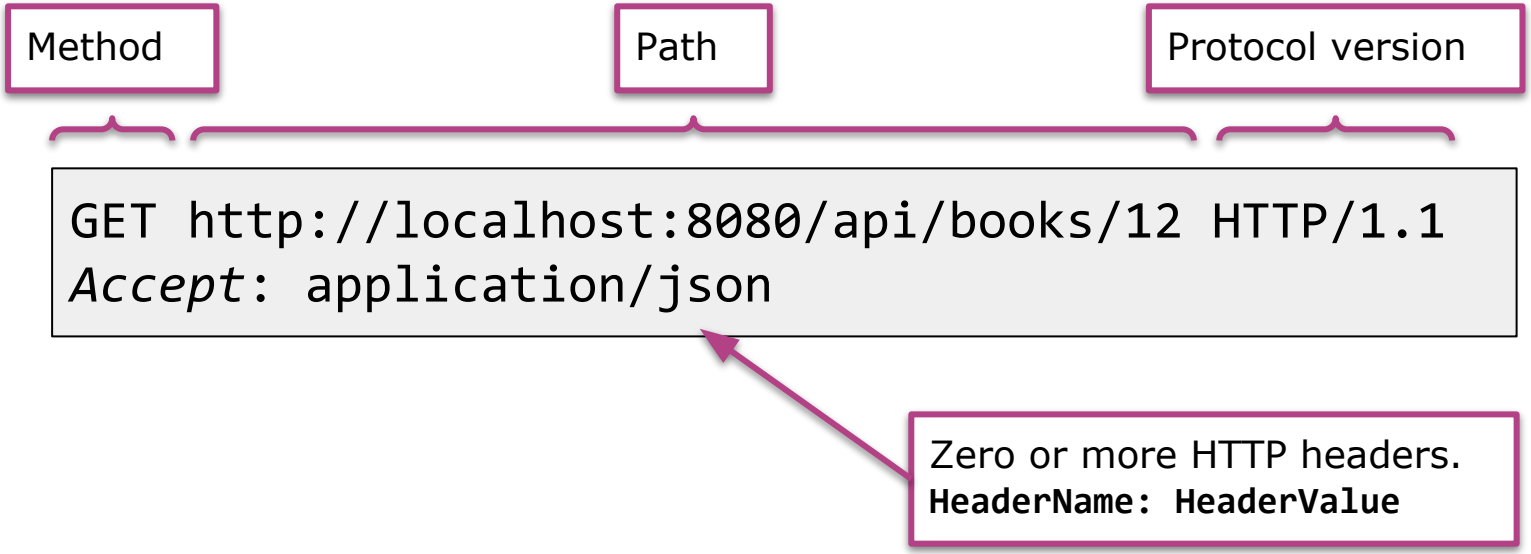
We'll encounter and transmit these ones regularly.

200	OK
201	Created
204	No Content
302	Found (Redirect)
400	Bad Request
401	Unauthorized
403	Forbidden
404	Not Found
405	Method Not Allowed
409	Conflict
500	Internal Server Error

- Place actual resources in the HTTP message body

REST - GET Example

- Request



- Additional data can be transmitted in the message body (see next couple of slides)

REST - GET Example

- Request

```
GET http://localhost:8080/api/books/12 HTTP/1.1
Accept: application/json
```

We "accept" a certain response…

- Response

```
HTTP/1.1 200
Content-Type: application/json
Transfer-Encoding: chunked
Date: Fri, 04 Feb 2022 08:36:11 GMT
Keep-Alive: timeout=60
Connection: keep-alive

{
    "id": 12,
    "title": "Black House",
    "genre": "HORROR",
    "rating": null,
    "pages": 700
}
```

Protocol version and status code

HTTP headers

Mandatory empty line

Message body

REST - POST Example

- Request

```
POST http://localhost:8080/api/books HTTP/1.1
Accept: application/json
Content-Type: application/json

{
      "title": "My First Book",
      "genre": "MYSTERY",
      "pages": 120
}
```

Mandatory empty line

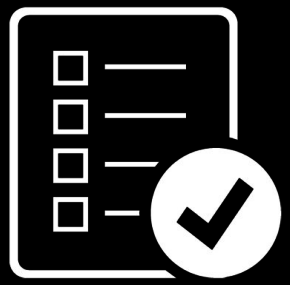
Message body

- Response

```
HTTP/1.1 201
Content-Type: application/json
Transfer-Encoding: chunked
Date: Fri, 04 Feb 2022 09:06:25 GMT
Keep-Alive: timeout=60
Connection: keep-alive

{
      "id": 13,
      "title": "My First Book",
      "genre": "MYSTERY",
      "rating": null,
      "pages": 120
}
```

This time, **`Content-Type`** is in *both* the request and the response



- Introduction to 2.3
- Recap Programming 2.1
- Web API, AJAX, and REST
- **The Java / Spring backend**
- **The JavaScript frontend**

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);
         }
     }
}
```

Corresponds to the example of this slide.

REST resources

- REST is 'resource-oriented' , not 'action-oriented'!
- The url(~uri) uniquely identifies a resource and not an action that is to be executed
- Resources within URIs are expressed in **plural**
- The action is determined by the HTTP verb (get, post, put or delete)

REST URLs

- Convention: prefix '**api**' and **lowercase**
  - **`http://www.domain.tld/api/...`**
- Examples:
  - **`http://www.domain.tld/api/books`**
    - All books
  - **`http://www.domain.tld/api/books/5`**
    - Book with ID 5
  - **`http://www.domain.tld/api/books/5/authors`**
    - All authors of the book with ID 5

REST URLs

- Request parameters can be used as well
- Example:
  - `http://www.domain.tld/api/books?format=pdf`
- Try to avoid request parameters when filtering by related entities.

  For example: "*All authors of the book with ID 5*"
  - Use: `http://www.domain.tld/api/books/5/authors`
  - Not: ~~`http://www.domain.tld/api/authors?bookId=5`~~

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);
         }
     }
}
```

Controller methods

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

Method parameters

- Query parameters (or request parameters)
  - Used often with MVC
  - `http://www.domain.tld/api/books?format=pdf`

Its name is visible, even in the URL.

- Path variables
  - Used often with REST to identify a resource
  - `http://www.domain.tld/api/books/5`

- Both query parameters and path variables are **part of the URL**.
  - Used for filtering or identifying a resource
  - Actual records are sent with the **body**

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`

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` 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 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 can be handled using controller advice.

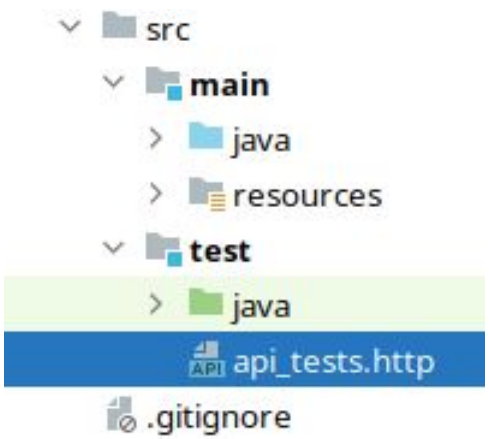
Corresponds to the example of this slide.

HTTP status codes

- We'll use these status codes:

200	OK	Success (*only* when the ones below are n/a)
201	Created	A new record has been created
204	No Content	Nothing to return (is different from Not Found!)
302	Found (Redirect)	Handled by Spring MVC (view: `"redirect: … "`)
400	Bad Request	*Usually* handled by Spring (validation, …)
401	Unauthorized	Handled by Spring
403	Forbidden	Handled by Spring
404	Not Found	Record or page was not found
405	Method Not Allowed	Handled by Spring
409	Conflict	Inconsistency or incorrectness
500	Internal Server Error	Should not occur!

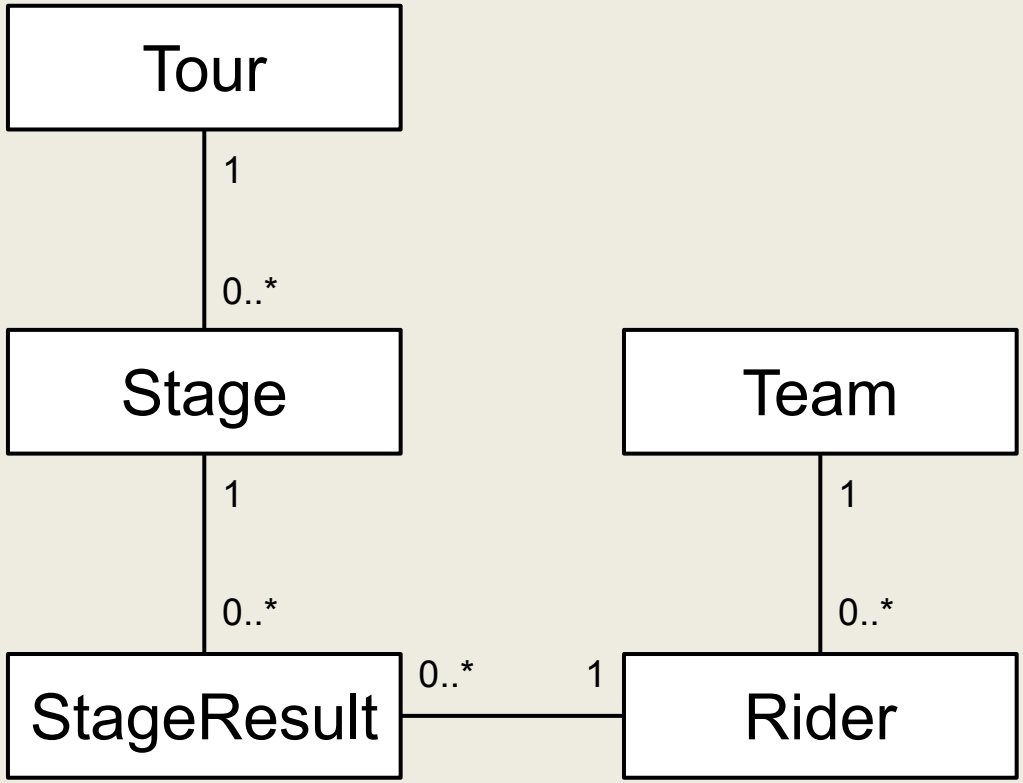
REST and HTTP in IntelliJ



- 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
- You must be able to read and write HTTP messages.
  - Include **`Accept`** and **`Content-Type`** whenever appropriate!
  - You can ignore other headers for now.

Assignment: Cycling

- Start from the code on GitLab.
  - https://gitlab.com/kdg-ti/programming-2.3/assignments/cycling
- Domain:



Assignment: Cycling

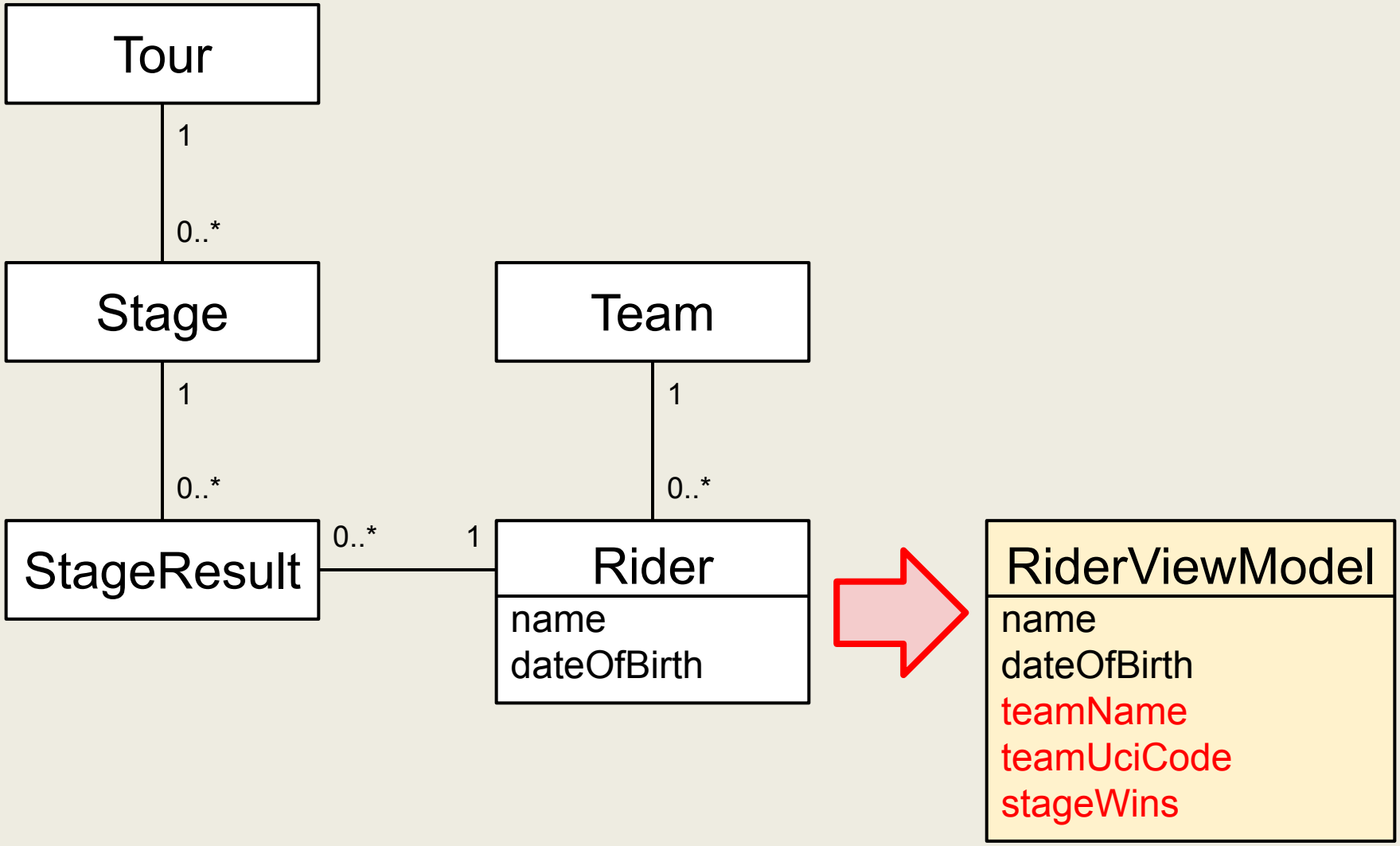
Name	Date of birth	Team Code	Team Name	Stage Wins
Thomas De Gendt	1986-11-06	LTS	Lotto–Soudal	1
Jelle Vanendert	1985-02-19	LTS	Lotto–Soudal	0
Tim Wellens	1991-05-10	LTS	Lotto–Soudal	0
Marcel Kittel	1988-05-11	EQS	Etixx–Quick-Step	1
Tony Martin	1985-04-23	EQS	Etixx–Quick-Step	1
Zdeněk Štybar	1985-12-11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

Rider

Team

???

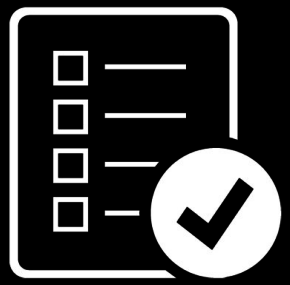
Assignment: Cycling



Assignment: Cycling

- Check out the assignment description on Canvas (click the logo! 🙂)

- Continue to the next section of the slides when you see "*Implement the necessary UI changes*"



- Introduction to 2.3
- Recap Programming 2.1
- Web API, AJAX, and REST
- The Java / Spring backend
- **The JavaScript frontend**

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


```
fetch(`/api/books/${getBookId()}/authors`,
    {
        headers: {
            Accept: "application/json"
        }
    })
    .then(resp => {
        if (resp.status !== 200) {
            // Handle error
        } else {
            return resp.json();
        }
    })
    .then(showAuthors)
    .catch(reason => {
        // Handle error
    });
```

Don't forget the header(s)

Handle additional status codes if applicable

Don't write `showAuthors()`! Why? 🤔

fetch with async/await


```
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
}
```

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

Assignment: Cycling

- Check out the assignment description on Canvas

- Complete the first part of the assignment

- Then, continue from where it says "*Implement the necessary UI changes*"