2025 — 2026

1TI – 1ACS



UCLL HOGESCHOOL

Back-End Development

Spring Boot — Intro and HTTP API GET

N. Jacobs, J. Pieck, E. Steegmans, B. Van Impe, S. Van Peborgh, V. Witters

CONTENTS

- What is a back-end?
- Spring Boot
- HTTP GET

Back-end

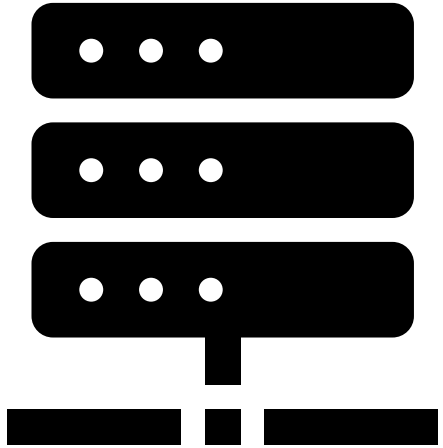
- What is a back-end?
- What is an HTTP API?
- Layered Architecture

WHAT IS A BACK-END?

FRONT END



BACK END



WHAT IS A BACK-END?

- Front-end applications need data to function correctly
  - This data is often dynamic and cannot be bundled together with the front-end
- This data is retrieved from another software application
  - This application functions as a *back-end* for the frontend
- A back-end is a software application that
  - Stores and retrieves data (most of the time using a database)
  - Guards application rules and constraints
  - Enables the user-facing applications to function correctly

WHAT IS A BACK-END?

- A back-end provides a set of operations that a frontend can use, for example:
  - “Return all users registered in the system”
  - “Store a new order in the database”
  - “Update the information of a customer”
  - …

- The full set of operations that a backend provides is called the

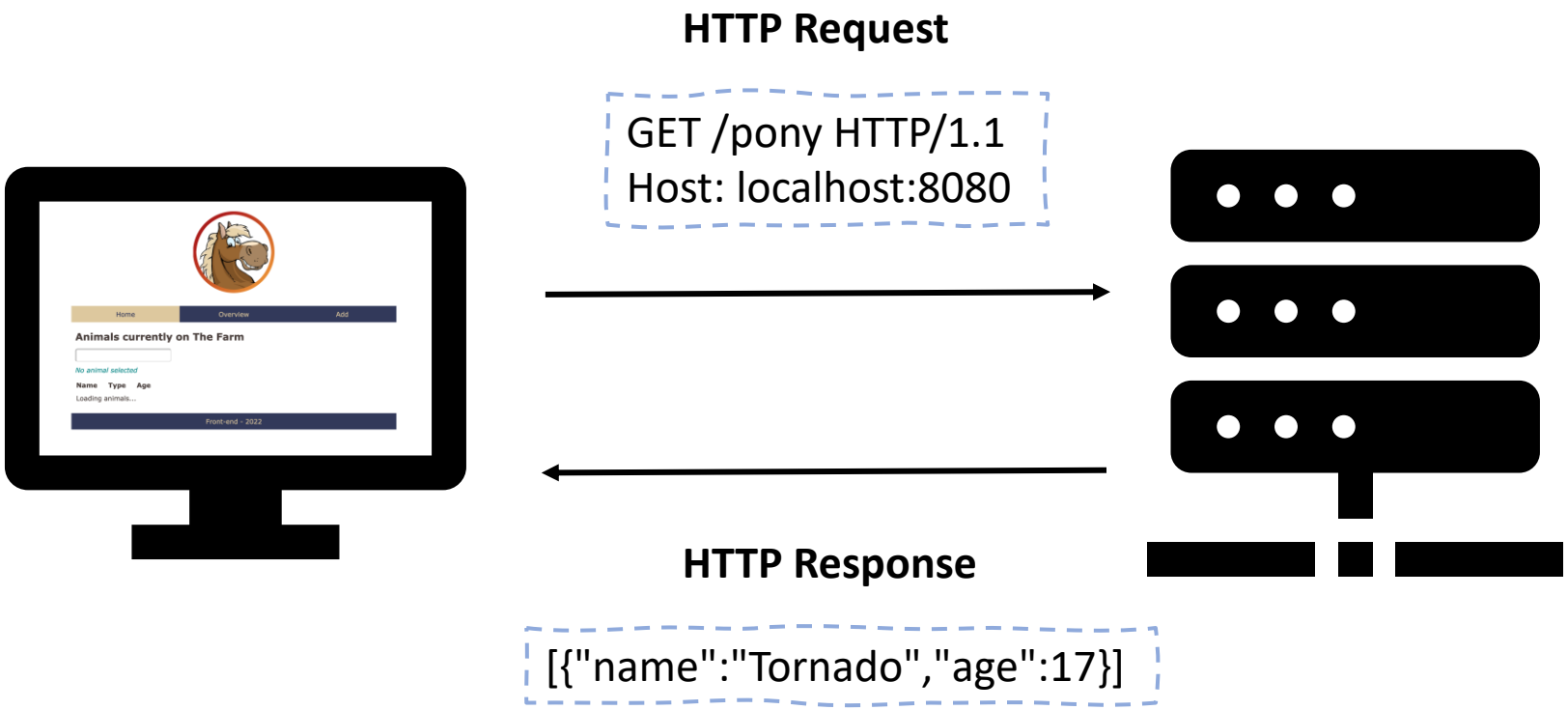
  Application **P**rogramming **I**nterface (API) of the backend

WHAT IS A BACK-END?

- Front-end and back-end need a way to communicate
  - Both need to follow the same rules to understand each other

- The **H**yper**T**ext **T**ransfer **P**rotocol (HTTP) provides a set of concepts and rules that the frontend and backend can follow for communication
  - How should the data look?
  - What operations are possible?
  - Where can the frontend find a specific operation?
  - How should the frontend invoke a specific operation?
  - How does the frontend know that the request succeeded?
  - …

WHAT IS A BACK-END?



WHAT IS A BACK-END?

- A back-end can be written in any language
  - Writing a back-end from scratch is a gargantuan job
  - Some languages provide more support than others

- The Java ecosystem has rich support for coding back-ends
  - Java is the core language
  - Java EE is a library that handles a lot of backend concerns
    - Storing data
    - Receiving requests and sending responses using HTTP
    - Security
  - Spring is a library similar to Java EE managed by VMWare

Spring Boot

SPRING BOOT

- Spring Boot is
  - A back-end framework that helps with typical back-end functionality
    - Starting and stopping a server that can respond to HTTP requests
    - Retrieving and storing data from a database
    - Securing your backend
    - ...
  - A modular framework
    - Every module can be loaded separately by specifying different dependencies
  - A high-level framework
    - It loads in many more low-level dependencies to provide its functionality



SPRING BOOT

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>3.4.1</version>
</parent>
```

- Spring Boot is a framework that takes full control of your application
- To convert your codebase to a Spring Boot application add the above code snippet in your pom file
- This makes your maven project the child of the Spring Boot template maven project, which gives you a lot of pre-configured plugins and other functionality

SPRING BOOT

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

- The Spring framework is *modular*
  - Separate parts can be included independently
- Include the above dependency to load the web-module that transforms your application to a long-running backend
  - This dependency does not need a version! It works together with the *parent* section to load multiple dependencies automatically.

SPRING BOOT

```
@SpringBootApplication
public class Application {

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

- @SpringBootApplication: Enables auto-configuration and component scanning.
- main method: Standard Java entry point.
- SpringApplication.run(): Bootstraps the application and starts the embedded server (Tomcat).

EXERCISES @CLASS

- Exercise 1: Convert Maven project into a Spring Boot Project
  - The instructions can be found in the README.md file



HTTP GET

- HTTP Building Blocks
- Implementing a HTTP GET Call with Spring

HTTP BUILDING BLOCKS

- Remember that the HTTP protocol governs the rules of communication
  - Every *resource* is given a URL (**U**niform **R**esource **L**ocator) to identify it
  - Methods can be performed on a resource
    - Every method consists of a request and a response
  - Requests & responses consist of a body and a header
    - The header contains metadata
      - Who executed this request?
      - In what format does the sender of the request expect a response from the backend?
      - …
    - The body contains the data of the request or response
  - HTTP is *stateless*
    - Each request - response cycle is self-contained
  - HTTP is not secure by default
    - Request and response are sent in plain text
    - The HTTPS (HTTP **S**ecure) extension encrypts messages

HTTP BUILDING BLOCKS — URL

- Every HTTP resource has a URL
  - The root URL of your backend is http://localhost:<port>
    - "localhost" stands for your computer, since the backend is running on your pc
    - You can see the port your backend started at in the logs of your application
  - Every resource is mapped to a subdirectory of the root URL
    - For example
      - http://localhost:8080/pony
      - http://localhost:8080/jockey/Craig/horse
  - You can perform operations on your backend by invoking methods on URLs

- The combination of method and URL must be unique within the backend

HTTP BUILDING BLOCKS — METHODS

- The method indicates what type of request is being sent to the URL
  - GET: retrieving information
  - POST: storing new information
  - PUT: updating existing information
  - DELETE: removing information

- Other, more specialised methods exist
  - https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods

- The combination of method and URL must be unique within the backend

HTTP BUILDING BLOCKS — STATUS CODES

- The status code of the response indicates whether the back-end could succesfully process the request
  - 2xx – Succes
  - 3xx – Redirect
  - 4xx – Client Error
  - 5xx – Server Error

- Other, more specialised status codes exist
  - https://en.wikipedia.org/wiki/List_of_HTTP_status_codes

HTTP BUILDING BLOCKS — REST API

- A HTTP API that follows a particular structure with its URLs, methods and status codes is sometimes referred to as a *REST API*
- In this course we will use the terms REST API and HTTP API interchangeably

HTTP BUILDING BLOCKS — REQUEST RESPONSE

- A HTTP request and response consist of a header and a body

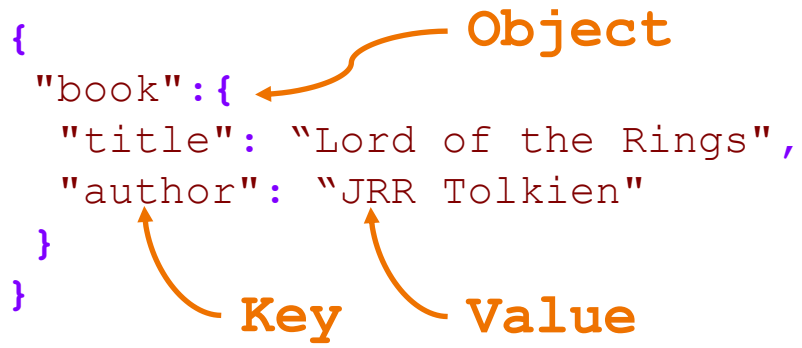
- The header contains key-value pairs with metadata
  - For example, what is the format of the body? When was the message sent? …
- The body contains the actual data of the request or response
  - We will structure the body of our HTTP messages with **J**ava**S**cript **O**bject **N**otation (JSON)
    - Describes information in a textual format, similar to XML
    - Hierarchical structure
    - Designed to have less overhead as XML
    - Closely linked to JavaScript

HTTP BUILDING BLOCKS — REQUEST RESPONSE


```
{
 "house":{
  "living-room":{
   "table":[
    {
     "name":"big table",
     "wood":"oak"
    },
    {
     "name":"coffee table",
     "wood":"fir-wood"
    }
   ]
  },
  "bedroom":{
   "name":"parent's bedroom",
   "bed":{
    "length":"",
    "width":"",
    "name":"parent's bed"
   }
  }
 }
}
```

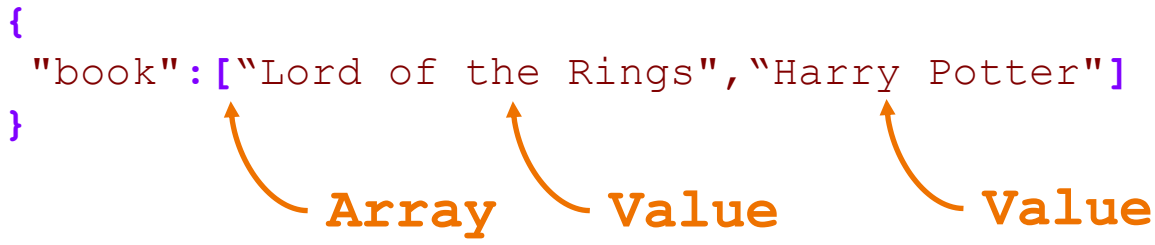
HTTP BUILDING BLOCKS — REQUEST RESPONSE

- JSON has two types
  - **Object**
    - Indicated with { }
    - Unsorted set of key/value pairs
    - Key and value are separated by :
    - Key/value pairs are separated by ,



HTTP BUILDING BLOCKS — REQUEST RESPONSE

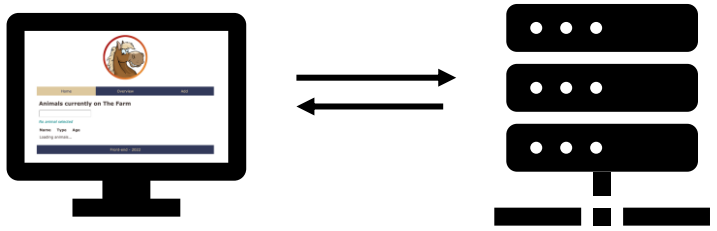
- JSON has two types
  - **Array**
    - Indicated with `[]`
    - Sorted list of values
    - Values are separated by `,`



HTTP BUILDING BLOCKS — REQUEST RESPONSE

Full HTTP Request Example

```
 Trying [::1]:8080...
* Connected to localhost (::1) port 8080
> GET /pony HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/8.2.1
> Accept: */*
```



Full HTTP Response Example

```
 HTTP/1.1 200
< Content-Type: application/json
< Transfer-Encoding: chunked
< Date: Thu, 22 Feb 2024 13:42:05 GMT
<
{ [35 bytes data]
7250[{"name":"Tornado","age":17}]
```

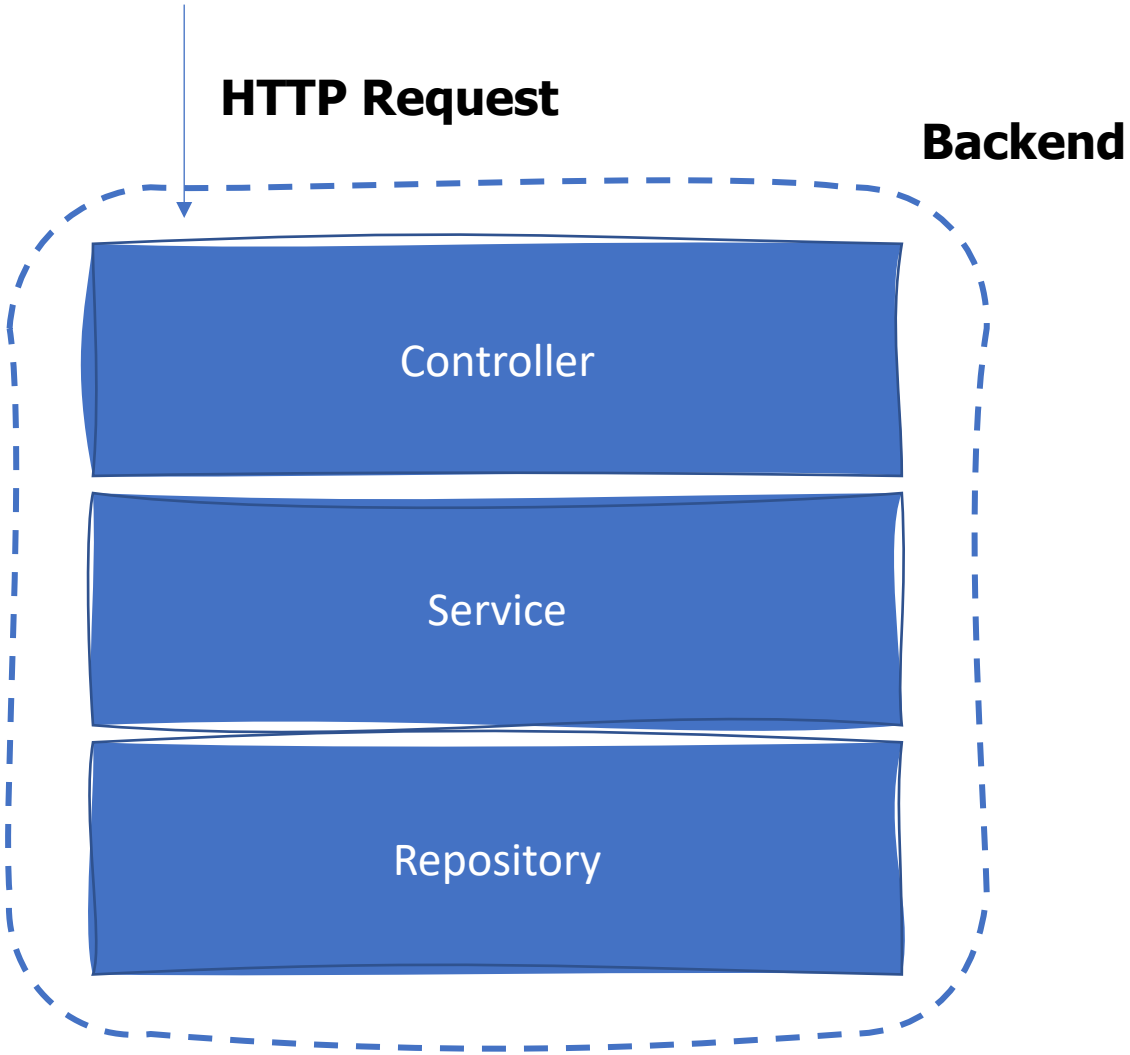
IMPLEMENTING HTTP GET

- Suppose that we want to write a back-end that can answer the following HTTP GET calls
  - "Retrieve all ponies stored in the backend"
  - "Retrieve a pony with a specific name"

- A back-end is a complex application
  - It is not feasible to write all the code of a backend in a single .java file
  - An important question is how we can structure the code so that the complexity of the backend remains manageable

- A good solution for a small backend is the *Layered Architecture* pattern
  - In a layered architecture, backend code belongs to one of three layers
  - We will build the codebase using the layered architecture pattern

IMPLEMENTING HTTP GET



Translates HTTP request to Java and translates Java back to a HTTP response

Guards application rules and gives commands to repositories to store, retrieve data

Communicates with the database and provides an API to use the database from within Java code

IMPLEMENTING HTTP GET

- To implement the HTTP GET calls we will perform the following steps

1. Write a Pony class to model a pony
2. Implement a repository to handle the storage of the ponies
    1. We will defer working with a database until later
    2. Our repository layer will hold the data in-memory as an alternative
3. Write a service to check application rules and direct the repository
    1. This will be a very simple class in this case
4. Write a controller to receive the HTTP request and to answer with a HTTP response
    1. The controller will use the service to return a response

IMPLEMENTING HTTP GET

```
public class Pony {

    private String name;
    private int age;

    public Pony(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }
}
```

- We can model a Pony with this simple class

IMPLEMENTING HTTP GET

```
@Repository
public class PonyRepository {

    private List<Pony> ponies;

    public PonyRepository() {
        ponies = new ArrayList<>();
        ponies.add(new Pony("Bella", 5));
        ponies.add(new Pony("Luna", 7));
        ponies.add(new Pony("Angel", 12));
    }

    public List<Pony> allPonies() {
        return ponies;
    }
}
```

- We will defer working with a database until later
- Our repository layer will hold the data in-memory as an alternative

IMPLEMENTING HTTP GET

```
@Service
public class PonyService {

    private PonyRepository ponyRepository;

    @Autowired
    public PonyService(PonyRepository ponyRepository) {
        this.ponyRepository = ponyRepository;
    }

    public List<Pony> allPonies() {
        return ponyRepository.allPonies();
    }
}
```

- Our service layer is very simple
- When instructed to retrieve all ponies, we get the data via the repository

IMPLEMENTING HTTP GET

```java
@Repository
public class PonyRepository {

    private List<Pony> ponies;

    public PonyRepository() {
        ponies = new ArrayList<>();
        ponies.add(new Pony("Bella", 5));
        ponies.add(new Pony("Luna", 7));
        ponies.add(new Pony("Angel", 12));
    }

    public List<Pony> allPonies() {
        return ponies;
    }
}
```

```java
@Service
public class PonyService {

    private PonyRepository ponyRepository;

    @Autowired
    public PonyService(PonyRepository ponyRepository) {
        this.ponyRepository = ponyRepository;
    }

    public List<Pony> allPonies() {
        return ponyRepository.allPonies();
    }
}
```

- Notice the use of @Repository, @Service and @Autowired
- @Repository and @Service are Spring annotations that tell Spring to manage these classes
  - They also indicate the layer of the layered architecture the class belongs to

IMPLEMENTING HTTP GET

```
@Repository
public class PonyRepository {

    private List<Pony> ponies;

    public PonyRepository() {
        ponies = new ArrayList<>();
        ponies.add(new Pony("Bella", 5));
        ponies.add(new Pony("Luna", 7));
        ponies.add(new Pony("Angel", 12));
    }

    public List<Pony> allPonies() {
        return ponies;
    }
}
```

```
@Service
public class PonyService {

    private PonyRepository ponyRepository;

    @Autowired
    public PonyService(PonyRepository ponyRepository) {
        this.ponyRepository = ponyRepository;
    }

    public List<Pony> allPonies() {
        return ponyRepository.allPonies();
    }
}
```

- @Autowired instructs Spring to search for managed classes that are necessary for this class to function
  - When the class is started, Spring injects the dependencies via the annotated constructor
  - Specifying @Autowired is optional

IMPLEMENTING HTTP GET

```java
@Repository
public class PonyRepository {

    private List<Pony> ponies;

    public PonyRepository() {
        ponies = new ArrayList<>();
        ponies.add(new Pony("Bella", 5));
        ponies.add(new Pony("Luna", 7));
        ponies.add(new Pony("Angel", 12));
    }

    public List<Pony> allPonies() {
        return ponies;
    }
}
```

```java
@Service
public class PonyService {

    private PonyRepository ponyRepository;

    @Autowired
    public PonyService(PonyRepository ponyRepository) {
        this.ponyRepository = ponyRepository;
    }

    public List<Pony> allPonies() {
        return ponyRepository.allPonies();
    }
}
```

- Spring based classes are **not instantiated manually**
  - When Spring is handed control, it searches for all annotated classes
  - Spring will then try to start all classes by injecting their dependencies via the constructor
  - You will get errors if classes are missing or misconfigured

IMPLEMENTING HTTP GET

```
@RestController
@RequestMapping("/pony")
public class PonyController {

    private PonyService ponyService;

    @Autowired
    public PonyController(PonyService ponyService) {
        this.ponyService = ponyService;
    }

    @GetMapping
    public List<Pony> allPonies() {
        return ponyService.allPonies();
    }
}
```

- @RestController and @Autowired are needed for Spring to find the class and treat it as controller

IMPLEMENTING HTTP GET

```
@RestController
@RequestMapping("/pony")
public class PonyController {

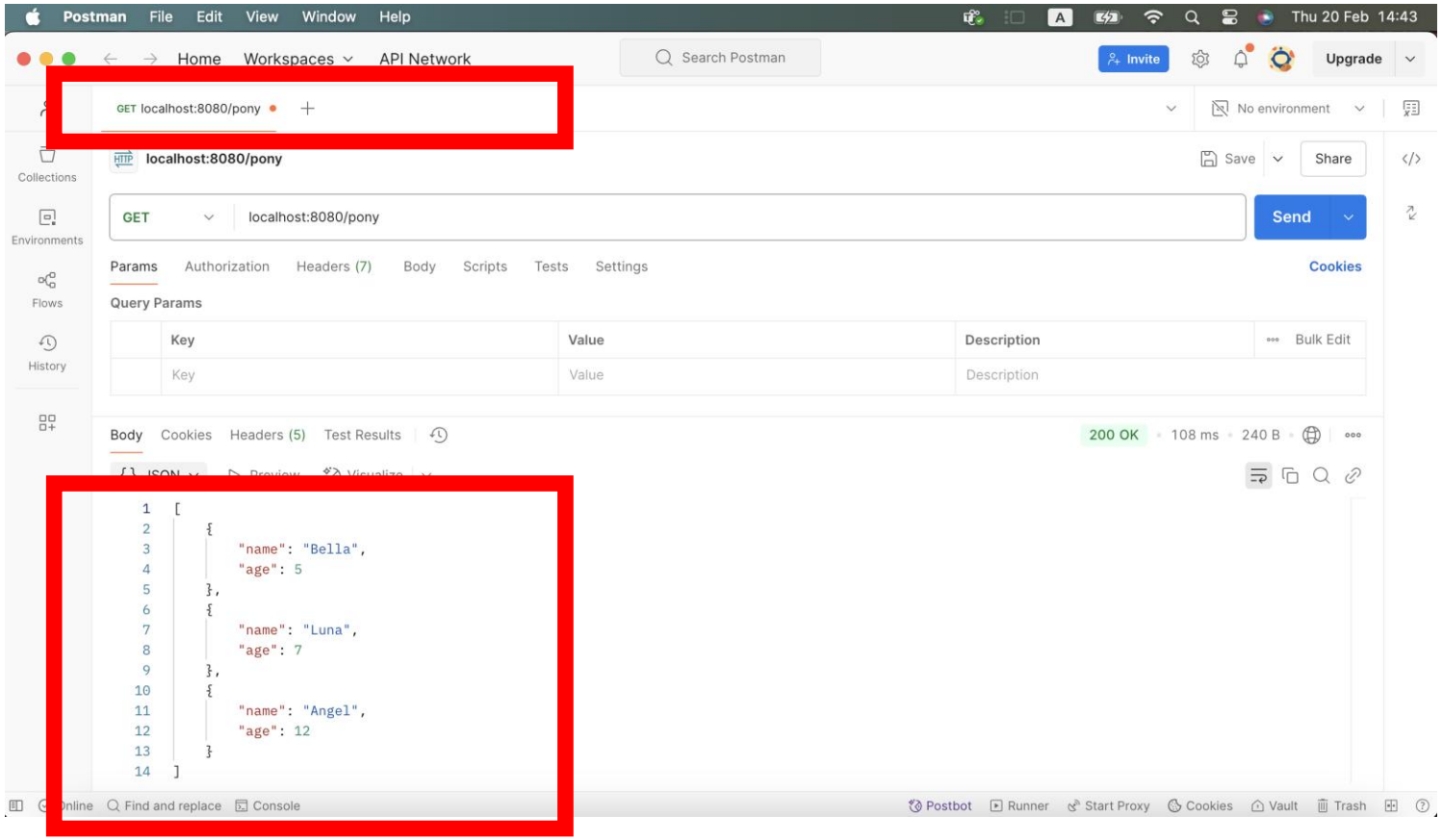
    private PonyService ponyService;

    @Autowired
    public PonyController(PonyService ponyService) {
        this.ponyService = ponyService;
    }

    @GetMapping
    public List<Pony> allPonies() {
        return ponyService.allPonies();
    }
}
```

- @RequestMapping registers all methods under the specified subdirectory of the backend's root URL
- @GetMapping indicates that this method offers a HTTP GET call

IMPLEMENTING HTTP GET



- The translation of HTTP request to java code and back to HTTP response is handled automatically by Spring

IMPLEMENTING HTTP GET

```
@RestController
@RequestMapping("/pony")
public class PonyController {

    // ...

    @GetMapping
    public List<Pony> allPonies() {
        return ponyService.allPonies();
    }

    @GetMapping("/{ponyName}")
    public Pony getPonyByName(@PathVariable(value = "ponyName") String name) {
        return ponyService.findPonyByName(name);
    }
}
```

- Additional HTTP methods can be added by adding more @GetMappings
- @GetMapping can specify a new subdirectory
  - The full URL of the method getPonyByName becomes “http://localhost:8080/pony/Bella”

IMPLEMENTING HTTP GET

```
@RestController
@RequestMapping("/pony")
public class PonyController {

    // ...

    @GetMapping
    public List<Pony> allPonies() {
        return ponyService.allPonies();
    }

    @GetMapping("/{ponyName}")
    public Pony getPonyByName(@PathVariable(value = "ponyName") String name) {
        return ponyService.findPonyByName(name);
    }
}
```

- @PathVariable maps a part of the URL to a method parameter

IMPLEMENTING HTTP GET

```
@RestController
@RequestMapping("/pony")
public class PonyController {

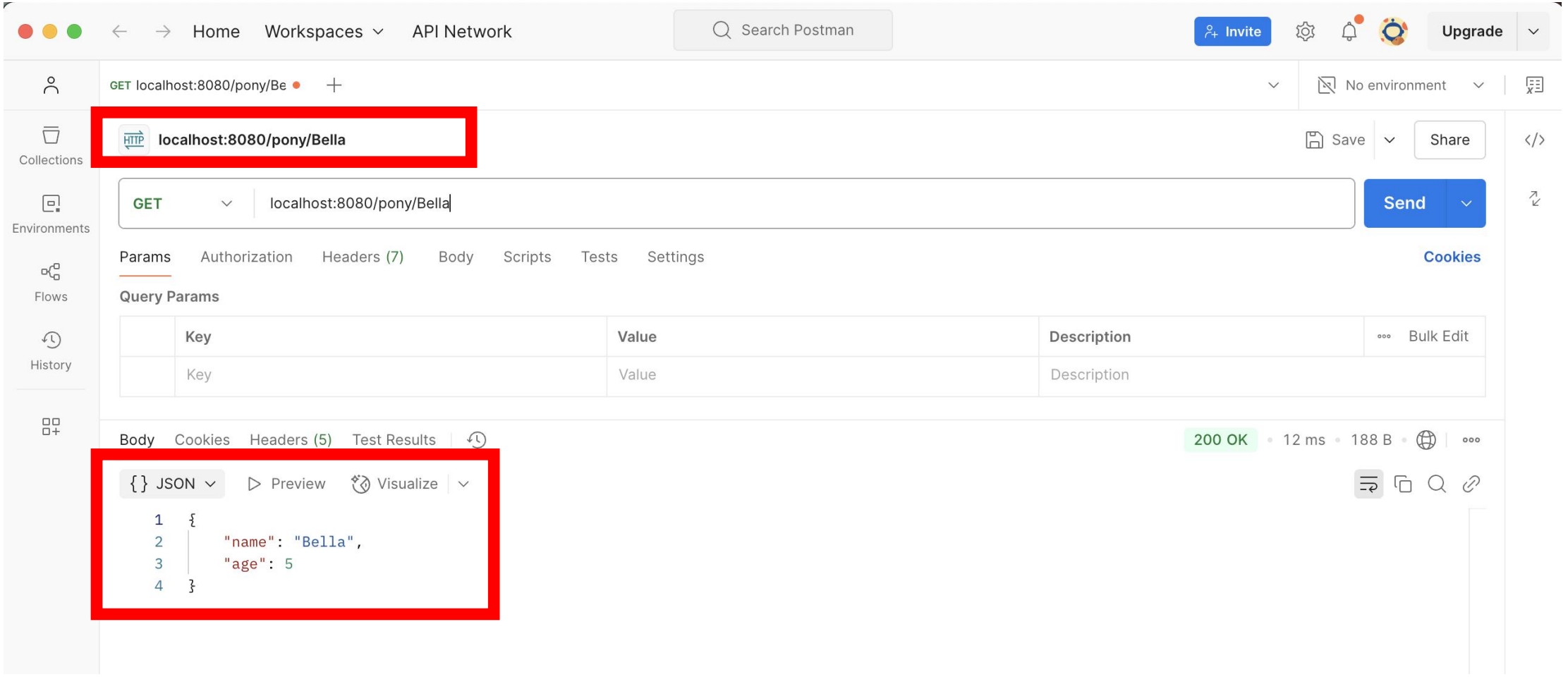
    // ...

    @GetMapping
    public List<Pony> allPonies() {
        return ponyService.allPonies();
    }

    @GetMapping("/{name}")
    public Pony getPonyByName(@PathVariable String name) {
        return ponyService.findPonyByName(name);
    }
}
```

- @PathVariable can be simplified if the method parameter has the same name as the templated part of the URL

IMPLEMENTING HTTP GET



IMPLEMENTING HTTP GET

```
@RestController
@RequestMapping("/pony")
public class PonyController {

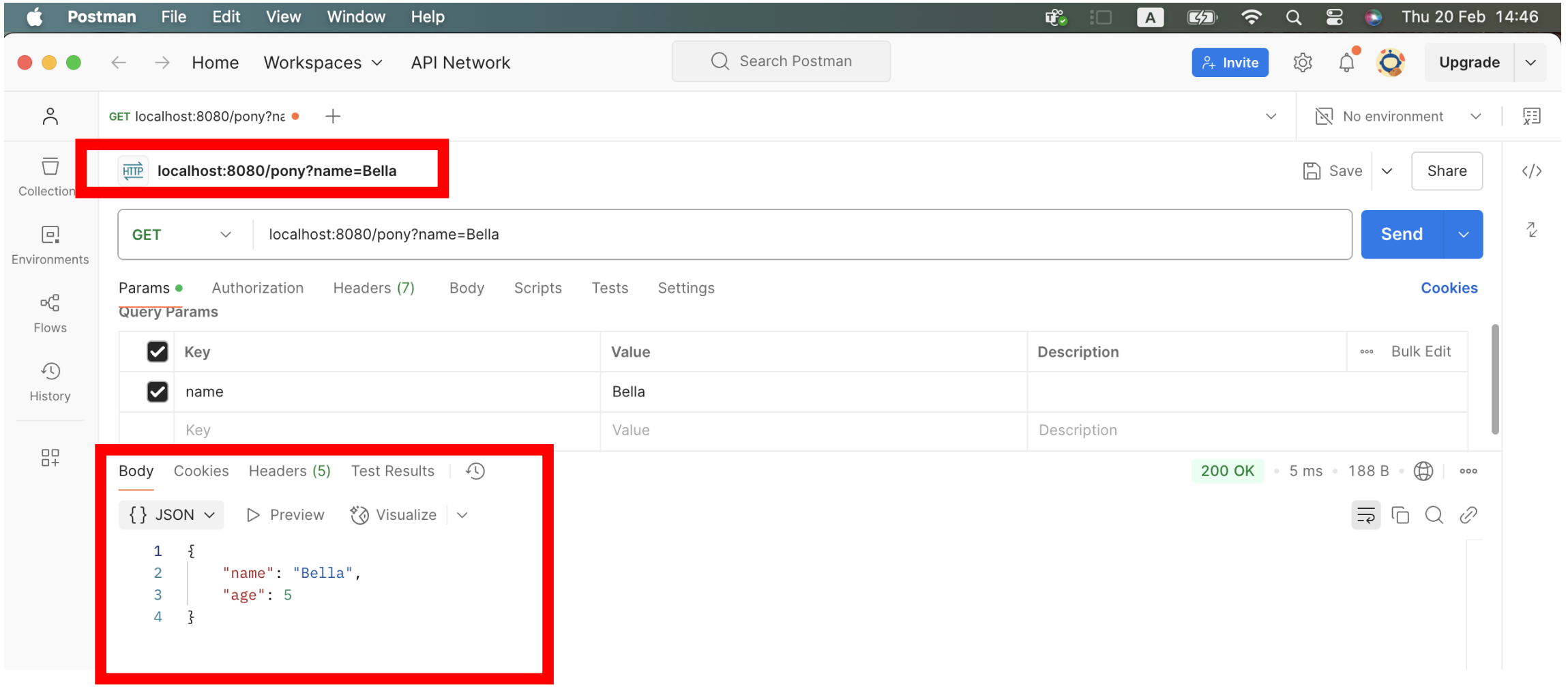
    // ...

    @GetMapping
    public Object allPonies(@RequestParam(value = "name", required = false) String name) {
        if (name == null) {
            return ponyService.allPonies();
        else {
            return ponyService.findPonyByName(name);
        }
    }

    // ...
}
```

- @RequestParam is a second way to pass information to the method via URL
  - Note that a @RequestParam is not a part of the URL, it is appended to the URL
  - Remember that every @GetMapping must have a unique URL

IMPLEMENTING HTTP GET



IMPLEMENTING HTTP GET — UNIT TEST

```
@Test
public void givenFindPonyByName_whenPonyExists_thenItIsReturned() {
    PonyRepository repository = new PonyRepository();
    PonyService service = new PonyService(repository);

    Pony foundPony = service.findPonyByName("Bella");
    Assertions.assertEquals("Bella", foundPony.getName());
}
```

- We can unit test controller, service and repository methods like before
  - Since our main method is not executed in our test, Spring will not create the classes
  - We can instantiate the class manually in our test and check happy and unhappy paths
  - If a class needs a dependency that Spring would provide, we can provide it manually

- These types of unit tests are called *social unit tests*, since they test multiple classes at the same time

EXERCISES @CLASS

- Exercise 2: Get user by email.



EXERCISES @HOME

- Exercise 3: Get users with name.
- Exercise 4: Analyze client-server interaction.



