Spring Boot Development

Version: Kela

Agenda

- Spring Boot Overview
- Hello Spring
- Beans
- Spring Web MVC
- Implementing REST services with Spring Boot
- Testing
- Database access with Spring Boot and JPA
- Spring Transactions
-
- Additional info: Spring Security, Spring Actuators

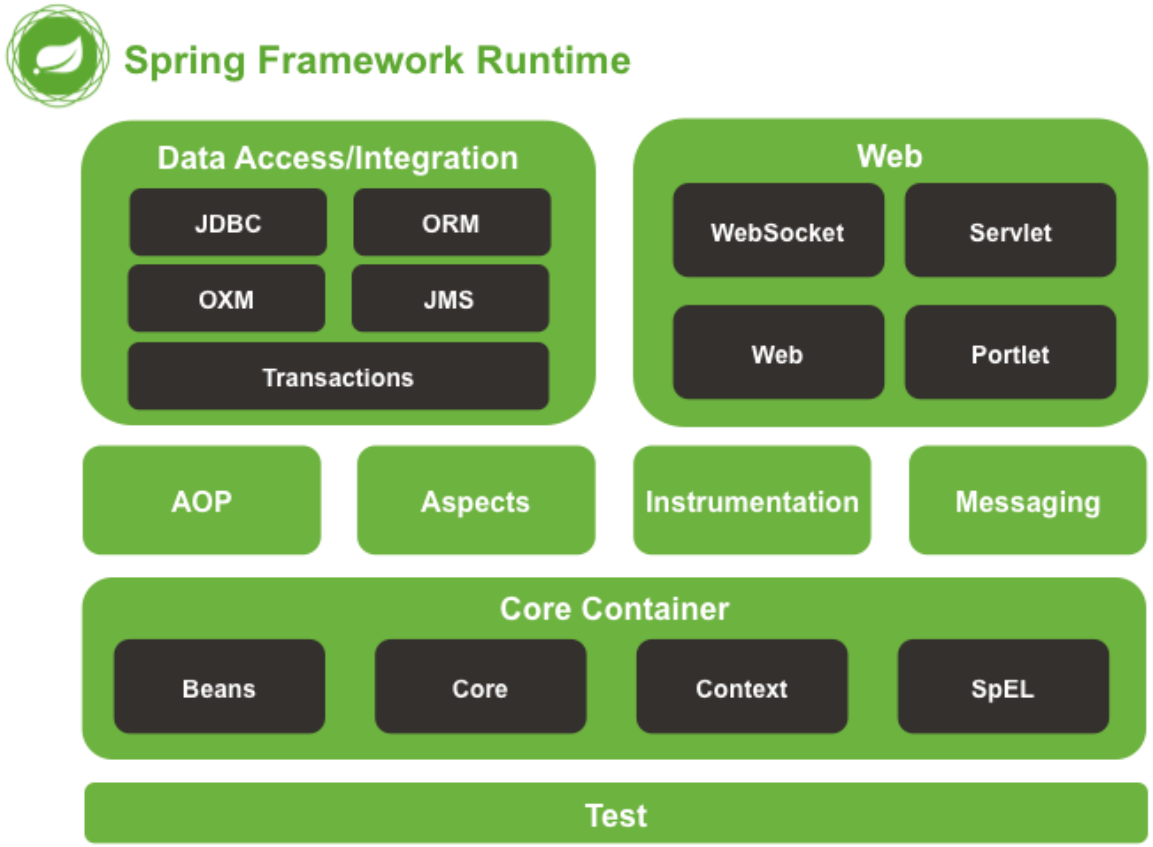
Spring Boot Overview

Spring Boot

- Spring Framework was created as an alternative solution for problems solved by Java EE
  - Since Spring has evolved, and Spring Framework is “merely” a project in the Spring family
- Spring and Java EE are not fierce competitors, but can be used together in a single project. Both have even been developed in a way to accommodate each other the best possible way
- Spring Boot is not a lightweight version of Spring, but *an opinionated way of creating Spring-based applications* – convention over configuration
- Main difference with Spring Boot and Java EE applications is that Spring Boot does not *require* a dedicated application server
  - Same applies to Spring-based applications in general
  - An application server *can* be used, and in case both Spring and Java EE are used, one *must* be used

Spring Framework

- Spring Framework (v. 4)



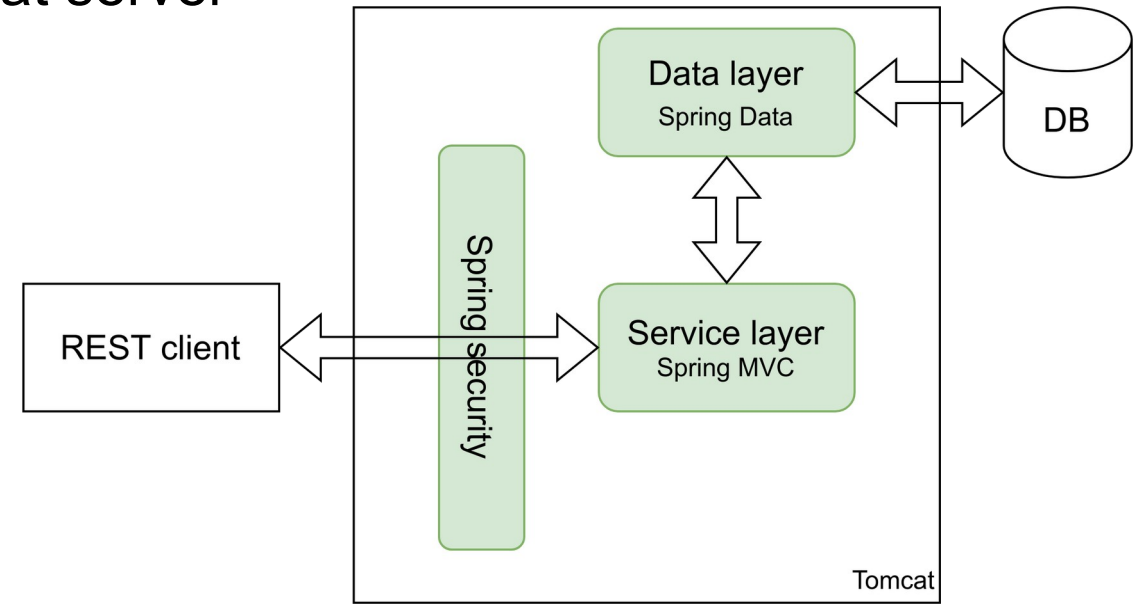
https://docs.spring.io/spring-framework/docs/4.0.x/spring-framework-reference/html/overview.html

Spring Projects

- Today Spring Framework considered by Canonical to be only one project among other Spring projects
  - More: https://spring.io/projects
- There are several projects in addition to Spring Framework, including:
  - **Spring Boot** – Convention over configuration style of creating Spring-based applications
  - **Spring Data** – consistent access to different data sources RDBMS, NoSQL, …
  - **Spring Security** – authentication and authorization support
  - Spring Cloud – e.g. building and deploying microservices in distributed environments
  - **Spring HATEOAS** – for REST services implemented using HATEOAS principles
  - **Spring Web Services** – for SOAP services

Spring Boot applications

- Since Spring Boot is a framework, there is not just one type of Spring Boot applications
- A common Spring Boot application is a microserver, accessing a database and offering a REST API for services
- The HTTP communication implemented using an embedded Tomcat server



Hello Spring Boot

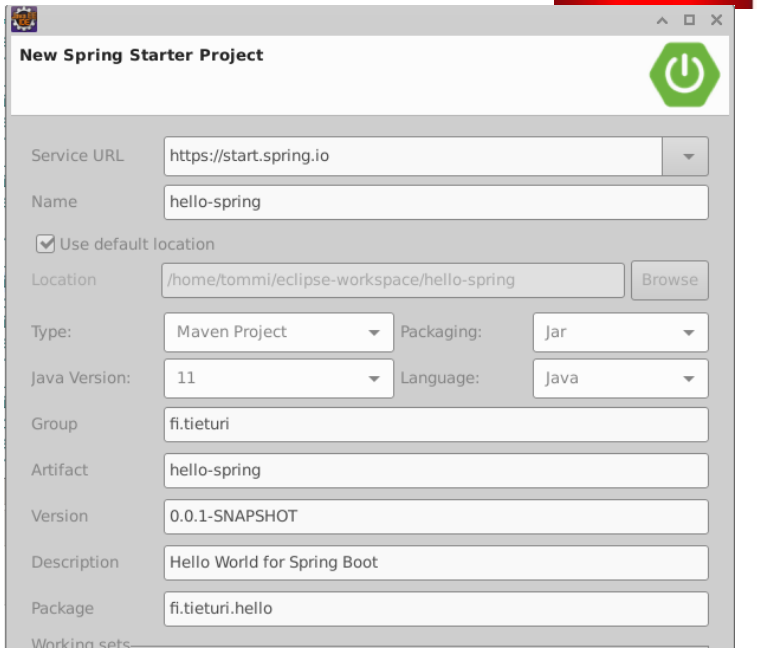
Creating a Spring Project

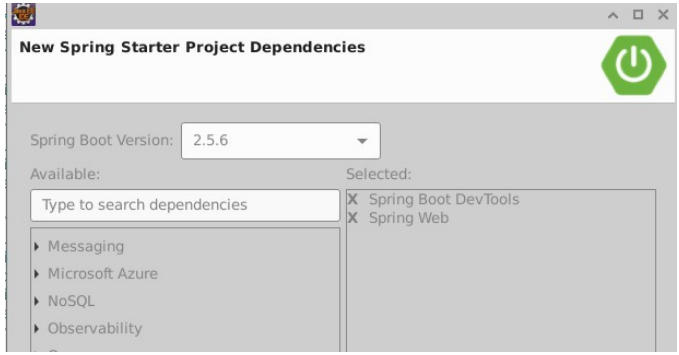
- Spring Boot projects are created using Maven or Gradle artefacts
- Spring Boot Initializer projects can be used
  - Obviously can be customized and company can use the customized parent when creating new projects
- Eclipse STS enables creating the project using a New Spring Boot project wizard
  - Spring Boot Starter project type

- Now, let's create and run a basic project template
- Lab 1: open Eclipse (or another IDE if preferred), and follow the instructor in creating the project

Creating a Spring Project – Eclipse with STS

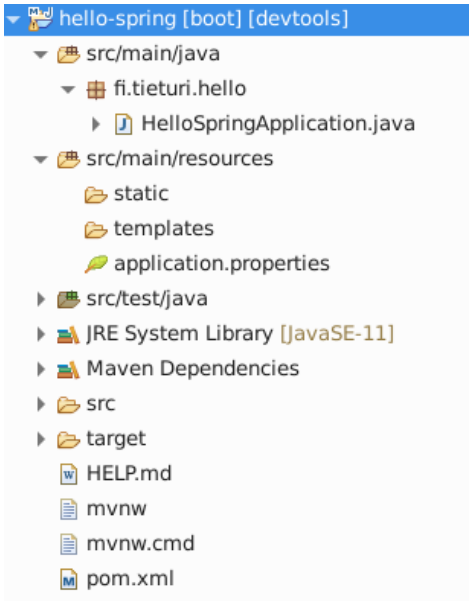
- Choose New | Other..
- Find the Spring Starter Project
- Setup the project, normal Maven settings
- Next choose dependencies, for Hello Spring we'll choose Spring Boot Dev Tools, and Spring Web
- Finish will load the project zip file from Spring tools, and unzip it to our project folder and sets up the project





Project contents

- The important files:
  - HelloSpringApplication.java main class
  - application.properties – empty for now
  - pom.xml – based on choices for SpringInitializr
  - (also test, but that's for later)



- HelloSpringApplication is annotated with @SpringBootApplication, a combination of
  1) @Configuration – can define Beans
  2) @EnableAutoConfiguration - Spring Boot automatically searches other Beans
  3) @EnableWebMvc – web support
  4) @ComponentScan – scans Beans for use

Running a Spring Project

- A Spring Boot application by default packages itself to a self executing Jar file with an embedded Tomcat server
- Since we have a self executing Jar file with a main-method, we just have to run the Jar as any Java stand alone application
  - With maven: `mvn -q spring-boot:run`
- The starter can be run immediately, we mainly see just some ASCII art and some Tomcat initialization logs
- With Spring Boot Dev Tools included we have also enabled Hotstart so saving the code while the application is running will relaunch it automatically
  - In many cases this is enough to see the differences made

Sidenote: Logging

- As with any Java project there are several choices for implementing logging with Spring Boot – internally Spring Boot uses Commons Logging
  - https://docs.spring.io/spring-boot/docs/2.1.13.RELEASE/reference/html/boot-features-logging.html
- In this material the default logging is used, so we can use

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

@SpringBootApplication
public class HelloSpringApplication {
	private static Logger LOGGER = LoggerFactory.getLogger(HelloSpringApplication.class);
	public void someMethod() {
		LOGGER.info("Logging a number: {}", 42);
	}
```

```
2021-11-16 22:50:23.792  INFO 44775 --- [  restartedMain] fi.tieturi.hello.HelloSpringApplication  : Logging a number: 42
```

Beans

Beans in Spring Boot

- Applications using Spring Framework, and hence also Spring Boot, utilize Beans and Bean lifecycle to create the overall memory model and application's internal communication
- Beans in Spring Boot are typically created using a proper stereotype annotation, and the available Beans are *injected* when and where they are needed
- A simplified explanation: A Bean is a Java object that can be instantiated using Spring Framework
  - No args constructor (or with injected arguments)
  - Setters and getters
  - Located where it can be found (@ComponentScan)

Own components

- Own classes are most often made to be Beans by using one of Springs stereotype annotations: @Component, @Service, @Resource, or @Controller
  - There is also a @Bean annotation for methods that return a Bean
  - Older Spring versions typically used XML configuration to define classes as injectable components
- Remember to have a default constructor (or one where Spring injects the arguments)

```
import org.springframework.stereotype.Component;

@Component
public class MyCounter {
	private int value;
	public int increment() {
		return ++value;
	}
	public int decrement() {
		return --value;
	}
	public int get() {
		return value;
	}
}
```

Injecting Beans

- Injecting means that we don't actively find/create a reference to an object, but let the run environment take care of object instantiation and setting
  - Use with member variables, method parameters, constructor parameters, ..
  - By default any component in the same package as the main class, or any subpackage, is automatically found
- Spring annotation to inject a bean is: **@Autowired**
  - **@Resource** (JSR-250) and **@Inject** (JSR-330) do work, but using a single injection type is preferred
- We can inject anything Spring sees as a Bean, but mostly Spring component stereotypes

Injection Sample

- Members can be injected directly, or we can use constructors
  - Often using constructors preferred, especially with chained injections
- https://stackoverflow.com/questions/7779509/setter-di-vs-constructor-di-in-spring

```
@RestController
public class HelloController {
	@Autowired private MyCounter counter;
}
```

```
@RestController
public class HelloController {
	private MyCounter counter;

	public HelloController(@Autowired MyCounter counter) {
		this.counter = counter;
	}
}
```

Bean configuration

- It often beneficial to configure the Beans, i.e. objects, to set them up for proper use
- Common way for configurations is to combine **@Configuration** class, properties files, and possibly a profile
- Config class: a class that has one or more methods returning a Bean that is created and configured in the class
- Properties file: key-value pairs that can be changed without modifying Java code
  - E.g. configuration classes can read the values using **@Value** annotation
- Profile: Can be used to e.g. use different configurations for production, development or testing

Basic Configuration example

- The Configuration class can implement code, read values from properties files, and also inject beans to be used for configuration
- The configuration below can be used ***instead*** of **@Component** annotation in MyCounter class
  - If both used, mark one as @Primary, or use a qualifier to let Spring know which one to use (class as is, or bean from config)

```
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import fi.tieturi.hello.MyCounter;

@Configuration
public class CounterConfiguration {
	@Value("${my.counter.value:7}") //optional default (:7)
	private int value;
	@Bean
	public MyCounter configMyCounter() {
		MyCounter counter = new MyCounter();
		counter.set(value);
		return counter;
	}
}
```

application.properties

```
spring.main.banner-mode=off
logging.level.root=INFO
my.counter.value=8
```

Special Configuration: ApplicationRunner

- **ApplicationRunner** is a special interface that can be used to execute code during application startup
- Any configuration class (including the main class) can return an **ApplicationRunner** type bean, and the runner's **run(ApplicationArguments)** will be executed
  - Example below gets MyCounter implicitly injected as our method argument – implicit use of @Autowired

```
@Bean
public ApplicationRunner someCustomStartupCodeToRun(MyCounter counter) {
      return args -> {
            LOGGER.info("Hello from custom startup code");
            LOGGER.info("Counter value: {}", counter.get());
            LOGGER.info("Incremented counter value: {}", counter.increment());
      };
}
```

Bean lifecycle and scopes

- Injecting beans as before produces a *singleton* scoped object, i.e. everywhere it is injected we access the same object
- The scope can be defined using **@Scope** annotation with one of the 6 supported (6 in Spring MVC):
  - **singleton** – same bean instance everywhere, default scope
  - **prototype** – each injection uses a different instance
  - **request** – web
  - **session** – web
  - **application** – web
  - **websocket** – web
- **@Scope** can be used with components, or bean configurations

Bean scoping example

- Even a single bean type can be provided with different scopes using configuration:
- Below are two new bean methods in config class, and some code in main class instantiating the beans

```
@Bean("singletonCounter") @Scope("singleton")
public MyCounter singletonCounter() {
      return new MyCounter();
}

@Bean("prototypeCounter") @Scope("prototype")
public MyCounter prototypeCounter() {
      return new MyCounter();
}
```

```
@Bean
public ApplicationRunner scopingCodeToRun(ApplicationContext context) {
      return args -> {
            MyCounter c1 = context.getBean("singletonCounter", MyCounter.class);
            System.out.println("Counter value C1: " + c1.get());                    // 0
            System.out.println("Counter value C1 incremented: " + c1.increment()); // 1
            MyCounter c2 = context.getBean("singletonCounter", MyCounter.class);
            System.out.println("Counter value C2: " + c2.get());                    // 1
            MyCounter c3 = context.getBean("prototypeCounter", MyCounter.class);
            System.out.println("Counter value C3: " + c3.get());                    // 0
      };
}
```

Implementing REST services with Spring Web

REST services

- REST = Representational State Transfer
- Request-response service based on resources
- Resource based vs. action based as SOAP
- REST is stateless, no sessions, each request is considered autonomous – idempotency principle used
- REST client accesses a resource based on a URI, and then uses a HTTP verb to determine what it wants to do with the resource
- E.g. https://mybank.org/api/account can define a resource of accounts, and then we could have an API:
  - GET https://mybank.org/api/account - return a list of accounts
  - POST https://mybank.org/api/account - create a new account
  - GET https://mybank.org/api/account/1234 - return a single account with account number 1234

HTTP methods

- CRUD functionality with HTTP methods

Method	Intended purpose
POST	**C**reate a resource
GET	**R**ead the resource, either a list or a single
PUT	**U**pdate a resource
DELETE	**D**elete a resource
PATCH	Update a part of the resource (rather rare, and if used PUT should always pass all information while PATCH can pass only the details that are modified)

REST services

- Because REST uses only a HTTP verb and a URI to access a resource, and does not define how data is transferred it can be very effective and light-weight
- Data can be transferred in JSON, XML, Protobuf, CSV, or any format imaginable
- The liberties however do come with responsibilities: we actually have to come up with the details of the API and its usages including data format and security, unlike SOAP where many of these things are determined by the protocol itself
- Another common problem is the lack of standard API description, again completely up to the development team
  - There are common solutions, e.g. OpenAPI and Swagger

REST services with Spring

- Spring Boot offers REST service support with the Spring Web (AKA Spring MVC) module
  - This course examples and labs have included it from the beginning to the projects, so no need for further dependencies
- The module is based on Controllers that implement the API. Spring will then call the methods behind the scene in the controller classes, they will then perform their duties
- By default the transferred data is assumed to be JSON, and the default JSON library is Jackson. Conversion between Java types and JSON types takes place mostly automatically
  - We will see some examples where we use ObjectMapper class to help

REST controller implementation

- The simplest controller can be implemented by using @RestController annotation for the class, and one of the request mapping annotations for the service methods
- The general annotation is @RequestMapping, that handles all HTTP verbs unless “method” is specified
- More common are specialized annotations, e.g. @GetMapping (shorthand for @RequestMapping(method=RequestMethod.GET))
  - @PostMapping, @PutMapping, @DeleteMapping
- Request mapping annotations can take arguments, most common being path – the path being added to the base URI as an identifier for this resource

REST controller example

```
//@RestController = @Controller + @ResponseBody
@RestController("/api/hello")
public class HelloController {

     private Logger LOGGER = LoggerFactory.getLogger(HelloController.class);

     @GetMapping()
     public String hello() {
          return "Getting hello";
     }

     @GetMapping("/more")
     public String helloMore() {
          return "More hello";
     }

     // Without method=RequestMethod.GET will handle also POST, PUT, ..
     @RequestMapping(path="/path")
     public String sayHello() {
          return "Hello";
     }

}
```

REST controller example - using

```
tommi:~$ curl http://localhost:8080/api/hello
Getting hello
tommi:~$ curl http://localhost:8080/api/hello/more
More hello
tommi:~$ curl http://localhost:8080/api/hello/path
Hello
tommi:~$ curl -X POST http://localhost:8080/api/hello/path
Hello
tommi:~$ curl -s -o /dev/null -w "%{http_code}" \
> -X POST http://localhost:8080/api/hello && echo
405
tommi:~$ curl -s -o /dev/null -w "%{http_code}" \
> -X POST http://localhost:8080/api/hello/path && echo
200
tommi:~$ curl -s -o /dev/null -w "%{http_code}" \
> http://localhost:8080/api/ && echo
404
tommi:~$
```

HTTP status codes

- The status code associated with a response from the API has much importance, no need to parse the payload – if one even exists
- Common status codes with REST include:

200	OK		400	Bad request
201	Created		401	Unauthorized
204	No Content		403	Forbidden
			404	Not found
500	Internal server error		405	Method not allowed

Lab – first controller

- Implement a REST controller for your project
- Minimal solution: have your service say “Hello”

- Additional tasks if there is time:
  - Inject the counter seen in the materials to your controller, and add the counter value to the response
  - Create a response class with fields: `String greeting`, and `int count` (with getters and setters). Return an object of the new response class instead of a `String`
  - Can you return the Counter as is?
  - Create e.g. dev and prod profiles, and configure your counter differently for each – verify you can run both profiles
    - Use vm arg -Dspring.profiles.active=dev when running

REST APIs – design and implement

- REST APIs can be implemented using API first or Code first paradigms
  - Design/contract/API first vs. code first / Top-down vs. bottom-up
- Without a pre-existing API the Code first approach can be tempting to developers: let's implement it and *then* document what was achieved
- If there are for example multiple teams working with multiple interconnected micro servers in a microservice platform, the API first becomes easily tempting
- The documentation can be done using OpenAPI, but the same OpenAPI can be used to generate the clients from API documentation
  - https://github.com/OpenAPITools/openapi-generator/blob/master/docs/generators.md

API documentation with Swagger

- REST itself does not define a way to document the API
- A common documentation method is to use OpenAPI, and it most common implementation Swagger
- Swagger 3 support is available through Springdoc

```
<dependency>
    <groupId>org.springdoc</groupId>
    <artifactId>springdoc-openapi-ui</artifactId>
    <version>1.5.12</version>
</dependency>
```

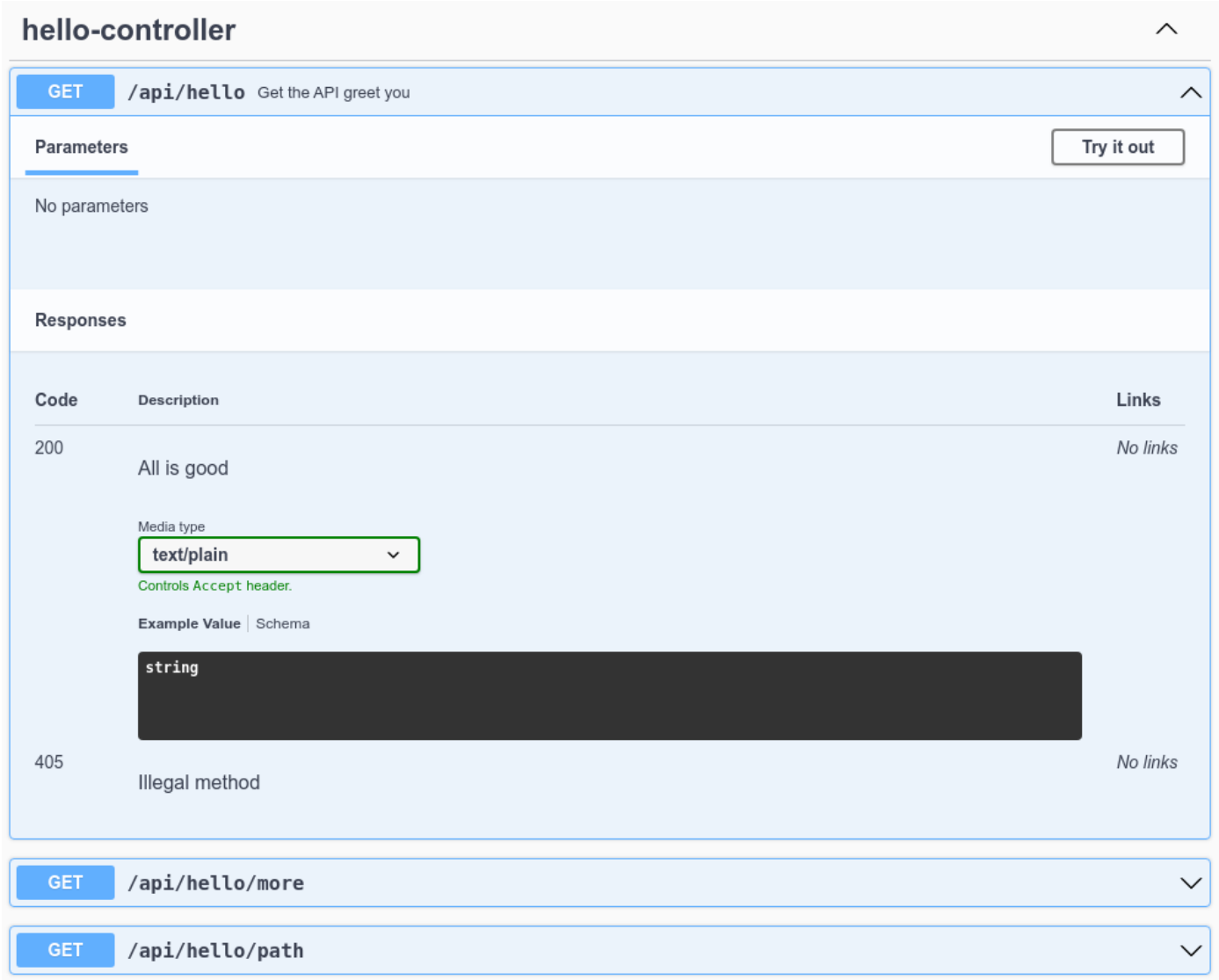
- After adding the dependency we can immediately access the api docs in JSON: http://localhost:8080/v3/api-docs/
- Even nicer is the UI: http://localhost:8080/swagger-ui.html

Improving the documentation

- The documentation is automatically generated, but we can give the generator additional pieces of information, a human understandable description of what a method does
- @Operation and @ApiResponses fill the description of an API operation e.g. for the developer wanting to use the API

```
@Operation(summary = "Get the API greet you")
@ApiResponses(value = {
          @ApiResponse(responseCode = "200", description = "All is good",
            content = { @Content(mediaType = "text/plain",
            schema = @Schema(implementation = String.class)) }),
          @ApiResponse(responseCode = "405", description = "Illegal method",
            content = @Content)})
@GetMapping()
public String hello() {
     return "Getting hello\n";
}
```

Swagger UI from example



Request data

- Resource URIs are often based on dynamic information, (e.g. bank account number), or we want to pass data from the client to the service (e.g. user information for creating a new account)
- Path parameters are used when the parameter is part of the resource URI /api/v1/accounts/*1234*
- Request parameters can provide additional info for response, e.g. sort order or paging info /api/v1/accounts?page=1&size=20
- Request body is used with POST and PUT to pass structural data, for example user information. This is not part of the resource URI, and the format is free to choose – most commonly JSON

Path parameters

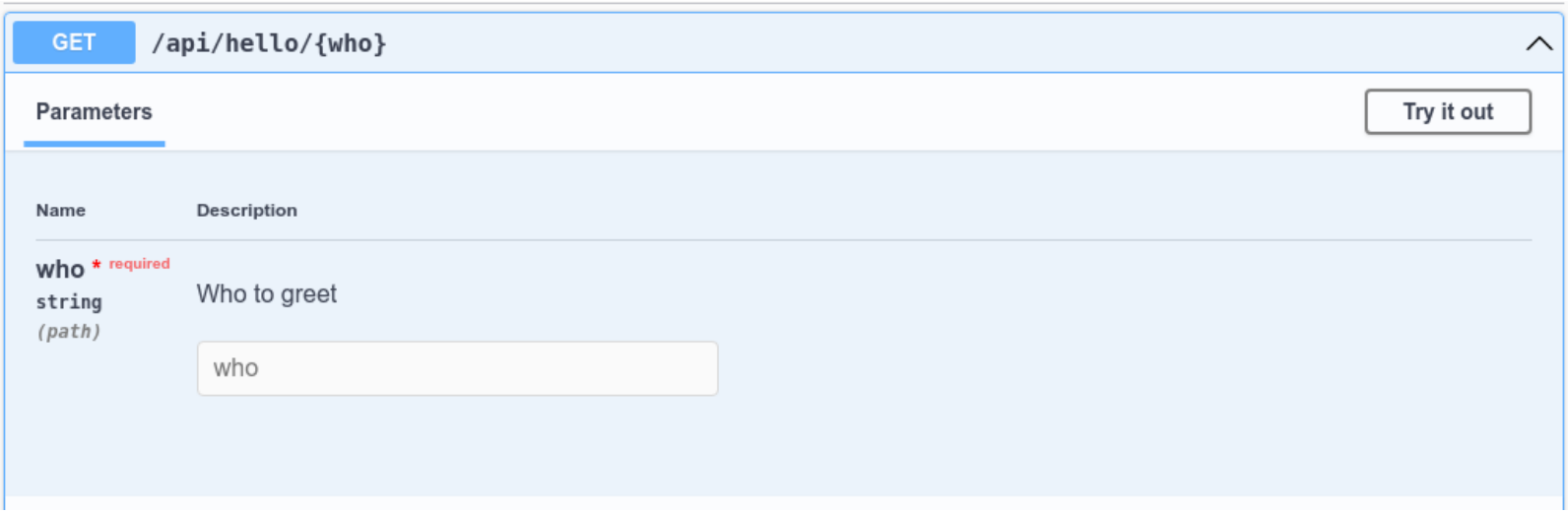
- Path parameters are used with dynamic resource URIs
- If we are trying to access a bank account, there is no point in hard coding every single account in our code
  - In addition new accounts would require rebuilding and deploying the applications
- Path variables are defined by

  1) adding a path part to our mapping annotation:
  `@GetMapping(“/{nr}”)`

  2) Annotating the request method parameter with `@PathVariable(name="nr")` annotation
- Swagger documentation: @Parameter

Path parameters - example

```
@GetMapping(value = "/{who}")
public String helloWithName(@Parameter(description = "Who to greet")
                            @PathVariable(name="who") String name) {
    return String.format("Hello \"%s\"", name);
}
```



Request parameters

- Request parameters (AKA GET parameters) are used to pass information not directly associated with the resource location, mostly used with GET requests to affect the result
- We can e.g. filter, sort, or limit number of results using request parameters
- Swagger uses the same @Parameter as path parameters for providing documentation details

```
@GetMapping("/")
public List<Product> products(@RequestParam(name="prefix", required=false) String prefix) {
  if (prefix == null) {
    return productRepository.findAll();
  }
  return productRepository.findAllNameBeginsWith(prefix);
}
```

Request body

- Request body is used with PUT and POST requests
- In most cases all that is needed is to add a variable to the request method, and annotate it with @RequestBody annotation
- Spring Boot parent adds a dependency to Jackson, and Jackson will automatically handle conversions between Java and JSON
  - The same applies to return values
- There are cases where automatic conversion does not suffice, first option is to use Jackson annotations (e.g. @JsonIgnore), but we can write code to handle all the mapping details

Lab – controller with CRUD support (1/3)

- Implement a simple service to manage colors with CRUD functionality
  - Don’t worry if you don’t have time to implement everything
- Create the data model
  - Before database support use a component that stores values in a Map, create the class (@Repository annotation) - ColorRepository
  - Create a Color class for storing the individual colors
- Implement a controller with CRUD support, you can concentrate on the Happy day scenario for now, we’ll add error handling later - ColorController
- Remember to add Swagger support for easy usage of the API – naturally curl is also available

Lab – controller with CRUD support (2/3)

- Controller:
  - GET / – return all colors
  - GET /name – return a single color identified with name
  - POST / - create a new color
  - DELETE /name – delete the color identified with name
  - PUT /name – modify a color identified with name
- Color: String name, String hex, boolean custom
  - e.g. {“name”: “blue”, “hex”: “#0000ff”, “custom”: false}
- ColorRepository:
  - ColorRepository() - add a few colors initially (could also use a configuration)
  - Map<String, Color> colors; // store colors, name as key
  - List<Color> find() - return all colors
  - Color findByName(String) – return a single color
  - Color save(Color) – store a color, and return the stored (also for modifying existing)
  - boolean delete(String name) – delete the color with key name

Lab – controller with CRUD support (3/3) (not important hints)

- Since this is a lab, and we will later be using an actual database to store our data we can cheat a little: You can initialize the ColorRepository with some colors for example
  1) Create an ApplicationRunner in the main class, inject the ColorRepository as the method argument, and add colors, or
  2) Use the ColorRepository constructor, and add colors

- To generate nicer Swagger-UI documentation for returning a list of objects, use arrays for the content definition

```
@ApiResponses(value = {
        @ApiResponse(responseCode = "200", description = "List of colors",
          content = { @Content(mediaType = "application/json",
              array = @ArraySchema(schema = @Schema(implementation = Color.class)))
         })
})
```

Responses

- So far we haven't payed much attention to responses, other than the Jackson provided conversions
- Error handling has been to minimum
- An easy way to get more control over responses is to return a ResponseEntity type from the request methods
- Either return an object wrapped in the ResponseEntity, or use builders to return e.g. en error message

```
@GetMapping("/{id}")
public ResponseEntity<?> product(@Parameter(description = "Product id")
@PathVariable(name="id") String id) {
     Color found = productRepository.findById(id);
     if (found == null) {
          return ResponseEntity.notFound().build();
     }
     return ResponseEntity.ok(found);
}
```

Responses

- 200 OK is fine for most purposes, but there are other success codes that are typically used
  - ResponseEntity.status(419) or ResponseEntity.noContent()
- DELETE often returns 204 No content, not that then you do not add a body to the response
- POST often returns 201 Created, but with a body (e.g. to include a generated id), POST also should have a "Location" header that has the value for the URI that can be used to find the newly created resource

```
@PostMapping("")
public ResponseEntity<Item> createItem(@Valid @RequestBody Item item) {
     Item created = itemRepository.save(item);
     URI location = ServletUriComponentsBuilder
            .fromCurrentRequest().path("/{id}")
            .buildAndExpand(created.getId()).toUri();
     return ResponseEntity.created(location).body(created);
}
```

Lab – add error handling

- Modify the responses to be wrapped in a ResponseEntity
  - Also use 201 with POST, 204 with DELETE
- Modify the previous lab to add error handling, use proper HTTP results, e.g. 404 (Not found) for getting a single color that is not available
  - Having the method return type to be ResponseEntity<Object> or ResponseEntity<?> allows returning different types of bodies from actual results and error results
  - Can use other errors if time to implement, e.g. not possible to create a color without a name, hex value is invalid, …

Centralized Error handling

- ResponseEntities can be used, but controller advices allows centralizing error handling using exceptions
  - Older exception based approaches were HandlerExceptionResolver and @ExceptionHandler
- Exception handling enables handling error situations everywhere with common solution
- Implement a class with @ControllerAdvice annotation, and one or more methods declaring which exceptions they handle
- The methods can return a ResponseEntity, just like normal controller request methods

ControllerAdvice example

- The controller advice class

```
@ControllerAdvice
public class ControllerAdviceResponseHandler {
      @ExceptionHandler(value = { IllegalArgumentException.class })
      public ResponseEntity<String> handle(RuntimeException ex) {
            return ResponseEntity
                        .status(HttpStatus.BAD_REQUEST)
                        .body("Your bad");
      }
}
```

- Invoking the controller advice

```
@GetMapping(value = {"/", "/{name}"})
public String hello(@Parameter(description = "The name", allowEmptyValue = true)
                    @PathVariable(name="name", required = false) String name) {
      if ("name".equals(name)) throw new IllegalArgumentException("Name is name");
      return String.format("Hello \"%s\"", name != null ? name : "anonymous");
}
```

ControllerAdvice example

- An example extending ResponseEntityExceptionHandler
- The base class adds some helper methods, but may make it more difficult when trying to provide nice Swagger UI documentation

```
@ControllerAdvice
public class HelloResponseEntityExceptionHandler
  extends ResponseEntityExceptionHandler {

    @ExceptionHandler(value = { IllegalStateException.class })
    protected ResponseEntity<?> handleConflict(
      RuntimeException ex, WebRequest request) {
        String bodyOfResponse = "Having trouble accepting";
        return handleExceptionInternal(ex, bodyOfResponse,
          new HttpHeaders(), HttpStatus.NOT_ACCEPTABLE, request);
    }
}
```

Validation

- Framework handled validation is available from e.g. Hibernate JPA validation, and Java Bean validation
- Some validation takes place automatically, but for proper support application should have a dependency to the spring-starter-validation artifact
  - Spring initializr: Validation is found under the IO category
  - Manually:

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

Lab – ControllerAdvices

- Add a controller advice to handle errors in the application
- Modify the ColorController to throw a NoSuchElementException if the repository does not contain a color – start with the method returning a single color based on the color's name
  - Could use a custom Exception class, but a quick way for the lab
- See what happens when executing, should be a 500 error
- Add a ControllerAdvice to handle the NoSuchElementException type
- Modify other applicable controller methods to throw
- Note what happens to Controllers Swagger UI documentation: the controller advice methods are included in the possible responses

Accessing REST services

- With microservice architecture we often need to access other services
  - Or write a service library for others to access our service
- The Spring method to do this is to use WebClient from Spring Webflux (Webflux => Spring Reactive Web, brings spring-boot-starter-webflux dependency)
  - Old RestTemplate is no longer recommended
- With WebClient we can build the request, and then process the results
  - Multiple ways to create a WebClient object, simplest:

```
private final WebClient webClient;

public SomeService() {
     this.webClient = WebClient.create();
}
```

WebClient example

- Getting a list of products from a product service can be done for example:
  1) Create the web client (webClient)
  2) Define the method to use (get())
  3) Define the endpoint URI to call (uri())
  4) Use Mono<String> class to get the results (bodyToMono)
  5) Wait for the response (block())
  6) Convert to Java (ObjectMapper::readValue)
     - Use TypeReference when List and generics

```
String jsonResult = webClient.get()
        .uri("https://api.service.com/api/products")
        .retrieve()
        .bodyToMono(String.class)
        .block();
List<Product> words = mapper.readValue(jsonResult,
               new TypeReference<List<Product>>() {});
```

Lab – create a service to call an API (1/2)

- Continue with the Color application
- Add a new @Service class to the project, DatamuseService
  - Datamuse offers e.g. an endpoint to get synonyms for words, more info: https://www.datamuse.com/api/
- Implement a method List<String> getSynonyms(String word) that will call the Datamuse API with the word to get a list of synonyms
  - Endpoint: https://api.datamuse.com/words?ml=word
- Use a helper class for results
- Convert to a list of word-strings

```
class MuseWord {
     private String word;
     private Integer score;
     private List<String> tags;
```

Lab – create a service to call an API (2/2)

- Easy way to test is to create an ApplicationRunner bean in your main class, inject the service, and then call the method
- Once the service is running, implement a new endpoint to the ColorController: /api/colors/{name}/synonyms
  - i.e. get the synonyms for the name of the color
-

Spring Testing

Spring Test

- Spring boot parent adds a rather comprehensive combination of test support to the project automatically
  - Junit 5, Mockito, among others
    - Note: junit-vintage-engine is not included from Spring Boot 2.4 onwards
  - We can naturally add other dependencies if preferred
- Spring Test itself offers at minimum decent support for unit tests and integration tests
-

Test implementation

- As always, there are several ways to set tests up, one is to use @SpringBootTest for the test class
- Integration tests can be performed in multiple ways, including MockMvc, or directly accessing a resource using WebView
- Configurations can be created specifically for tests, we can for example return Mocked objects for some services that are injected in code tested

Test example

```
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
public class HelloControllerTest {
     @Autowired
     private WebApplicationContext webApplicationContext;

     private MockMvc mockMvc;

     @BeforeEach
     public void setup() throws Exception {
          this.mockMvc = MockMvcBuilders.webAppContextSetup(this.webApplicationContext).build();
     }

     @Test
     public void helloIsOk() throws Exception {
        mockMvc.perform(get("/api/hello")
                .accept(MediaType.TEXT_PLAIN))
                .andExpect(status().isOk());
     }

     @Test
     public void helloSaysHello() throws Exception {
        MvcResult res = mockMvc.perform(get("/api/hello")
                .accept(MediaType.TEXT_PLAIN))
                .andExpect(status().isOk())
                .andReturn();
        assertThat(res.getResponse().getContentAsString())
               .contains("hello");
     }
}
```

Inject mocks in tests

- Classes tested can inject other components, those can inject other components etc.
- Often we want to mock the injected dependencies, or create alternative implementations

1) Add a component to your test class with @MockBean annotation
2) Use Mockito's @Mock annotation, and/or @InjectMocks to inject them to the class tested
3) Create a configuration class in your test sources that will return a customized object
   - The configuration class can return a Mock object, or
   - We can for example have a MyComponent class injected normally, but for tests create a TestMyComponent class extending MyComponent – then return that from the configuration class Bean method returning MyComponent type bean

Test configuration example

- A small example for a configuration class in the test sources
- Shows returning two beans, one a mock bean, and another one returning a child class of an actual service
  - Mock beans methods can be mocked in the test class, the child class can override any base class methods

```
@Profile("test")
@Configuration
public class TestConfiguration {
    @Bean @Primary
    public SomeService someService() {
        return Mockito.mock(SomeService.class);
    }

    @Bean @Primary
    public SomeOtherService someOtherService() {
     return new TestSomeOtherService();
    }
}
```

Lab – test the colors

- Write test code for the ColorController
- Test your endpoints, and as normal in good and bad
- Use mocking when testing the DataMuse part
  - Remember to add @Profile("!test") to the DataMuseService class, otherwise there will be a conflict when Spring tries to figure out which location is where the instantiation takes place

Spring Data - JPA

Spring Data

- Spring Data provides access to relational databases, NoSQL databases and other data providers
- Adding support only needs proper dependencies, e.g. spring-data for JDBC, or spring-data-jpa for JPA
  - Default JPA implementation is Hibernate
- The database driver still needs to be a dependency

Data source

- Spring Boot takes care of setting up the Persistence Unit, and with simple cases all that needs to be done is to provide connection details in a .properties file
- If we use multiple data sources we can use the normal Spring way of using a Configuration class, and implement the necessary bean providing methods

```
spring.datasource.url: jdbc:postgresql://localhost:5432/hellodb
spring.datasource.username: postgres
spring.datasource.password: adminadmin
spring.datasource.driver-class-name: org.postgresql.Driver
spring.jpa.hibernate.ddl-auto=create-drop
spring.jpa.show-sql=true

spring.jpa.properties.hibernate.temp.use_jdbc_metadata_defaults = false
spring.jpa.database-platform=org.hibernate.dialect.PostgreSQL9Dialect
```

Entity objects

- Since JPA in Spring is JPA, we need to remind us with a few basic rules for Entity objects

1) Annotate with @Entity
2) No args constructor
3) @Id field is mandatory, @GeneratedValue if needed
4) Map with mapping annotations (@OneToOne, @OneToMany, @ManyToMany)

-

Schema for Databases

- Setting up database schema (for initial development phase, or temporary/short lived memory DBs)
  - Normal JPA schema generation from Entity classes is available
    - Configure with property spring.jpa.hibernate.ddl-auto
    - Set to **none** when schema should not be changed
  - The schema can also be defined using resources/schema.sql file
    - resources/data.sql for DB data initialization
  - Flyway and other similar mechanisms probably work better during a longer project lifecycle
- JPA does not provide a generate-classes-from-database type of feature, some tools do exist

Repositories

- Instead of directly accessing the entity manager, Spring-based applications normally use a generated repository layer to access the data
- All that needs to be done is to create an interface that extends one of Springs repository interfaces - and if needed: implement/declare additional access methods

```
public interface MyEntityRepository extends CrudRepository<MyEntity, Long>{}
```

```
public class MyEntityController {
     private MyEntityRepository entityRepository;
     public MyEntityController(MyEntityRepository entityRepository) {
          this.entityRepository = entityRepository;
     }

     @GetMapping("")
     public ResponseEntity<Iterable<MyEntity>> entities() {
          return ResponseEntity.ok(entityRepository.findAll());
     }
```

Repository methods

- Extended repository interfaces contain some methods, but additional can be generated
  - No code, just method declaration using a predefined naming scheme
- Existing methods can also be changed, e.g. findAll() can be made to return List<T> instead of Iterable<T>

Repositories – custom methods

- Declaring interface methods with predefined naming scheme will generate the necessary JPQL queries automatically
  - https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#jpa.query-methods.query-creation
- Search methods:
  - find multiple: findAll, find one: find or findOne
  - Constraints include
    - By*Field*Containing, By*Field*GreaterThan
  - Examples for entity Country with fields including area and name

```
List<Country> findAllByAreaGreaterThan(Double area);
List<Country> findAllByNameContainingIgnoreCase(String name);
List<Country> findAllByNameContainingIgnoreCaseAndAreaGreaterThan(String name, Double area);
```

Repositories – custom methods

- Custom JPQL queries can also be used if the generated don't suffice

```
@Query("SELECT u FROM User u WHERE u.name NOT LIKE '% %'")
List<User> findUsersWithOnePartName();
```

- Named queries can be used, just name the repository method as the named query (with the required arguments)

PagingAndSortingRepository

- PagingAndSortingRepository allows returning a Page object that has the possible results embedded. In addition is has the current page number, page size, page count etc.
- The repository methods take a Pageable parameter that is used to tell the Database which page of results is wanted
  - Page findAll(Pageable pageable)
- The REST endpoint can create the Pageable object manually, but using a Pageable type parameter as a request parameter
  - size and page are then automatically used as parameter names. Default values are page=0, size=20

Validation

- We can use Bean validation (and Hibernate validation) to have automatic checks for our entity objects' fields
- Common validations in javax.validation.constraints include:
  - @NotNull
  - @NotBlank
  - @Min
  - @Size
- Remember to ***add Spring validation starter as a dependency*** otherwise the validation may not take place
- In request methods mark a request body parameter with @Valid annotation
- If validation fails a MethodArgumentNotValidException is thrown
  - Controller advice can handle this for proper responses

Testing with databases

- In testing phase Spring applications typically use the default H2 memory based database
- The default settings are similar to:

```
spring.datasource.url=jdbc:h2:mem:testdb;DB_CLOSE_ON_EXIT=FALSE
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
```

- If you choose to use the default settings or not, you need to create the file /test/resources/application.properties.
  - This will result in ignoring the whole application.properties file in /main/resources/

Testing with databases

- Test classes testing a Controller can inject the repository, and directly access the database through the repository to set the data up for the tests
-

Testing the databases

- We can naturally test our repository implementation
- A test class annotated with @DataJpaTest will set up Entity classes
  - Scanning excludes @Component and @ConfigurationProperties beans – may still need to mock injected beans to those beans that are found in the scan (@MockBean, or @Mock and @InjectMocks)
- Test version of Entity manager TestEntityManager can also be useful to inject to our test class
- Each test case runs by default in its own transaction, rollback at the end
  - Disable for class: @Transactional(propagation = Propagation.NOT_SUPPORTED)

Lab – Product database (1/3)

- Start a new Product based application, create with Spring Starter project
- Add dependencies:
  - Developer Tools: Spring Boot DevTools
  - I/O: Validation
  - SQL: Spring Data JPA, H2 Database, PostgreSQL Driver
  - Web: Spring Web
- Modify the H2 scope to be test in the project pom.xml
- Add Database properties to your resources/application.properties file
  - The example in the materials should do nicely

Lab – Product database (2/3)

- Add a Product entity class with fields Long id, String name, String description, and Double price
- Add a ProductRepository interface extending CrudRepository<Product, Long>
- Create a ProductController, implement initially a request method to get all products in our database
- Database initial data:
  - If you use tests to verify: add Products in your test class @BeforeEach method
  - If you use Swagger UI to play with the API: initialize the database with a few products in an ApplicationRunner bean
    - spring.jpa.hibernate.ddl-auto property should be create-drop, so the db is recreated between all runs – good for early development phase
  - If you use both: add @Profile("!test") or @Profile("dev") to the application runner, and set your run configurations accordingly

Lab – Product database (3/3)

- When the list of products comes from the controller, add more implementation. The following list is a suggestion of things to do, do as many as you have to to implement, and you can also try out something not in the list:

1) CRUD support
2) Add a ControllerAdvice to handle exceptions
3) Allow filtering products based on price
4) Validate the products, e.g. NotBlank name, non negative price – remember the ControllerAdvice
5) Add the DatamuseService to the product names

JPA relationships

- Relationships between Entity classes are defined in JPA, so Spring operates with the the same way as “normal” JPA code
- In OneToMany and ManyToMany relationships JPA defaults to LAZY fetching, but in REST endpoints we often would like eager results
- Instead of making the fetch eager by default, we can add an eager version of the fetch to the repository by using a JPQL query

```
public interface CompanyRepository extends CrudRepository<Company, Long> {
     @Query("SELECT c FROM Company c JOIN FETCH c.addresses addresses")
     List<Company> findAllCompaniesEager();
}
```

Lab JPA relationships (1/2)

- We now want to add stores to our product application, and also know how many and which products we have in each of our stores
- Add two more Entity classes to the application: Store, and Inventory
- Store is a store that sells products, for the lab you can add a name in addition to the generated id, and a OneToMany mapping to a list of inventories
- Inventory entity has a OneToOne mapping to a Product, and ManyToOne mapping to a Store, and a numeric field for quantity (how many particular products does a particular store have)

Lab JPA relationships (2/2)

- Implement StoreRepository, extend the interface you want (Crud, Jpa, PagingAndSorting)
  - Can also implement the JpaTests before the Controller, or first the controller
- Implement a StoreController, for this lab don't bother with all the nice endpoints we could have. Implement one: GET /api/v1/store/{id}/products that lists all the products the store with the id has with quantities
- Initialize the database with some values, and see you got the endpoint working (test or Swagger UI or both, up to you)
-

Spring Transactions

Transactions

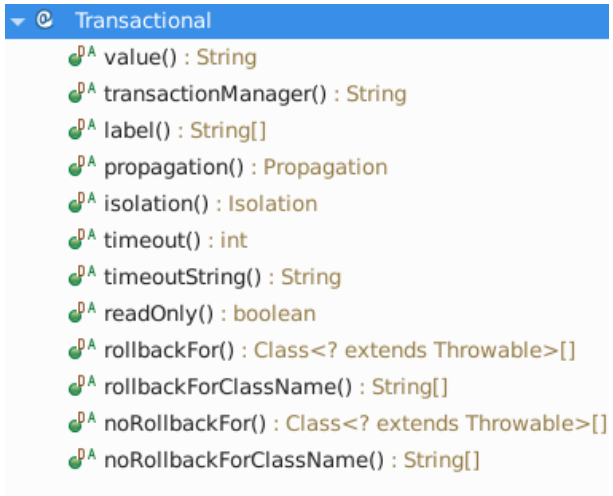
- Transactions are essential with non trivial data accesses
- Java EE supports Transactions through JTA and JTS, the transaction implementation is handled by the application server
- Spring does not require application server, but does support transactions even when running a Spring boot application on a Tomcat server
- For the simple straightforward cases we can use the @Transactional annotation for our methods requiring transactions
  - Use the annotation from org.springframework.transaction.annotation not javax.transaction
- Manual, or programmatic transactions can utilize the e.g. TransactionTemplate class
- Distributed transactions are **not** available in Spring by default

Annotation based transactions

- Annotation based declarative transactions are enabled when using spring-data-* dependencies for your project
  - Spring Boot based projects almost always are, if not the transactions need to be configured and then enabled
  - Alternative to annotations there is also, as always with Spring, a possibility to use XML based configuration
- Use @Transactional annotation to add transaction support to a class, or individual *public* methods
  - Protected and private methods are a little trickier, easiest way to support them is most likely using AspectJ
- @Transactional supports fine tuning: Isolation level, readOnly, Rollback rules, among others

Annotation based transactions

- Just add the annotation to the class
- Configure if needed




```
@Service
@Transactional
public class SomeTransactionalService {
     public void update(Entity entity) {
          // update code here
     }
}
```

- Note that by default rollback takes place ***only*** when a RuntimeException is triggered, not with checked exceptions. To change add rollBackFor

```
@Transactional(rollbackFor = SQLException.class)
public void create(Entity entity) throws SQLException {
     // try to create
}
```

Programmatic transactions

- If annotation is not enough, there is always code
- Create a TransactionTemplate, and implement code requiring transaction in a callback

```
@Service
public class ManualTransactionalService {
    private final TransactionTemplate transactionTemplate;

    public ManualTransactionalService(PlatformTransactionManager transactionManager) {
        this.transactionTemplate = new TransactionTemplate(transactionManager);
    }
    public Object returningMethod() {
        return transactionTemplate.execute(new TransactionCallback() {
            public Object doInTransaction(TransactionStatus status) {
                return "Hello";
            }
        });
    }
    public void noreturnMethod() {
        transactionTemplate.execute(new TransactionCallbackWithoutResult() {
            protected void doInTransactionVoid(TransactionStatus status) {
                // do something
            }
        });
    }
}
```

Additional info

Spring actuator

- Adding spring-boot-starter-actuator dependency to a project will automatically expose endpoints describing e.g. the health of the application
- By default the only enabled endpoint is the health
  - http://localhost:8080/actuator/health
- For more information about the endpoints: https://docs.spring.io/spring-boot/docs/current/reference/html/actuator.html
-
  - DEMO TIME

Spring security

- Spring Security can be enabled by simply adding the spring-starter-security dependency to our project – that will actually secure everything automatically, so we need to configure what/who to let in
- Spring security supports various authentication methods, so e.g. username/password that is not applicable for REST services does not need to be used, instead we can configure e.g. one of the available token based alternatives
-
    - DEMO TIME

The End