Java Programming 3

Application Context - Logging - Intro SpringMVC

KdG

Agenda this week

Last week - project Review
Spring Application Context Details
Spring Boot
Logging
Spring MVC introduction
Spring MVC: My first @Controller
Spring MVC: My first Thymeleaf template

KdG

Agenda this week

Last week - project Review
Spring Application Context Details
Spring Boot
Logging
Spring MVC introduction
Spring MVC: My first @Controller
Spring MVC: My first Thymeleaf template

KdG

Last week - Project Review

- Project Remarks:
  - Be sure to tag your releases with the correct name (sprint1)!

    $ git checkout sprint1 → this should work in your project root folder

  - Be sure to use a correct .ignore file! (.idea .gradle build should be ignored, gradle folder should NOT be ignored!
  - Create 1 project (not a different project for each week…)
  - Be sure the project can be build using the gradle wrapper:

    $ ./gradlew build → this should work in your project root folder…!

KdG

Last week - Project Review

- Project Remarks:
  - I have a Repository interface for my two main entities (ObservationRepository and AstroObjectRepository)
    - The implementation uses a List to store the objects (eg. ListObservationRepository)
    - The DataFactory class fills the repositories using the create… methods of the repositories
  - My entities have an id field that is filled in by the repositories (using java stream API…)
    - It’s a bit to simple, but ok for now…
  - The main classes of my application:
    - Menu - Presenter
    - AstroObjectServiceImpl - ObservationServiceImpl
    - ListAstroObjectRepository - ListObservationRepository
    - JSonWriter
    - DataFactory

KdG

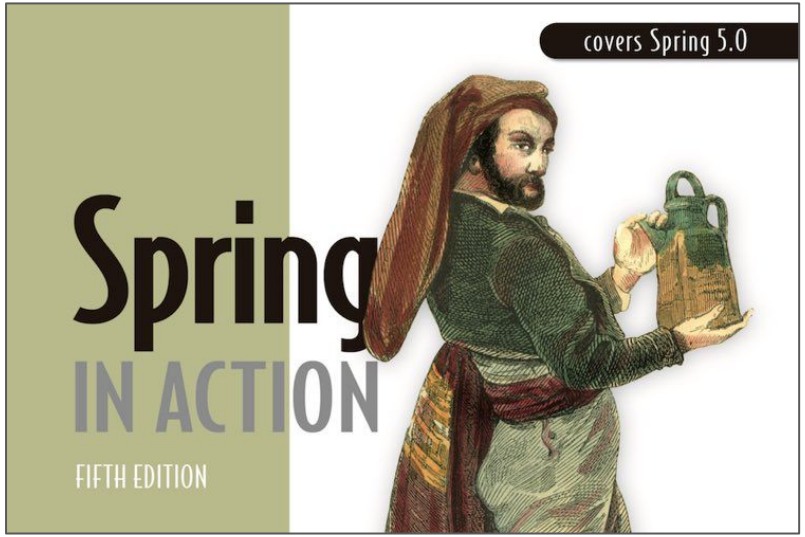
Agenda this week

Last week - project Review
Spring Application Context Details
Spring Boot
Logging
Spring MVC introduction
Spring MVC: My first @Controller
Spring MVC: My first Thymeleaf template

KdG

Spring Documentation…

- Good Spring documentation: https://spring.io/
- Good book about Spring: https://www.manning.com/books/spring-in-action-sixth-edition



KdG

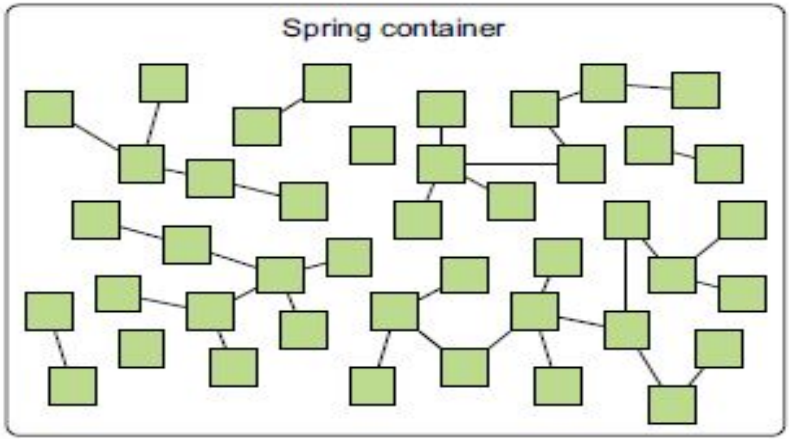
Spring Application Context Details

Enterprise application: is composed of many components that need to work together. Each component is responsible for a part of the functionality.

Components need to be created and work together

Spring Container ("Spring Application Context") creates and manages these components (or *beans*)

This is done by using *Dependency Injection*



KdG

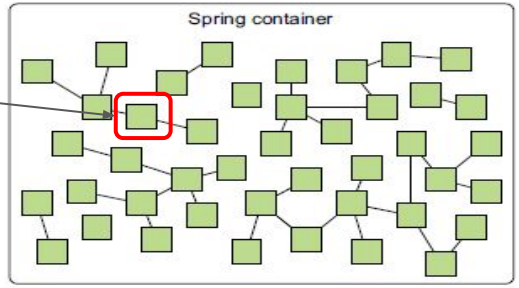
Demo 1: The Application Context

- In this demo we will
  - Create a CalculatorService with implementation
  - Load it into the context using various techniques
  - Demonstrate the use of these annotations:
    i. @Bean
    ii. @Scope
    iii. @Configuration
    iv. @ComponentScan
    v. @Component and @Service
    vi. @Qualifier
    vii. @Profile
    viii. @AutoWired

KdG

@Component

A *component* or *bean* in the Spring Application Context (= container)

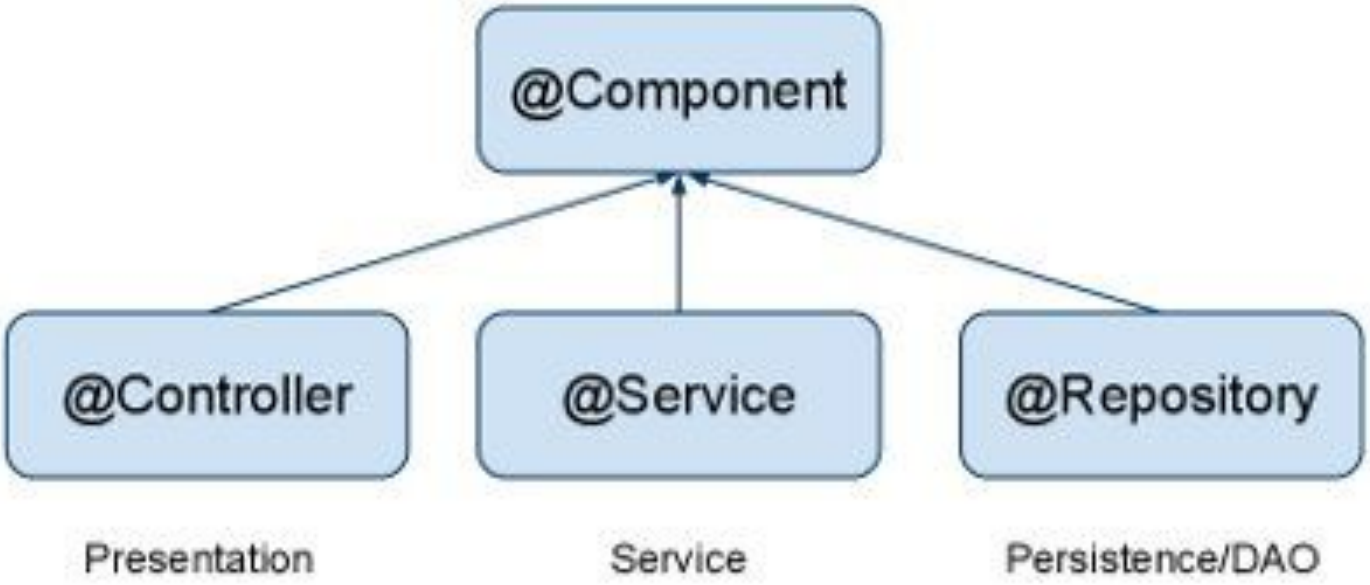


- = Class annotation
- **Use above the implementation classes of your interfaces (eg: StudentServiceImpl)**
- This annotation tells Spring this class should be loaded as a bean in the *application context*.
- Will be detected by Spring via ***component scanning***
- The class needs at least
  - A default constructor
  - Or a constructor with parameters that are annotated as @Component
- You can give the bean an optional name (@Component(“myBean”). (Default name is name of the class starting with small letter.))

KdG

Other stereotypes

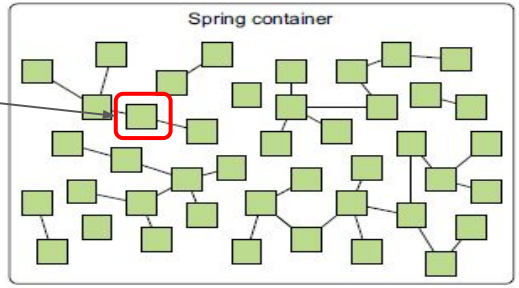
- @Controller, @Service and @Repository inherit from @Component
- They also tell Spring to load these beans into the application context
- They give extra *meaning* to the annotated class



KdG

@Bean

A *component* or *bean* in the Spring Application Context (= container)



- = Method annotation
- **Use for methods in a @Configuration class**
- This annotation tells Spring this method should be used to load the bean in the application context.
- Typically used for beans for which we did not write the class code ourselfs
  - For example: the gson object…
- The name of the method = name of the bean
- You can give the bean an optional name

```
@Configuration
public class
MyApplicationConfiguration{
   @Bean("johnny")
   public Robot myRobot() {
       return new MyRobot();
   }
}
```

KdG

Bean scope

- By default a Bean is created as a Singleton: there exists only 1 object in the container, all request for that bean return the same object
- You can also create a Bean with scope prototype: every requests for the bean returns a new bean object
- Use the @Scope annotation:

```
@Bean
@Scope("singleton)
public Person person() {
    return new Person();
}
```

```
@Bean
@Scope("prototype")
public Person person() {
    return new Person();
}
```

The @Scope annotation can also be added next to @Component

Redundant: scope singleton is the default

KdG

@SpringBootApplication

- This annotation is a combination of
  - @ComponentScan
  - @Configuration
  - @AutoConfiguration
- You typically annotate your main class with @SpringBootApplication
  - All components in current package and subpackages will be scanned
  - You can add @Bean methods to your main class for external beans
  - Spring Boot will autoconfigure your application based on your classpath (for example: if it finds HSQLDB on your classpath, it will autoconfigure the database)
    - https://docs.spring.io/spring-boot/docs/1.3.8.RELEASE/reference/html/using-boot-auto-configuration.html

KdG

@Autowired

- Inject a dependency into a class
- 3 possibilities:
  - **Via constructor (you can leave out the @Autowired)**
  - Via setter (or any other method)
  - Via the attribuut (even private attributes!) → considered bad practice, use only in testclasses
- Spring should find exactly one implementation or the application will not start

  → throws `NoUniqueBeanDefinitionException`

- Use @Autowired(required=false) if it is not a problem when Spring does not find an implementation

KdG

@Autowired

- More than one implementation (`NoUniqueBeanDefinitionException`)?
  - Add @Primary label to one @Component → Spring will load that one
  - Or: add @Qualifier(“<name of preferred bean>”) to the @Autowired annotation
  - Or: use @Profile → we will look at profiles in a future lesson

```
@Component("fooStudentRepo")
public class FooStudentRepo implements StudentRepo {
   //...
}

@Component("barStudentRepo")
public class BarStudentRepo implements StudentRepo {
  //...
}
```

```
public class FooService {

    @Autowired
    @Qualifier("fooStudentRepo")
    private StudentRepo repo;
}
```

KdG

Exercise 1

- Create a new project Robots (Spring project!)
  - Create an interface Robot with one method: sayHello()
  - Create an implementation of the Robot: MyRobot
  - Create a configuration class RobotConfiguration
- Let the Spring Container create a MyRobot bean in different ways (and test by getting the bean from the context and running the sayHello() method):
  - With @Bean annotation → in the RobotConfiguration class…
  - With @Component annotation (in the MyRobot class) → In RobotConfiguration: comment the @Bean, add @ComponentScan!
  - Test @Scope (“prototype”) versus (“singleton”)→ can you show the difference?
- Create a second implementation MyLoudRobot, it shouts hello in capitals
  - Add @Component annotation → you get an error when running now, inspect
    - Solve using @Primary and using @Profile
- Create a class RobotRoom, it has 2 robots as attributes
  - @Autowire via setters (not via constructor)
  - Use @Qualifier to select different implementations

KdG

Quiz: explain the use of these annotations!

- @Component
- @Bean
- @Autowire
- @Scope
- @Primary
- @Qualifier
- @ComponentScan
- @Configuration
- @Service
- @Profile
- @Controller
- @Repository

KdG

Agenda this week

Last week - project Review
Spring Application Context Details
Spring Boot
Logging
Spring MVC introduction
Spring MVC: My first @Controller
Spring MVC: My first Thymeleaf template

KdG

What is *Spring Boot*?

Spring Boot is a Spring Project that makes it really easy to create Applications that *just run:*

- Features:
  - Create **stand-alone Spring applications**
  - **Embed Tomcat,** Jetty or Undertow directly (no need to deploy WAR files)
  - Provide opinionated '**starter' dependencies** to simplify your build configuration
  - **Automatically configure** Spring and 3rd party libraries whenever possible
  - Provide production-ready features such as **metrics, health checks, and externalized configuration**
  - Absolutely **no code generation** and no requirement for XML configuration

https://spring.io/guides/gs/spring-boot/

https://docs.spring.io/spring-boot/docs/current/reference/html/index.html

https://www.baeldung.com/spring-vs-spring-boot

Demo 2: Spring Boot

- In this demo we will
  - Make a Spring Boot application with Spring MVC and Thymeleaf dependencies of the SMS demo app
  - Run it: a webserver is running
  - Create the jar using gradle bootJar
  - Inspect this jar

KdG

Exercise 2

- Create a Spring application with Spring Web dependency
- Run the application: what happens?
- Use the Gradle bootJar task to create a runnable jar
- Extract this jar and inspect the contents. What do you find?
  - A lot of library classe are added!
  - There is even a complete webserver included (Tomcat)...!
- Try running the application standalone (no IntelliJ)
  - In command prompt enter: “`java -jar demospringweb-0.0.1-SNAPSHOT.jar --debug`”
  - Surf to localhost:8080 → the webapplication serves a page (“whitelabel error page”)

KdG

Agenda this week

Last week - project Review
Spring Application Context Details
Spring Boot
Logging
Spring MVC introduction
Spring MVC: My first @Controller
Spring MVC: My first Thymeleaf template

KdG

Logging

- Up until now we used `System.out.println` as our ‘log’-messages
- This is considered **bad practice**!
- There exists different log libraries that are much better suitable for this task:
  - In standard JDK: java.util.logging
  - Log4J
  - Logback
  - …
- Spring Boot uses Logback as it’s default logging library

KdG

Logging in Spring: documentation

https://docs.spring.io/spring-boot/docs/current/reference/html/features.html#features.logging

https://www.baeldung.com/spring-boot-logging

https://www.baeldung.com/logback

https://stackify.com/compare-java-logging-frameworks/

KdG

Demo 3: Logging

- In this demo we will
  - Add logging to the SMS application
  - Demonstrate the loglevels
  - Demonstrate the configuration
    i. Using application.properties
    ii. Using logback-spring.xml

KdG

Logging example

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


@Component
public class LogTestrunner implements CommandLineRunner {
   private Logger logger = LoggerFactory.getLogger(LogTestrunner.class);
   @Override
   public void run(String... args) throws Exception {
       logger.trace("A TRACE Message");
       logger.debug("A DEBUG Message");
       logger.info("An INFO Message");
       logger.warn("A WARN Message");
       logger.error("An ERROR Message");
   }
}
```

Simple Logging Facade 4 Java: facade for different logging frameworks, makes it easy to switch to other logging framework.
You can also use the Logback packages:

```
import org.apache.logging.log4j.Logger;
import org.apache.logging.log4j.LogManager;
```

You create a logger at the top of each class and pass it the class of the Class you are in.

You can then use different methods to log at different *loglevels*

Parameterized messages

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


@Component
public class StudentServiceImpl {
   private Logger logger = LoggerFactory. getLogger(LogTestrunner.class);

   //…
   @Override
   public void save(Student student) throws Exception {
       logger.debug("Saving student {} to the repository…", student);
       //…
   }
}
```

You often like to log information of certain objects. Using {} in the string and extra parameters will run the toString of the parameter and insert it into the string.
Logback ensures this toString is only run when necessary!

In here the toString of student is not called if the loglevel is to info or higher…

KdG

Running the example:

Messages logged by the Spring Framework

```
| '_| | '_ \/ _` | \ \ \ \
| | | | | || (_| |  ) ) ) )
|_| |_|_| |_\__, | / / / /
============|___/=/_/_/_/
 ::                (v2.5.5)

5:48.949  INFO 16856 --- [           main] b.k.j.l.LoggingDemoApplication          : Starting LoggingDem
5:48.949  INFO 16856 --- [           main] b.k.j.l.LoggingDemoApplication          : No active profile s
5:49.453  INFO 16856 --- [           main] b.k.j.l.LoggingDemoApplication          : Started LoggingDemo
5:49.457  INFO 16856 --- [           main] be.kdg.java2.logging_demo.LogTestrunner  : An INFO Message
5:49.457  WARN 16856 --- [           main] be.kdg.java2.logging_demo.LogTestrunner  : A WARN Message
5:49.457 ERROR 16856 --- [           main] be.kdg.java2.logging_demo.LogTestrunner  : An ERROR Message
```

Messages logged by you.
Trace and Debug log messages are not shown by default

KdG

Configuration of Logging

- Zero configuration: Spring Boot uses *opinionated* configuration
- You only need Apache Commons Logging as dependency in build.gradle, but already part of spring-boot-starter dependency!

KdG

Log Levels

- The Level tells *how important* the message is
- From low to high:
  - ERROR
  - WARN
  - INFO
  - DEBUG
  - TRACE
- You can set the loglevel of your application: it defines what messages will be logged.
- For example: setting the loglevel to WARN will only log WARN and ERROR messages. Setting loglevel to DEBUG will only log DEBUG, INFO, WARN and ERROR messages

KdG

Change the loglevel

- Different possibilities
  - Via VM Options
  - Via gradle
  - Via Logback configuration file
  - **Via application.properties**
  - **Via logback XML configuration file**
- Using **application.properties** file:

```
logging.level.be.kdg.programming3 =DEBUG
```

This file can be found in src/main/resources

Log level of all messages from classes in be.kdg.programming3 package (or subpackages) is set to DEBUG

KdG

Other configurations?

- You can log to files
- You can change the output pattern
- You can log to different sources
- You can change the color of the logmessages
- You can rotate over different logfiles
- …

→ Best way: put a **logback-spring.xml** file in the resources folder.

KdG

Example logback-spring.xml file (part 1/3)

```
<?xml version="1.0" encoding="UTF-8" ?>
<configuration>

   <property name="LOGS" value="./logs" />

   <appender name="Console"
             class="ch.qos.logback.core.ConsoleAppender">
       <layout class="ch.qos.logback.classic.PatternLayout">
           <Pattern>
               %black(%d{ISO8601}) %highlight(%-5level) [%blue(%t)]
%yellow(%C{1}): %msg%n%throwable
           </Pattern>
       </layout>
   </appender>

   ...
```

Define different appenders for different outputs. This one outputs to the Console.

Example logback-spring.xml file (part 2/3

This appender outputs to a file. It creates a new file every day or if the file > 10 Meg

```
  <appender name="RollingFile"
            class="ch.qos.logback.core.rolling.RollingFileAppender">
      <file>${LOGS}/spring-boot-logger.log</file>
      <encoder
              class="ch.qos.logback.classic.encoder.PatternLayoutEncoder">
          <Pattern>%d %p %C{1} [%t] %m%n</Pattern>
      </encoder>

      <rollingPolicy
              class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
          <!-- rollover daily and when the file reaches 10 MegaBytes -->

<fileNamePattern>${LOGS}/archived/spring-boot-logger-%d{yyyy-MM-dd}.%i.log
          </fileNamePattern>
          <timeBasedFileNamingAndTriggeringPolicy
                  class="ch.qos.logback.core.rolling.SizeAndTimeBasedFNATP">
              <maxFileSize>10MB</maxFileSize>
          </timeBasedFileNamingAndTriggeringPolicy>
      </rollingPolicy>
  </appender>
```

Example logback-spring.xml file (part 3/3)

```
    <!-- LOG everything at INFO level -->
    <root level="info">
        <appender-ref ref="RollingFile" />
        <appender-ref ref="Console" />
    </root>

    <!-- LOG "be.kdg.programming3*" at TRACE level -->
    <logger name="be.kdg.programming3" level="trace" additivity="false">
        <appender-ref ref="RollingFile" />
        <appender-ref ref="Console" />
    </logger>

</configuration>
```

Everything is logged at INFO level, using the 2 appenders. Logmessages from be.kdg.programming3 are logged at TRACE level

This file is a nice example, you can use it as a starting point to configure your own logging...

KdG

Exercise 3

- Use the Robot project from the first exercise
- Add a logger to the MyRobot class
  - Add DEBUG logging messages to the constructor and the sayHello methods
- Add a logger to the RobotApplication class
  - Add an INFO logging message to the main method
- Try to set the log level using the application.properties
  - To ERROR → check the logmessages
  - To TRACE → check the logmessages
  - Now remove the loglevel from application.properties → check the logmessages
- Add the logback-spring.xml file to the resources folder
  - Run the application → check the logfile in the logs folder
- Try changing the logback-spring.xml file
  - Set the loglevel of the MyRobot class to DEBUG → check the logmessages
  - Change the timestamp of the logmessages in the console to green

KdG

Agenda this week

Last week - project Review
Spring Application Context Details
Spring Boot
Logging
Spring MVC introduction
Spring MVC: My first @Controller
Spring MVC: My first Thymeleaf template

KdG

Spring MVC

- Spring project to build *dynamic web applications*

KdG

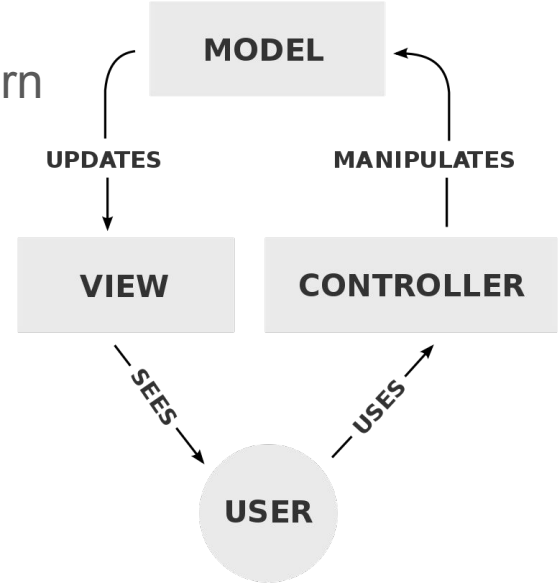
Web Application?

- Application that runs on a **Web Server**
- is accessed via **Web Browser**
- **Web Server:**
  - Listens to HTTP request, returns answers in the form of
    - HTML/CSS/JavaScript or JSON (REST Server)
- Web server can serve *dynamic content*: HTML, JSON, … etc is generated at runtime using *server-side programming*
- Popular Web Server frameworks: Node.js (JavaScript), IIS (C#), Tomcat (Java), ...

KdG

Spring MVC: **M**odel **V**iew **C**ontroller

- **Design pattern** used in many applications that have a user interface
- **Part of the presentation layer** of your application
- Separates the View code from the Model code and uses a Controller as mediator
- Somewhat **similar to the Model View Presenter** pattern
- *Many different "versions" of this design pattern exists..*



KdG

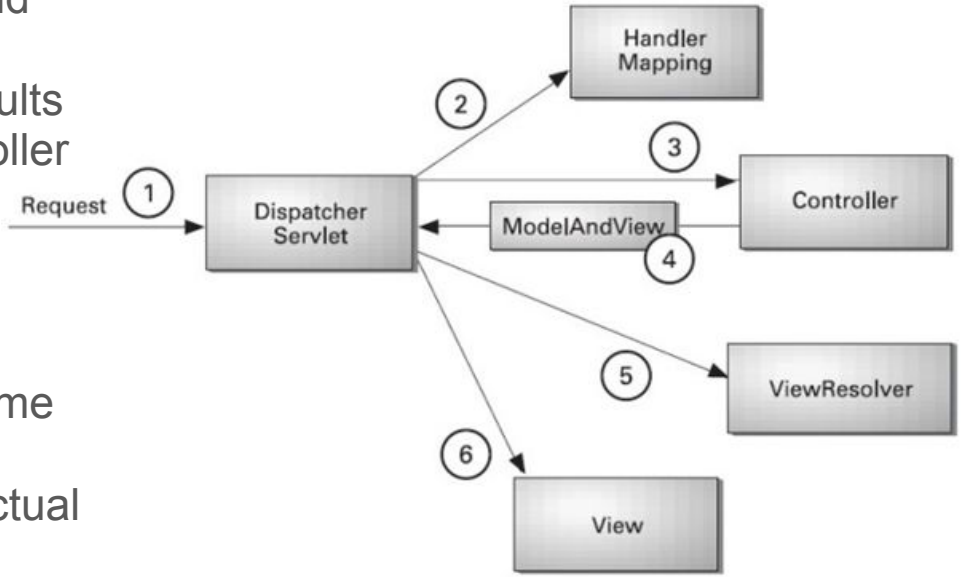
Spring MVC: **M**odel **V**iew **C**ontroller

- The Spring MVC framework uses (it's own version of) the MVC design pattern:
  - The Controller :
    - **Talks to the Service layer**
    - Gathers data into a Model object
    - Passes this Model to the correct View
  - The View:
    - **Generates the UI** (HTML/CSS pages) using the data from the Model
    - We will use **Thymeleaf templates** for this
  - The Model:
    - Object that contains the data needed by the View to be able to generate the UI

KdG

Spring MVC: the life of a request

1. Request from browser, contains URL and additional information
2. DispatcherServlet: front controller, consults HandlerMapping to decide which Controller will handle the request
3. Controller receives the request and processes it (using services from the service layer)
4. Sends collected data and the logical name of a View to the DispatcherServlet
5. He asks the ViewResolver who is the actual view.
6. Sends the model data to this view who will render the output that will be send back to the client



KdG

Configure Spring MVC

- Configure DispatchServlet
- Configure ViewResolver
- Configure HandlerMappi
- Set up the Th
- ...

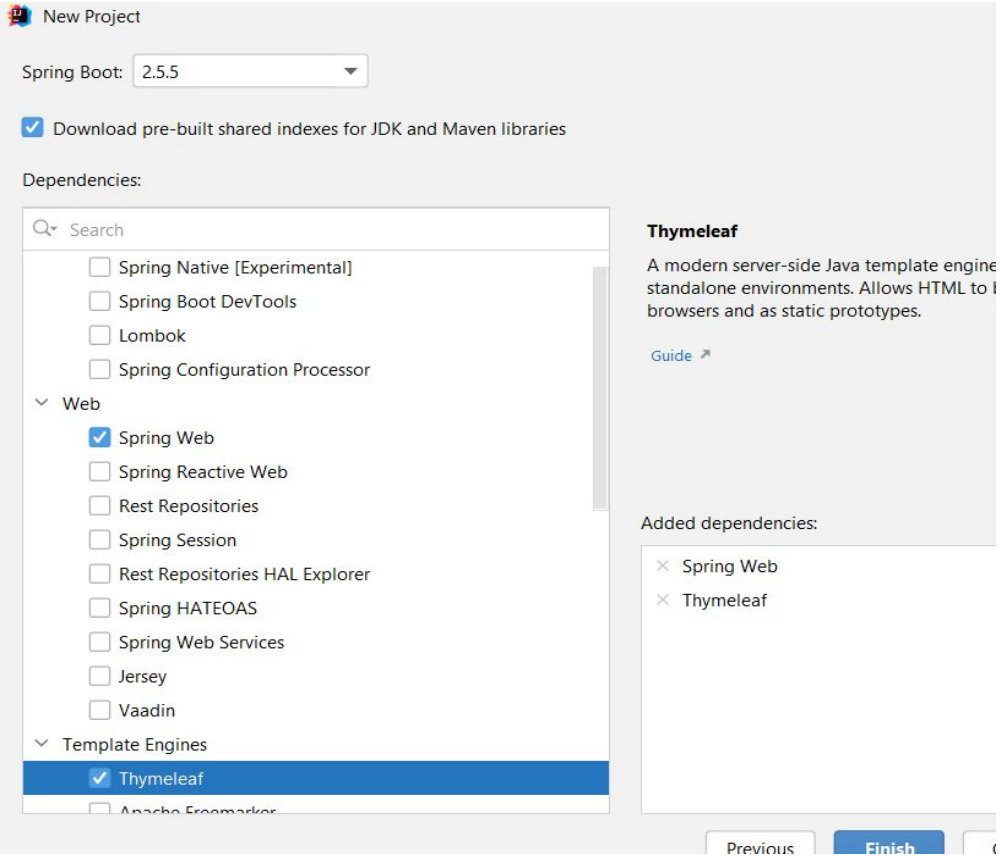
Spring Boot to the rescue!

Just create a Spring project in IntelliJ with Spring Web dependency and Spring Boot will configure and create those beans automagically!

KdG

Create Spring MVC Project

- Use Spring Initializr and select
  - Spring Web dependency
  - Thymeleaf



KdG

Spring MVC build.gradle

- Thanks to the spring-boot-starter-.. dependencies, all required libraries are included
- And:
  - Spring Boot will autoconfigure your project as a Spring MVC project!
  - Spring Boot will autoconfigure Thymeleaf as the template engine...

```
dependencies {
   implementation 'org.springframework.boot:spring-boot-starter-thymeleaf'
   implementation 'org.springframework.boot:spring-boot-starter-web'
   testImplementation 'org.springframework.boot:spring-boot-starter-test'
}
```

KdG

Agenda this week

Last week - project Review
Spring Application Context Details
Spring Boot
Logging
Spring MVC introduction
Spring MVC: My first @Controller
Spring MVC: My first Thymeleaf template

KdG

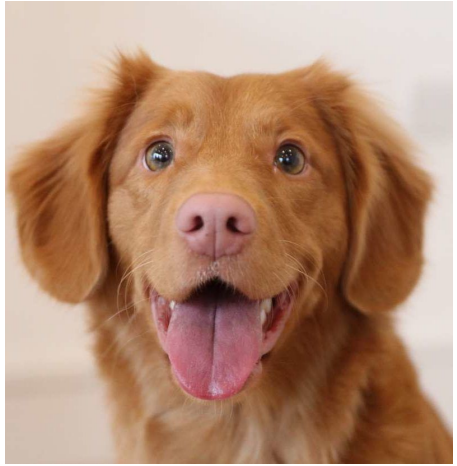
Demo 4: Spring MVC - The Controller

- In this demo we will
  - Create a Controller for the SMS application
    i. Annotate with @Controller
    ii. Annotate with @RequestMapping
  - Create a showStudents method in it
    i. Annotate with @GetMapping
    ii. Return the logical view name
  - Create a html page as a first View

KdG

Let’s create a Dog Cloud application

- We will make a small Spring web application (“Dog Cloud”) to be able to list all my dogs online and be able to add new dogs



→ download this start project and inspect!

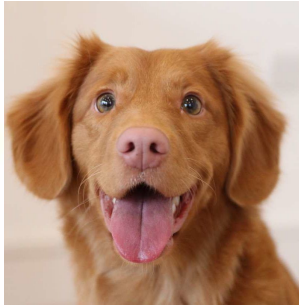
https://gitlab.com/kdg-ti/programming-3/exercises/dogcloud

KdG

Dog Cloud Domain Layer

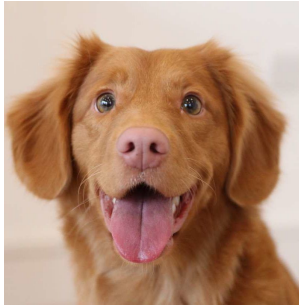
- Domain layer has 1 class: Dog

Dog	
id	int
name	String
type	Type
Dog(String, Type)	
getId()	int
getName()	String
getType()	Type
setId(int)	void
setName(String)	void
setType(Type)	void
toString()	String



KdG

Dog Cloud Repository and Service layers



- Small hardcoded repository ListDogRepository that holds dogs in a List. The repository has 2 methods:

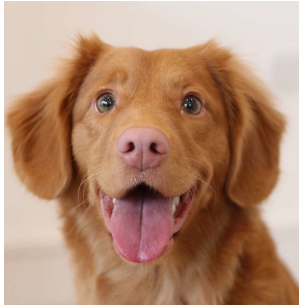
```
public interface DogRepository {
   Collection<Dog> readDogs();
   Dog createDog(Dog dog);
}
```

- We use a CommandLineRunner to seed it with some data…
- The DogServiceImpl implements this interface:

```
public interface DogService {
   List<Dog> getAllDogs();
   Dog addDog(String name, Dog.Type type);
}
```

KdG

Dog Cloud Presentation Layer



- Menu class: shows all dogs on the console…

```
public void show() {
   System.out.println("List of all dogs:");
   dogService.getDogs().forEach(System.out::println);
}
```

→ Let’s convert the application into a dynamic web application!

KdG

Exercise 4: add a Controller to the Presentation Layer

- Will handle the HTTP request and and hand it over to a view to render HTML

```
@Controller
@RequestMapping("/dogs")
public class DogController {
   // inject a dogService via constructor
   @GetMapping
   public String showDogsView(Model model){
       model.addAttribute("dogs", dogService.getDogs());
       return "dogs";
   }
}
```

If the application receives a GET request for /dogs, it will run this method

You get a Spring Model object, you can add data to it to pass on to the view

Try running this: what happens?

The method returns the logical **view name.** The DispatcherServlet will use the ViewResolver to find the real View

KdG

Server listening on port 8080

```
main] b.k.java2.dogcloud.DogcloudApplication   : Starting DogcloudApplication using Java 11.0.11
main] b.k.java2.dogcloud.DogcloudApplication   : No active profile set, falling back to default p
main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat initialized with port(s): 8080 (http)
main] o.apache.catalina.core.StandardService   : Starting service [Tomcat]
main] org.apache.catalina.core.StandardEngine  : Starting Servlet engine: [Apache Tomcat/9.0.53]
main] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring embedded WebApplicationConte
main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization compl
main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port(s): 8080 (http) with cont
main] b.k.java2.dogcloud.DogcloudApplication   : Started DogcloudApplication in 1.606 seconds (JV
main] b.kdg.java2.dogcloud.bootstrap.SeedData  : Seeding repository with some dogs...
```

KdG

Change port?

- Change in application.properties:

```
server.port=8081
```

```
main] b.k.java2.dogcloud.DogcloudApplication    : Starting DogcloudApplication using Java 11.0.11 on
main] b.k.java2.dogcloud.DogcloudApplication    : No active profile set, falling back to default pro
main] o.s.b.w.embedded.tomcat.TomcatWebServer   : Tomcat initialized with port(s): 8081 (http)
main] o.apache.catalina.core.StandardService    : Starting service [Tomcat]
main] org.apache.catalina.core.StandardEngine   : Starting Servlet engine: [Apache Tomcat/9.0.53]
```

KdG

Surf to localhost:8081/dogs

- Standard Spring errorpage:

We didn’t define a View yet!

localhost:8081/dogs

Whitelabel Error Page

This application has no explicit mapping for /error, so you are seeing this as a

Mon Oct 04 00:03:17 CEST 2021
There was an unexpected error (type=Internal Server Error, status=500).

KdG

Controller: other example

```
@Controller
public class AccountController {
   //Code left out..

   @GetMapping("/accounts")
   public ModelAndView showAccountBalance(){
       final ModelAndView modelAndView = new ModelAndView();
       modelAndView.setViewName("showAllAccounts");
       modelAndView.getModel().put("accounts", accountsService.getAllAccounts());
       return modelAndView;
   }
}
```

A Get on /accounts will run this method

You can also return a ModelAndView object. It contains the logical view name and the data

KdG

Spring Model

https://www.baeldung.com/spring-mvc-model-model-map-model-view

- Spring Model != Domain Model (DM)
- Spring Model contains information from DM for a view
- Different flavours:
  - Model: provides attributes to the view
  - ModelMap: same as Model but in the form of a Map
  - ModelAndView: same as ModelMap but includes the logical view name

KdG

Controller annotations

- @GetMapping, @PostMapping, @PutMapping, @DeleteMapping, @PatchMapping
  - Typically for each mapping a method, for each entity a controller
- Before Spring 4.3:
  - @RequestMapping(method = RequestMethod.GET)

→ Still useful at Class level!

KdG

Exercise 5

- Create an HTML file named dogs.html in main/resources/templates
- Put an <h1>No Dogs Yet</h1> in it
- Rerun the application, does it work?
- Add the following functionality:
  - Surfing to /dogs/info should return an HTML page saying: “No info on the dogs yet!”:
    - Create a new method in your DogController for this
    - Add the @GetMapping annotation with “/info” as parameter
    - Return the logical name of the view (eg dogsinfo)
    - Create the dogsinfo.html page in the resources/templates folder
  - Surfing to /cats should return an HTML page saying: “It’s no cat cloud!”
    - create an extra controller class (CatsController) for this!

KdG

Agenda this week

Last week - project Review
Spring Application Context Details
Spring Boot
Logging
Spring MVC introduction
Spring MVC: My first @Controller
Spring MVC: My first Thymeleaf template

KdG

Thymeleaf

- Thymeleaf is a template engine
- Other examples of template engines:
  - Java: JSP / JSTL / Apache Tiles
  - .Net: Razor
- https://www.thymeleaf.org/
- Version 3.1:
  - https://www.thymeleaf.org/doc/tutorials/3.1/usingthymeleaf.html
  - https://www.thymeleaf.org/doc/tutorials/3.1/thymeleafspring.html

KdG

Thymeleaf example

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
   <meta charset="UTF-8">
   <title>Dog Cloud</title>
</head>
<body>
<h1>Number of dogs: <span th:text="${info}"></span></h1>
</body>
</html>
```

You need to add this nampespace to your HTML. That way you can use the th: attributes.

Thymeleaf uses th: attributes. When the template is run, it will execute the th:attributes to alter the HTML...

KdG

Demo 5: Spring MVC - The View

- In this demo we will
  - Create a Thymeleaf template to show the students in a table
  - Create a Thymeleaf template to show the addstudents form
  - Create a method in the Controller to show this View
  - Create a method in the Controller to process the POST of the form

KdG

Exercise 6

- Add the thymeleaf template for doginfo and make sure it shows the correct number of dogs…
- Read section 5 in this tutorial https://www.baeldung.com/thymeleaf-in-spring-mvc
- Now create a template to show all dogs (name and type) in a small HTML table

KdG

Read data from a form...

- Let’s create a small form to add a dog (adddog.html):

```
<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="UTF-8">
   <title>Dog Cloud</title>
</head>
<body>
<form method="post">
    <label for="name">Name:</label><input type="text" id="name" name="name"/>
    <input type="submit" value="Submit"/>
</form>
</body>
</html>
```

This form will POST the form data to the webserver. Let’s write a method to process the post request!

KdG

Add 2 methods to the DogController

```
@GetMapping("/adddog")
public String showAddDogForm(){
   return "adddog";
}

@PostMapping("/adddog")
public String processAddDog(Dog dog){
   dogService.addDog(dog.getName(), dog.getType());
   return "redirect:/dogs";
}
```

A POST on /add will run this method. The HTTP body will be mapped onto the Dog object. We save the dog to the repository

Spring will magically map the form values into a dog object…

We redirect to /dogs: the /dogs GET will be run again.

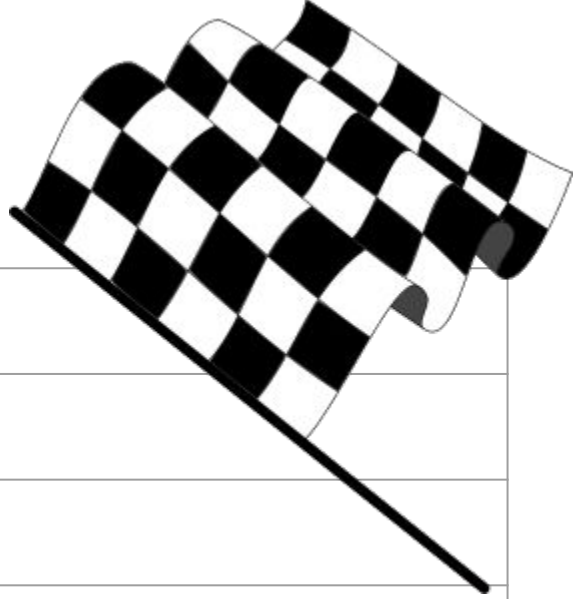
KdG

Exercise 7

- Add the adddog functionality to your application
- Now alter the form: the user must be able to also choose a dog type from a dropdown list (<select>)
- Do the necessary changes to the different controller methods to get everything up and running

KdG

Agenda this week



Last week - project Review
Spring Application Context Details
Spring Boot
Logging
Spring MVC introduction
Spring MVC: My first @Controller
Spring MVC: My first Thymeleaf template

KdG

Project

You will turn your application into a **web application** (in what follows the 2 main entities are book and author, you use your entities of course)

- Your application has **4 pages**: books, authors, addbook and addauthor
- Every page has a menu on top to switch to the other pages:
  - All Books
  - All Authors
  - Add Book
  - Add Author
- **All Books**: shows a table with all the books, showing for each book all the attributes of the book (not the relationship attributes)
- **All Authors:** shows a table with all the authors, showing for each author all the attributes of the author (not the relationship attributes)
- **Add Book**: shows a form to add a new book. When submitting the book is added to the repository.
- **Add Author:** shows a form to add a new Author. When submitting the author is added to the repository.
- You add a logger to each class. You **log debug messages** in all ‘complex’ methods (not in getters and setters). **Change the loglevel** to debug via the application.properties file.

KdG