* Introduction to Spring Boot

- *Spring Boot* is a Spring project that aims to make it easier to develop and run Spring applications
  - a Spring application can require a large amount of configuration metadata – even if you use components and autowiring
  - Spring Boot simplifies application development, because it automatically configures them (where possible), based on "smart" default values – a Spring Boot application usually requires only minimal configuration
  - to this end, Spring Boot uses an "opinionated" approach ("conventional", based on own opinions and conventions)
  - the default choices can still be overwritten using explicit configurations
  - Spring Boot also provides options forbuilding and deploying applications into production

Spring Boot Essentials

Some main features of Spring Boot – are related to different aspects

- starter dependencies – automatic configuration of application libraries and dependencies (not to be confused with dependency injection)
- automatic configuration of beans and components, and the relationships between them (even of beans not explicitly declared) – also on the basis of the dependencies (libraries) used
- actuator – to inspect a running Spring Boot application

- A first simple example

- The development of a Spring Boot application begins with the creation of a project, the structure of its folders, as well as a file for the construction (build) of the application (e.g., Gradle or Maven)
  - this activity can be carried out with Spring *Initializer* – https://start.spring.io/ – starting from the choice of the type of application (and the corresponding starter dependencies) – e.g., a web application
  - the Initializer generates the project, which can be downloaded on own computer
  - the project contains the folder structure the class for the application a test class a file for the application properties (empty), and the Gradle or Maven build file
  - the Initializer can also be accessed by a plugin of the IDE
  - at this point you are ready to start developing the application

Files and folders generated by Spring Initializer

Here is the structure of files and folders generated by the Initializr for a minimal application

```
|
+--- build.gradle  (or  pom.xml)
+--- settings.gradle  (with project name)
\--- src
        +--- main
        |   +--- java
        |   |   \--- asw.springboot.hello
        |   |    \helloapplication ---.java
        |   \--- resources
        |       \--- application.properties
        \--- test
                \--- java
                        \--- asw.springboot.hello
                                \--- HelloApplicationTests.java
```

Dependency Management

In general, Spring applications depend on thepresence of certain libraries (jar files, in specific versions) in the application's classpath – for compilation, execution and/or testing

- e.g., Spring applications typically depend on the spring-core and spring-context libraries (the latter deals with the injection of dependencies)
- a dependency can be *direct* (the application depends on X) or *indirect* or *transitive* (if the application depends on X and X depends on Y, then the application also depends on Y)
- transitive dependencies are the most difficult to identify and manage

Starter dependencies

- Spring Boot simplifies dependency management by providing and officially supporting a curated set of starter dependencies
  - a *starter* dependency is a *dependency* (usually broad and transitive) – whose inclusion automatically impliesthe inclusion of its transitive dependencies
  - thanks to them, a Spring Boot application usually requires few dependencies – in the example, spring-boot-starter and springboot-

starter-test – while a traditional Spring application requiresa dozen dependencies or more

- e.g., spring-boot-starter implies spring-boot,whichimplies spring-core and spring-context (which deals with the injection of dependencies), as well as some dependencies for logging
- In addition, spring-boot-starter-test implies to (transitively) some fundamental dependencies commonly used for testing – such as junit (for unit tests), mockito-core (for integration tests) and hamcrest-core (for assertions)

A Spring Boot application

This is the main class for the application (default)

```
package asw.springboot.hello;

import org.springframework.boot.SpringApplication; import
org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication public class
HelloApplication {

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

}
```


Running a Spring Boot Application

- How does a Spring Boot application run?
  - the **SpringApplication** class (of Spring Boot) is used to "bootstrap" and boot the application – among other things, it creates an application context for the application

- the annotation **@SpringBootApplication** used for the main class of the application indicates a combination of
- **@Configuration –** which labels the class as a Java configuration class
- **@ComponentScan –** which enables automatic scanning and identification ofcomponents/beans
- **@EnableAutoConfiguration –** takes care of the automatic creation of missing or necessary components/beans (according to Spring Boot), based on the application context and the dependencies specified for the application

Running the Hello application

The application can be run, as it is

- you can build the application jar and then run it with java -jar hello.jar
- in this case, the application simply generates a log and terminates

```
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot :: (v2.2.6.RELEASE)

2020-03-27 15:09:30.167 INFO 26646 --- [ main] asw.springboot.hello.HelloApplication : Starting HelloApplication on workstation with PID 26646
2020-03-27 15:09:30.168 INFO 26646 --- [ main] asw.springboot.hello.HelloApplication : No active profile set, falling back to default profiles: default
2020-03-27 15:09:30.580 INFO 26646 --- [ main] asw.springboot.hello.HelloApplication : Started HelloApplication in 0.726 seconds (JVM running for 1.024)
```


Specific behavior

- Per specify code to run with the application, you can define a component/bean as the following

```
package asw.springboot.hello;

import org.springframework.stereotype.Component; import
org.springframework.boot.CommandLineRunner;

@Component
public class HelloRunner implements CommandLineRunner {

    public void run(String[] args) {
        System.out.println("Hello, world!");
    }

}
```

```
.   ____          _            __ _ _
/\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
\\/  ___)| |_)| | | | | || (_| |  ) ) ) )
'  |____| .__|_| |_|_| |_\__, | / / / /
=========|_|==============|___/=/_/_/_/
:: Spring Boot :: (v2.2.6.RELEASE)

2020-03-27 15:11:40.315 INFO 26729 --- [ main] asw.springboot.hello.HelloApplication : Starting HelloApplication on workstation with PID 26729
2020-03-27 15:11:40.317 INFO 26729 --- [ main] asw.springboot.hello.HelloApplication : No active profile set, falling back to default profiles: default 2020-03-27 15:11:40.734 INFO 26729 --- [ main]
asw.springboot.hello.HelloApplication : Started HelloApplication in 0.72 seconds (JVM running for 1.018) 2020-03-27 15:11:40.735 INFO 26729 --- [ main] class asw.springboot.hello.HelloRunner
: Hello, world!
```


Purpose of a Spring Boot Application

Intuitively, the purpose of a Spring Boot application is to make sure that all the components/beans of the application are created and started/activated.

- these components should generally be defined separately from the main class of the application (the one noted **@SpringBootApplication**)

* Web applications with Spring Web MVC

- Let's now consider developing a web application with Spring Boot – and the Spring Web MVC framework – starting with a very simple application
  - with Spring Initializr the "Web" dependency must be selected
  - this uses the springboot-starter-web starter dependency (instead of the more generic spring-boot-starter)

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

The web starter dependency

Here are some implications of the spring-boot-starter-web dependency

- the spring-boot-starter-web dependence implies springboot-starter (already discussed before), which as we know implies spring-core and spring-context (for the injection of dependencies)
- the spring-boot-starter-web dependency also implies springwebmvc (the k Spring Web MVC framewor,discussed below)
- moreover, spring-boot-starter-web implies tomcat-embed-core – therefore the application can be run on an embedded Tomcat application server
- this is a conventional choice ("opinionated") of Spring Boot – which can be overwritten and modified (but in this case, for simplicity, we are fine with that)

Structure of a web application

- In a web application, two folders (initially empty) are also created between the resources for the static (static) and dynamic (templates) contents of the application

```
|
+--- build.gradle  (or  pom.xml)
+--- settings.gradle
\--- src
        +--- main
        |   +--- java
        |   |   \--- asw.springboot.web.hello
        |   |       \helloapplication ---.java
        |   \--- resources
        |       +--- static
        |       +--- templates
        |       \--- application.properties
        \--- test
                \--- java
                        \--- asw.springboot.web.hello
                                \--- HelloApplicationTests.java
```


Main class for the application

The main class for the application is still as before

```
package asw.springboot.web.hello;

import org.springframework.boot.SpringApplication; import
org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication public class
HelloApplication {

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

}
```

- in fact the purpose of this application is simply to make sure that all the components/beans of the application are created and started/activated

Applicationproperties

- Spring Boot applications can be configured using property files – where you can specify both common Spring Boot properties (e.g., server.port)and application-specific properties
  - application properties must be specified in the application.properties file (or in the application.ymlfile, with a different syntax)
  - example of application.properties file

application.properties server.port=8080

  - in fact, Spring Boot defines default values (often sensible) for common properties – and therefore it is not always necessary to configure all the properties of the application
  - e.g., the default value of the server.port property is just 8080 – but, if desired, it can be modified

Running the Hello web application

The web application can run as is, even without any additional component/bean

- in thiscase, from the log generated by the application it can be deduced that
  - tomcat starts on port 8080
  - inside it runs our simple web application
  - handlers associated with URLs and paths have also been defined
  - the applicationis ready to accept requests
  - for now, being the application empty, access to http://localhost:8080/ leads to an error page generated by Tomcat – this indicates that Tomcat is actually listening
  - alternatively, the application can be assembled as a WAR and released to a separate application server

Test class for the application

- Here is the test class defined by Spring Initializr
  - is just a sample test skeleton class
  - the test (even if empty) checks if the loading of the application context takes place without problems – e.g., that all beans can be initialized

```
package asw.springboot.web.hello; import org.junit.jupiter.api.Test;
import org.springframework.boot.test.context.SpringBootTest;

@SpringBootTest public class
HelloApplicationTests {

    @Test public void contextLoads() {
    }

}
```


- Customization of the Hello application

Here is a simple component that is a web controller (by virtue of **@Controller)**to accept requests to the /hello path

```
package asw.springboot.web.hello;

import org.springframework.stereotype.Controller; import
org.springframework.web.bind.annotation.RequestMapping; import
org.springframework.web.bind.annotation.ResponseBody;

@Controller public class
HelloController {

    @RequestMapping("/hello") public
    @ResponseBody String hello() { return
    "Hello, world!";
    }

}
```

- a GET request http://localhost:8080/hello returns the string Hello, world!

Customizing the Hello application

- Some indications to understand this class
  - the **@Controller** annotation indicates a type of **@Component** – a Spring Web MVC controller, to receive web requests
  - the annotation **@RequestMapping** binds a web controller method to an HTTP operation (GET, default) for the specified path (in the example, /hello)
  - the annotation **@ResponseBody** specifies that the value returned by the method is to be interpreted as the contents of the response
  - altrimenti, in Spring Web MVC, the value returned by a controller method is interpreted as the name of the view to be displayed after the method is finished executing (discussed later)
  - for this, a GET request http://localhost:8080/hello remainstuisce the string Hello, world!

- The Spring Web MVC framework

The *Spring Web MVC* framework defines the structure and programming model of web applications with Spring – based on the MVC (Model-View-Controller) pattern and composed of

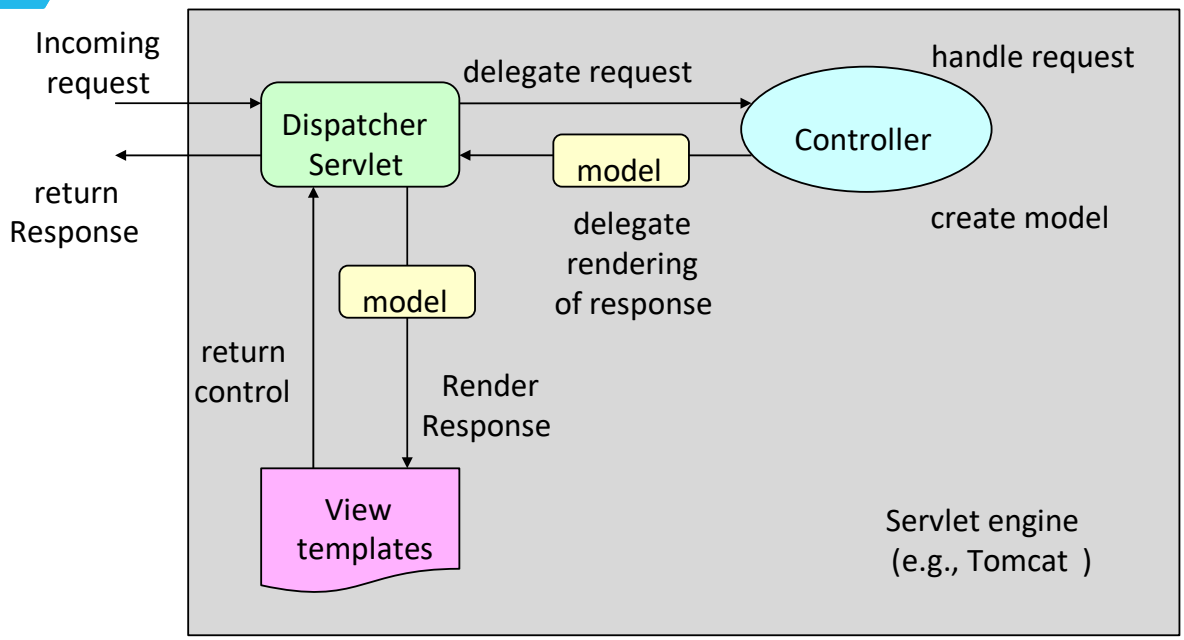
- *controller* objects – responsible for processing users' web requests
- the *model* – responsible for managing the information of interest of the application
- *views* – responsible for displaying responses and model information to users

The Spring Web MVC framework

- In the Spring Web MVC framework
  - the *model* is a simple Map<String, Object> map from attributes to values – or you can use a Model object with addAttribute and getAttribute operations
  - each *controller* usually performs these tasks
  - receives a request with its parameters
  - process the request and populate the model
  - returns the name of the view to be used to render the response
  - each *view* is a templatand web pages
  - views can be implemented with different technologies (such as JSP and Thymeleaf)
  - rendering a view is usually based on replacing template elements with template attribute values

Request management

Here's a high-level description of how to handle a request



- Another example

- Let's now show another small example, based on Spring Web MVC, with Thymeleaf views
  - in this case the spring-bootstarter-thymeleaf dependency should also be used – see the Spring Boot documentation for a list of possible starter dependencies

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

  - we want to create an application to handle type requests */hello/ name*, which returns a custom greeting
  - we use a controller for /hello/*name,*a template with only one name attribute and one greeting view

Controller for the Hellor

Here is the controller to handle requests of type /hello/*name*

```
package asw.springboot.web.hello;

import java.util.Map; import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping; import
org.springframework.web.bind.annotation.PathVariable;

@Controller public class
HelloController {

    @RequestMapping("/hello/{name}") public String
    hello(Map<String, Object> model,
                                @PathVariable String name) {
        model.put("name", name); return
        "greeting";
    }

}
```

- the controller receives the name to be greeted as a parameter (@PathVariable),copies it to the name attribute of the model and delegates the response to the greeting view

View for Helloapplication

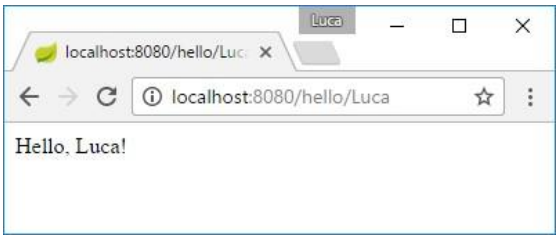
- Here is the Thymeleaf greeting view – specified by the greeting file.html in the templates folder

```
<html>
<body>
    <p>Hello, <span th:text="${name}">name goes here</span>!</p>
</body>
</html>
```

  - this view has a element associated with the name attribute of the template
  - the rendering of this view is based on replacing the fictitious content of this element (in the example, "*name goes here*") with the content of the name attribute in the modello (by virtue of the th:text="${name}" attribute of the span element)

The Hello application

A get request http://localhost:8080/hello/Luca




```
<html>
<body>
    <p>Hello, <span>Luca</span>!</p>
</body>
</html>
```

Automatic configuration

- Usually, a web application based on the Spring Web framework MVC requires complex configuration
  - in terms of Java configuration – e.g., a class to initialize the web application, which creates the servlet dispatcher
  - or XML configuration – e.g., a web file.xml
- Conversely, with Spring Boot, setting up a web application is much easier – in the example, you don't need any other files besides those already shown
  - Spring Boot then takes care of identifying, creating and configuring (automatically and intelligently) all the components /beans necessary for the application
  - in particular, it automatically creates all MVC beans (DispatcherServlet, HandlerMapping, Adapter, ViewResolver) – in this case the ViewResolver is a ThymeleafViewResolver (automatically confined, by virtue of the starter dependency)

Model and controller scopes

Element Scope Observations in Spring Web MVC

- the model has scope *request* – the dispatcher servlet prepares, for each request, a new model for the selected controller
- therefore, typically, model attributes are not shared between different requests – however, you can specify with **@SessionAttributes** that some model attributes have scope *sessions* – to manage session state as model attributes

e.g. shopping cart: something you want to have a visibility at the level of session

- the controller has *scope singleton* (by default)
- so there is a sharing of controller instance variables between all requests and sessions handled *by that controller* – but not between different controllers
- however, you canspecify the scope *session* for the controller – to manage session state as controller instance variables (if all use case operations are implemented by a single controller)

- Test of an applicationand web

- Testing a Spring Web MVC application can be based on the verification of different aspects
  - e.g., with reference to a GET /hello/Luca request
  - that there is an HTTP response OK
  - that the name of the view returned by the controller is greeting
  - that the model returned by the controller contains a name attribute
  - that the value of the name attribute in the model is Luke
  - that the contents of the returned HTTP page contain the string Hello, Luca!

An example of a test

```
package asw.springboot.web.hello; import ...;

@WebMvcTest(HelloController.class) public class
HelloApplicationMockMvcWebTests {

    mockMvc private
    @Autowired;

    @Test public void helloLucaTest() throws Exception {
    mockMvc.perform(MockMvcRequestBuilders.get("/hello/Luca"))
            .andExpect(MockMvcResultMatchers.status().isOk())
            .andExpect(MockMvcResultMatchers.view().name("greeting"))
            .andExpect(MockMvcResultMatchers.model().attributeExists("name"))
            .andExpect(MockMvcResultMatchers.model().attribute("name", "Luke"))
            .andExpect(MockMvcResultMatchers.content()
                .string(containsString("Hello, <span>Luca</span>!")));
    }

}
```


* Spring Data JPA

- *Spring Data* is a Spring project (composed in turn of other projects) to support the management of persistent objects and access to databases
  - in particular, the *Spring Data JPA* project supports the implementation of JPA-based repositories
  - in JPA, an *entity* is a type of persistent object – persistent objects must be labeled with the annotation **@Entity**
  - a *repository* is an object that provides a CRUD interface for accessing an entity in the database Spring Data provides *dynamic repositories*
  - the developer only defines the interface – and the implementation is carried out automatically by Spring Data

- different Spring Data subprojects implement different implementations, e.g., for JPA or MongoDB

Using Spring Data JPA

To use Spring Data JPA, the spring-boot-starter-data-jpa starter dependency must be used

- this dependencytransitively implies the use of Hibernate as a JPA provider
- this is also a conventional choice ("opinionated") of Spring Boot, which can be overwritten and modified
- instead a dependency for the database driver should be added separately
- e.g., the org.hsqldb:hsqldb dependency to use HSQL (an in-memory db, useful during development and testing)

```
dependencies { ... implementation 'org.springframework.boot:spring-boot-starter-data-
    jpa' implementation 'org.hsqldb:hsqldb' ...
}
```


Example: RestaurantServiceApplication

- We want to define (in the context of an **efood** application for the management of an ordering and home delivery service of meals from restaurants, on a national scale) a **restaurant-service** application service for the management of a set of restaurants
  - restaurants are defined as a JPA **Restaurant** entity
  - access to restaurants is through a repository **RestaurantRepository**
  - it also defines a **restaurantservice service**
  - the application is made (for now) as an application Spring Web MVC

- this example will in fact also be taken up in subsequent handouts, to exemplify other ways of interacting with the application and other technologies

Hexagonal service architecture

The **efood restaurant-service** application service is structured with the hexagonal architecture, using the following packages

- **asw.efood.restaurantservice** is the basic package of the service
- **asw.efood.restaurantservice.domain** defines the interior (business logic) of the service, including its ports
- accountsand between entities, services and repositories (interfaces) – services and repositories are ports packages for adapters
- **asw.efood.restaurantservice.web** defines the web adapter used by users to access the service – contains web controllers and presentation models
- you do not need to define the adapter for jpa by using dynamic repositories
- other adapters for this service will be defined later, each in its own package

The Restaurant Entity

```
package asw.efood.restaurantservice.domain; import
javax.persistence.*;
@Entity
public class Restaurant {
    Private @Id
    @GeneratedValue
    Long id;
```

private String name; private String location;

... constructors and methods get, set and toString ...

}

Repository for Restaurant

In Spring, a repository for an entity is defined as an interface that extends **CrudRepository<Entity,Id>**

- this interface deends methods such as **save, delete** and **findById** – to which you can add other methods (using a conventional naming scheme)

package asw.efood.restaurantservice.domain;

import org.springframework.data.repository.CrudRepository; import java.util.*;

public interface RestaurantRepository extends CrudRepository<Restaurant, Long> {

public Restaurant findByName(String name); public Collection<Restaurant> findAll(); public Collection<Restaurant> findAllByLocation(String location);

}

- the implementation of this interface is provided by Spring Data as a dynamic repository

A service to manage restaurants

- It is generally useful to define a "service" class to expose the functionality of the application – e.g., to controllers
  - services are a type of components, annotatthe **@Service** package

asw.efood.restaurantservice.domain;

import org.springframework.stereotype.Service; import org.springframework.transaction.annotation.Transactional; import org.springframework.beans.factory.annotation.Autowired; import java.util.*;

```
@Service @Transactional public class
RestaurantService {

    @Autowired private RestaurantRepository
    restaurantRepository;

    ... service operations ...

}
```


A service to manage restaurants

Some operations of the service for the management of restaurants

```
public Restaurant createRestaurant(String name, String location) { Restaurant
    restaurant = new Restaurant(name, location); restaurant =
    restaurantRepository.save(restaurant); return restaurant;
}

public Restaurant getRestaurant(Long id) {
    Restaurant restaurant = restaurantRepository.findById(id).orElse(null);
    return restaurant;
}

public Collection<Restaurant> getAllRestaurants() {
    Collection<Restaurant> restaurants = restaurantRepository.findAll();
    return restaurants;
}
```


Some web operations

- Let's now examine the definition of some simple controller operations

```
package asw.efood.restaurantservice.web; import

asw.efood.restaurantservice.domain.*;
```

```
import org.springframework.stereotype.Controller; import
org.springframework.beans.factory.annotation.Autowired; import
org.springframework.web.bind.annotation.*; import
org.springframework.ui.Model; import java.util.*;

@Controller
@RequestMapping(path="/web")  public class
RestaurantWebController {

    @Autowired
    private RestaurantService restaurantService;

    ...

}
```

S

Search for a restaurant

```
/* Find the restaurant with restaurantId. */
@GetMapping("/restaurants/{restaurantId}") public
String getRestaurant(Model model,
                                    @PathVariable Long restaurantId) { Restaurant
    restaurant = restaurantService.getRestaurant(restaurantId);
    model.addAttribute("restaurant", restaurant); return "get-
    restaurant";
}
```

- **@GetMapping** and **@PostMapping** annotations can be used instead of **@RequestMapping(path="...", method=RequestMethod.GET)** and **@RequestMapping(path="...", method=RequestMethod.POST)**

Some web operations

- Here is the view Thymeleaf get-restaurant

```
<html>
<body><h1><span th:text="${restaurant.name}">name goes
here</span></h1>
<p>
    Restaurant <span th:text="${restaurant.name}">name</span> (id=<span
    th:text="${restaurant.id}">id</span>) is located in <span
    th:text="${restaurant.location}">location</span>.
</p>
</body>
</html>
```

  - note the use of *object*notation.*field* to access a field of an object passed as an attribute of the template

S

- in practice these expressions are evaluated by invoking the getfield() methods of the object

List of all restaurants

```
/* Find all restaurants. */ @GetMapping("/restaurants")
public String getRestaurants(Model model) {
    List<Restaurant> restaurants = restaurantService.getAllRestaurants();
    model.addAttribute("restaurants", restaurants); return "get-
    restaurants";
}
```


Some web operations

- Here (part of) the view Thymeleaf get-restaurants

```
<table>
    <thead>
        <tr>
            <th>Name</th>
            <th>Location</th>
            <th>Id</th>
        </tr>
    </thead>
    <tbody>
        <tr th:each="restaurant : ${restaurants}">
            <td th:text="${restaurant.name}">name</td>
            <td th:text="${restaurant.location}">location</td>
```

S

```
                <td><a th:href="@{'/web/restaurants/'+${restaurant.id}}">
                        <span th:text="${restaurant.id}">id</span></a>
                </td>
            </tr>
        </tbody>
    </table>
```

- note the iteration th:each="restaurant : ${restaurants}"

Adding a new restaurant

- the addition of a restaurant requires the use of a form for entering the data of the restaurant (name and city)
- the presentation model pattern suggests that you define a class that represents the data on the form

```
package asw.efood.restaurantservice.web; public class
AddRestaurantForm {
    private String name; private
    String location;
    ... constructor and methods get and set ...
}
```


Some web operations

- Adding a new restaurant
  - in the controller an operation must be defined to access the form for entering the restaurant data

S

- the operation must pass to the form an object of the class that represents the data of the form – this could also containdata to be displayed in the form

```
/* Create a new restaurant (form). */
@GetMapping(value="/restaurants", params={"add"}) public String
getAddRestaurantForm(Model model) {
model.addAttribute("form", new AddRestaurantForm()); return
"add-restaurant-form";
}
```

- link to the page to add a new restaurant

```
<a href="/web/restaurants?add">Add a restaurant</a>
```


The form in the Thymeleaf add-restaurant-form view

```
<form th:action="@{/web/restaurants}" method="POST" th:object="${form}">
    <div>
        <label>Name</label>
        <input type="text" th:field="*{name}" required>
    </div>
    <div>
        <label>Location</label>
        <input type="text" th:field="*{location}" required>
    </div>
    <button type="submit">Add restaurant</button>
</form>
```

Some web operations

- Adding a new restaurant
  - the controller must then define an operation to manage the receipt of the form with the restaurant data

```
/* Create a new restaurant. */ @PostMapping("/restaurants")
public String addRestaurant(Model model,
            @ModelAttribute("form") AddRestaurantForm form) {
    Restaurant restaurant = restaurantService
                .createRestaurant(form.getName(), form.getLocation());
    model.addAttribute("restaurant", restaurant); return "get-restaurant";
}
```

  - the operation receives as a parameter a new form object which contains the data entered by the user

Configuration

Also in this case no further configuration is required in addition to what is shown

- Spring Data provides dynamic implementation of the specified repositories – it also handles data source and processingaccording to a predefined configuration
- it is also possible to explicitly provide all the configuration information for access to the database – which is usually necessary for access to a "real" database, to be used in production

* Spring Boot Actuator

Spring Boot Actuator provides a set of additional features for monitoring and managing applications – to be used even when an application is in production

- provides monitoring (e.g., traces and statistics of requests received) and management mechanisms (e.g., to remotely stop an application)
- it also provides inspection mechanisms, to analyze the configuration of running beans – e.g., to inspect automatic configurations made by Spring Boot
- all in the form of RESOURCES and REST operations
- the spring-boot-starter-actuator dependency should be added

```
dependencies { compile('org.springframework.boot:spring-boot-starter-web')
    compile('org.springframework.boot:spring-boot-starter-actuator') ...
```

}

Actuator and endpoints

- Here are some default endpoints provided by Spring Boot Actuator (are REST resources)
  - /actuator – provides a list of available endpoints
  - /actuator/info – application information (customizable)
  - /actuator/health – application health (and metrics)
  - /actuator/beans – list of beans and their relationships
  - /actuator/conditions – report on the automatic configuration of the application
  - /actuator/mappings – lists the application PATH URIs and the controllers they are associated with
  - /actuator/metrics – metrics on application usage and resource consumption
  - /actuator/httptrace – track of the last (100) HTTP requests

Actuator e endpoint

Additional Spring Boot Actuator Endpoint Considerations

- endpoints can be enabled or disabled – they can also be exposed via HTTP or JMX (or even un exposed)
- by default, mostendpoints are enabled, but only a few are also exposed – to be precise, /actuator, /actuator/info and /actuator/health
- the endpoint configuration can be specified in the application.properties file – for example

# in addition to info and health, also expose the endpoint beans
management.endpoints.web.exposure.include=info,health,beans

- you can also customize the default endpoints from the spring boot actuator and add new endpoints

* Configuration via properties and profiles

- Many applications require complex configurations
  - here we deal with the configuration of an application based on external properties and configuration profiles
  - a *property* is an attribute (that is, a name) that has a value associated with it
  - the properties are useful, for example, to configure the credentials for access to the database used by the application
  - configuration *profiles* allow you to define multiple configurations for the same application for use in different execution scenarios
  - we do not, however, deal with the use of configurations to explicitly overwrite automatic (implicit) configurations of Spring Boot – which is always possible

A simple example

Consider a simple web application (REST) to give you a lucky word – this is its controller

```
package asw.springboot.luckyword;

import org.springframework.web.bind.annotation.*; import
org.springframework.beans.factory.annotation.Value;

@RestController public class
LuckyWordController {
```

```
@Value("${lucky.word}") private
String luckyWord;

@GetMapping("/lucky-word") public String
luckyWord() { return "The lucky word is: " +
luckyWord;
}

}
```

- intuitively, **@RestController** is a type of **@Controller** that implies **@ResponseBody** for its operations – therefore **luckyWord()** returns a string and not the name of a view

- External properties

- To complete the application you need to specify the value of the lucky.word property – there are several ways in Spring Boot to specify a property, including
  - using command line arguments
  - e.g., java -jar lucky-word.jar --lucky.word=Happy
  - using JVM system properties
  - java -jar -Dlucky.word=Happy lucky-word.jar
  - using operating system environment variables
  - LUCKY_WORD="Happy" java -jar lucky-word.jar
  - with Gradle: LUCKY_WORD="Happy" gradle bootRun
  - by using an application.properties or application.yml property file external to the application or stored with the application (discussed later)
  - using a property source specified with @PropertySource

External properties

SpringApplication loads properties from an application.properties property file or from an application.yml YAML file

- example of application.properties file

```
# application.properties lucky.word=Happy
```

- example of application.yml file – YAML is amarkup linguaggio, which uses a hierarchical structuring based on the indentation of names

```
# application.yml
lucky: word: Happy
```


.properties and .yml formats

These additional examples show the differences in the syntaxes of application.properties and application.yml

- example of application.properties file

```
# application.properties
spring.datasource.url=jdbc:postgresql://localhost:5432/restaurantDB
spring.datasource.username=dbuser spring.datasource.password=dbpass
spring.jpa.database-platform=org.hibernate.dialect.PostgreSQLDialect
```

- example of application.yml file

```
# application.yml spring:
  datasource:
    url: jdbc:postgresql://localhost:5432/restaurantDB username: dbuser
  password: dbpass jpa: database-platform:
  org.hibernate.dialect.PostgreSQLDialect
```


- Configuration profiles

Some applications must be able to be released in different execution environments (e.g., test and production) – and in different cases with a slightly different configuration

- this scenario can be managed by using configuration *profiles*
- an application can have multiple profiles
- the application.properties or application.yml files can specify properties that are common to all profiles
- for each *profile* profile of the application you can define a file of additional properties application-*profile*.properties or application-*profile*.yml
- the yaml format also allows you to specify multiple profiles in a single file
- the active profile of an application is specified by the spring.profiles.active property (discussed later)

Configuration profiles

An example of a multi-profile configuration file with YAML

- profiles are separated by --- and their names specified by the spring.config.activate.on-profile property

```
# application.yml

---
# this is the default profile: lucky: word:
Default

--spring: config.activate.on-profile: english
lucky: word: Happy

--spring: config.activate.on-profile: italian
lucky: word: Hurray
```


Profile selection

The active profile of an application is specified by the spring.profiles.active property – there are several ways to select the active profile of an application

- using command line arguments

- e.g., java -jar lucky-word.jar --spring.profiles.active=italiano
- using JVM system properties
- java -jar -Dspring.profiles.active=english lucky-word.jar
- using operating system environment variables
- SPRING_PROFILES_ACTIVE=english java -jar lucky-word.jar
- or SPRING_PROFILES_ACTIVE=english gradle BootRun

* Discussion

- Spring Boot is intended to simplify the development and execution of Spring applications
  - minimal code and configurations
  - starter dependencies
  - automatic application configuration
  - an opinionated approach
  - Spring Web MVC – web applications (JPS, Thymeleaf and more)
  - applications (web) can be run as JAR or WAR, embedded or in its own application server
  - Spring Data JPA – dynamic repositories
  - Spring Boot Actuator – remote application management and monitoring
  - configurations based on properties and profiles
  - we will see further characteristicsthat in subsequent handouts