Spring Boot Concepts

We will be covering in this slide deck:

- Model View Controller (MVC) philosophy
- Spring Boot Controllers
- Passing data to and from Thymeleaf
- Inversion of Control and @Configuration
- Entities and the JPA
- Input Validation

Model View Controller Philosophy

Code is much easier to manage both mentally and logistically if you can compartmentalize and separate the code into logical sections.

When graphical user interfaces were created (at Xerox Parc in the late 70s) they involved a lot of code.

Logically it would be helpful to find a natural division point to divide the code into three different domains of responsibility.

Model View Controller Philosophy

- Model – Code or storing and working with the data model of the application. State variables of the application that do not directly have to do with view and control logic would go in here. You ultimately usually make a user interface to manipulate and view a model.
- View – Code for formatting, and rendering data to the screen.
- Controller – Code responsible for handling and responding to user input, managing on-screen controls, etc. Usually the Controller code also drives the initialization and usage of the Model and View.

Model View Controller & Web

MVC has traditionally been used as a design paradigm for traditional graphical user interface applications. However it has been adopted for the ***web***:

- Model – Each web interaction is largely stateless. That typically means that the vast majority of model data is stored in some form of permanent, persistent storage (like a database), or for some cases server memory tied to a session somehow.
- View – To view something on the Web is to make a web page. View code would be associated with the generation and population of web pages.
- Controller – To the backend, the only ways a the web browser interacts with the server is via web requests. Web requests specify urls, and may pass parameters. Controller code is responsible for processing these web requests.

MVC & Spring Boot

For Spring Boot specifically:

- Model – The model is usually largely kept in classes tagged with @Entity, which are persisted using the Java Persistence Api (JPA). Spring Boot also allows you to have objects that are not persisted across server restarts that sometimes hold transient model state.
- View – To view something on the Web is to make a web page. To that end, there is a special language we use in our Spring Boot environment called Thymeleaf. It is a so called HTML templating language. It renders web pages, modifying them based on data is passed via a “model” variable.
- Controller – Processes all browser to back-end communications. A typical controller call is dispatched based on what URL is requested, then the code interacts with the model to update any changes based on data passed in the request, and then finally chooses a view template to display.

Web Requests

A Web Request can come in multiple forms from the client (browser). Controllers can choose to behave differently depending on the type of request.

The two most common type of requests are:

- GET Request – Usually used to GET information. When variables are passed to a get request they are always passed in the URL. IE http://someurl.com?vara=1@varb=foo@varc=yes

  This passes in the three variables vara=1, varb=foo, varc=yes

  General philosophically GET Requests variables **should** not be used to modify the model. Visiting a GET request generally shouldn't change anything permanently. Instead the variable might change aspects of the view.

- POST Request – Used to write information. Variables are sent in the web request and do not become part of the URL. Post requests are typically used when the user is submitting data that will potentially change model information.

  Link to Documentation About Request Methods

Web Request Parameters

The variables in web requests are called “request parameters”, and they can come from:

- User modification of the URL of a GET Request
- Form data. the <form> tag has a “method” property which

  can only be set to “get” or “post”. Other request types are not supported in HTML for complicated reasons. When the form is submitted, all the variables defined in the form get sent via the request.
- API calls. This could be Javascript from other web pages calling you, or it could even be a standalone non-web application.

Web Request Headers

A Web request has “request parameters”, but it also passes other state you can access, mainly a set of “request headers”.

Request headers hold information from the web browser (and the web server) to manipulate request processing, and inform the processor of request of data about the request.

Some examples of important header info:

- Cookies – The only way you can store information from your site in the users browser, which is usually necessary to associate the incoming request with an existing session. Spring Boot automatically does session association which is nice, so generally you don’t need to play with cookies
- Referrer – The address of the previous web page from which a link to the currently requested page was followed.
- User-Agent – Identifies the browser brand, version, and operating system.

Link to Documentation about Headers

Controllers

Any class marked with an “@Controller” annotation anywhere in your Java src tree is going to act as a Spring Controller.

In it there will be methods to process web requests. These methods are sometimes called endpoints and/or mappings.

Each of the endpoint methods for processing a web request will be marked with an annotation that ends in “Mapping”. The most common annotations are:

- `@RequestMapping` – Can be used for any type of mapping
- `@GetMapping` – This is really just a shortcut for `@RequestMapping(method = RequestMethod.GET)`
- `@PostMapping` – This is really just a shortcut for `@RequestMapping(method = RequestMethod.POST)`

Docs for @RequestMapping -- documents parameters

Controller Endpoints/Mappings

Pseudo-code Anatomy of a controller mapping:

```
@SomeMapping(value=“/my/url”, ...)
OutputType anyMethodNameIWant( inputs...)
{
    //Any code I want can go here.
    return output;
}
```

That’s a very vague format... but it is vague because it there are so many options. Let’s break it down

Controller Mapping Annotation Urls

Mapping Annotation Parameters:

All mapping annotations take a named “value” parameter which specifies the url of the endpoint. If you just pass in one unnamed parameter, it will be this value.

This url can contain a wildcard sections that are wrapped in braces “{}”, IE:

```
@RequestMapping(value=”/product/{id}”)
```

The data from these sections of the URL can then be used as inputs to the controller, and they are called “path variables”. Do not confuse them with “request parameters”, however. Path variables are from the file path of the url. Request parameters **can** come from the url too, but only for GET requests after a “?”.

Controller Mapping Method Return Value

A Mapping method returns a value.

If the value is a String it can be interpreted multiple different ways:

- If a Thymeleaf dependency is installed, it can be the path of a template file relative to the templates folder. The .html file suffix is omitted.
- A whole set of installed and default handlers look at the string to see if it matches a format they understand. IE, anything that starts with “redirect:” causes a web redirect.
- If the controller is a @RestController or the @ResponseBody annotation is used, the return value will become the actual content of the web page. If it is not a string, a JSON representation will be returned.

Link -- But it doesn't have to be a string return value

Controller Mapping Method Inputs

Mapping method input parameters have many types:

- Model – you can use a variable of this type to pass attributes to Thymeleaf.
- a Class variable whose *name* matches a a Thymeleaf th:object=”${*name*}” in a form. That object’s fields will be initialized as per the th:field=”*{fieldname}” declarations for various inputs.
- A variable annotated with @PathVariable. The *name* of the variable should match a part of the mapping url that is enclosed in braces: IE /myurl/{*name*}
- A variable annotated with @RequestParam. The *name* of the variable should match the name of the request param.
- BindingResult – This gives you the result of the validation of the previous argument. It must go right after the argument to be validated. You can also update the errors of a BindingResult object
- HttpServletRequest – you can access information about the http request from this object.
- Anything that the system knows how to build via @Autowired (via Inversion of Control)

Link to many other possibilities!

Passing Data to Thymeleaf via Model

The controller can use a Model variable to pass Thymeleaf values:

```
@GetMapping(value=“/my/url”, ...)
String controllerMethodName( Model model )
{
    //Any code I want can go here.
    model.addAttribute(“nameForThymeleaf”, 32);
    return output;
}
```

Thymeleaf can now display it inside the html page:

```
<div th:text=${nameForThymeleaf}><div>
```

Thymeleaf turns this into:

```
<div>32<div>
```

Passing Data to Thymeleaf via Model

Another way attributes can be passed to Thymeleaf is via @ModelAttribute annotation inside a Controller:

```
@ModelAttribute(“nameForThymeLeaf”)
public String methodName()
{
    return “John”;
}
```

The above is basically the same as magically putting:

```
model.addAttribute(“nameForThymeleaf”, “John”);
```

at the start of every mapping method in the Controller java file that contains the @ModelAttribute annotation.

You can override this value by setting the same attribute name inside a mapping method using the technique from the previous slide.

Thymeleaf Commands

Here is a list of some Thymeleaf commands:

- `<tag th:text=”${hello}”></tag>` puts the contents of hello variable inside the tag...IE `<tag>Hello world!</tag>` if hello = “Hello world!”
- `<tag th:each=”book : ${books}”>...</tag>` repeats the tag and all its content once for each value in ${books}. Each time through an attribute “book” will be set to a different book.
- `<tag th:if=”${value} == 0”></tag>` The tag will only appear in the document if value == 0. You can use any expression supported by Thymeleaf.
- `<tag th:unless=”${value} == 0”></tag>` The tag will appear in the document *unless* value == 0. Inverted logic from th:if
- `<tag th:attr=”class=${style_classes}”></tag>` Would become <tag class=”selected emphasized”></tag> if ${style_classes}=”selected emphasized”
- There are a vast slew of th:??? commands that just replace specific attributes/properties used a lot in html. IE: th:action, th:href, th:id, …

  See https://www.thymeleaf.org/documentation.html

Data Coming Back From Web Page

There are multiple ways we can send data params back in a request.

- If the data param selects a unique record that we want a web page for it, we might consider storing it in the url as a Path Variable: IE

```
<a th:href=”@{/users/${user_id}}>User page</a>
```

- We might pass it as a Request Parameter. This is particularly useful to control how data is displayed.

```
<a th:href=”@{/tweets?filter=${filter_mode}}”>View
tweets</a>
```

- But usually data will come in from HTML form fields. You must declare an object that you wish to store your form data in. On the next slide we use “User”.

Data Coming Back From Web Page

```
@GetMapping("/register")
public String mappingMethodName(User user_variable)
{
     ....
```

The data coming back from the web page will be mapped to whatever variable type the matching parameter is. In the case of the last slide, that will be a String.

```
<form th:action="@{/signup}" th:object="${user_variable}" method="post">
  <input type="text" th:field="*{firstName}"/>
  <input type="text" th:field="*{lastName}"/>
  <input type="text" th:field="*{userName}"/>
  <input type="text" th:field="*{email}"/>
  <input type="text" th:field="*{password}"/>
  <button name="Submit" value="Submit" type="Submit">Submit</button>
</form>
```

th:object specifies the object to be stuffed. each th:field entry specifies a field. Note that "*{fieldName}" notation is used, rather that "${variableName}". Each fieldName must be the name of a field member of the object.

POJO (Plain Old Java Object)

Generally in normal Java code you have to instantiate a lot of objects, and then you are responsible for configuring and managing those objects. IE:

```
public class UserService
{
      private DatabaseAccess userdb;
      private DatabaseAccess roledb;
      private PasswordEncoder pwEncoder;
      public UserService()
      {
            userdb = new DatabaseAccess(“user”);
            roldedb = new DatabaseAccess(“role”);
            pwEncoder = new PasswordEncoder();
       }
       ....
```

Then when UserService needs to be used, it needs to **also** be instantiated like:

```
UserService userService = new UserService();
```

Beans

However in Spring Boot, you can choose instead to have objects that are managed and instantiated by a special container called the inversion of control container. These are called beans.

```
public class UserService
{
	@Autowired
	private UserRepository userRepository;
	@Autowired
	private RoleRepository roleRepository;
	@Autowired
	private PasswordEncoder pwEncoder;
	....

}
```

Beans aren’t instantiated with new usually. They are automatically created as needed based on type compatibility to fill autowires and methods with flexible parameters like Controller Mappings. There some ways that you can automatically make your class or interface a Bean, but you can also explicitly control how beans are created and configured.

Beans

Another way to accomplish the autowiring of last slide:

```
public class UserService
{
       private UserRepository userRepository;
       private RoleRepository roleRepository;
       private PasswordEncoder pwEncoder;

       @Autowired
       public UserService(UserRepository userRepository,
                          RoleRepository roleRepository,
                          PasswordEncoder pwEncoder)
        {
              this.userRepository=userRepository;
              this.roleRepository=roleRepository;
              this.pwEncoder=pwEncoder;
        }
        ....
```

The constructor parameters are automatically generated in this case.

Inversion of Control

Beans are part of a design philosophy called Inversion of Control. The idea is instead of having the code inside classes themselves be in control of instantiating and configuring objects, they can just ask for a type of object, and something that works will be provided.

This essentially makes it much easier to radically change the behavior and configuration of an object type across the entire system, without having to change its initialization and configuration everywhere.

Let’s see how it works by defining our own Bean.

@Configuration

We define how beans are created and configured inside classes marked with @Configuration annotation.

```
@Configuration
public class BeanDefiner
{
    @Bean
    public Fruit fruit()
    {
        return new Apple();
    }
    ....
}
```

Assuming that there is a Fruit.java and there is an Apple.java that sub-classes it, this code just told Spring Boot that every time it needs to autowire or spontaneously create a Fruit... this code will be run to create an Apple. The name of the method “fruit” isn’t that important, but the name of the bean in an internal registry does become the name of the method.

Entities and the JPA

The JPA is the Java Persistence API. Using the JPA we can mark a class as persistently stored via the @Entity tag

```
@Entity
public class MyPersistentClass
{
        //All fields inside the class by default become database columns
        String fieldA;  //This would stored in a column named “fieldA”

         //A no argument constructor is required for the system to work,
         public MyPersistentClass() {
               ....
         }
         ....
}
```

Entities and the JPA

The JPA is the Java Persistence API. Using the JPA we can mark a class as persistently stored via the @Entity tag

```
@Entity
public class MyPersistentClass
{
        //All fields inside the class by default become database columns
        String fieldA;  //This would be stored in a column named “fieldA”

         //A no argument constructor is required for the system to work,
         public MyPersistentClass() {
               ....
         }
         ....
}
```

Entities and the JPA

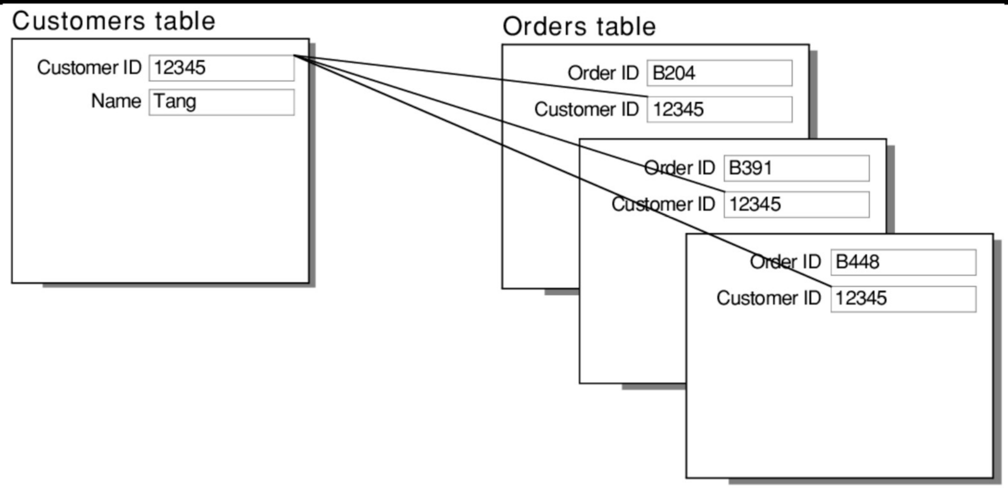
By definition, Entities always have an identity, but if you actually want to have an id property inside an entity object, you have to define one. You will need one in order to reference the object in most cases. To mark something an Id use the @Id tag. It is usually a good idea to make the Id a Long type. Also almost 100% of the time you'd want to also want to make it a auto-generated value using the @GeneratedValue annotation below:

```
@Entity
public class MyPersistentClass
{
        @Id
        @GeneratedValue(strategy = GenerationType.AUTO)
        private Long  id;
         ....
}
```

Useful JPA Annotations on Fields

- @Transient – If you want to mark a variable as something that should be stored persistently, then annotating it with @Transient will keep it from being stored in the database.
- @Column – This has three possible named parameters “name”, “nullable”, and “length”. The name of the column, whether column can be set to null, and what the max length of the item can be in the database can be set here.
- @CreationTimeStamp – Will set a Date/Calendar/Time/Timestamp type automatically when entity is created.

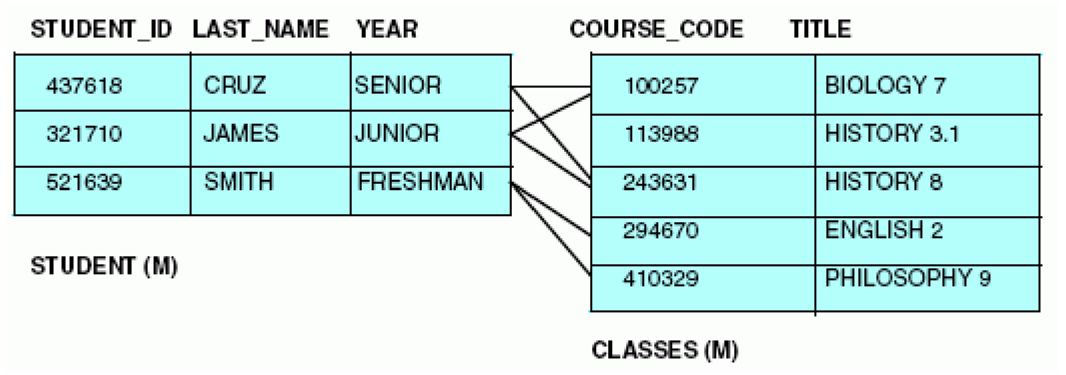
One to Many Mapping



```
@Entity
public class Customer
{
        @Id
        @GeneratedValue(strategy = GenerationType.AUTO)
        private Long id;

        String name;

        @OneToMany( cascade = CascadeType.ALL,
                        mappedBy = “orderingCustomer” )
        private List<Orders> orders; //A collection type that
}
```

```
@Entity
public class Order
{
        @Id
        @GeneratedValue(strategy = GenerationType.AUTO)
        @Column(name = “order_id”)
        private Long id;

        @ManyToOne( fetch=FetchType.LAZY)
        //This just gives the join column a name.
        @JoinColumn( name = “customer_id” )
        Customer orderingCustomer;
}
```

Many to Many Mapping

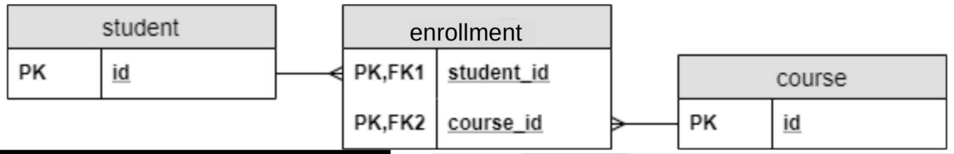


STUDENT_ID	LAST_NAME	YEAR
437618	CRUZ	SENIOR
321710	JAMES	JUNIOR
521639	SMITH	FRESHMAN

STUDENT (M)

COURSE_CODE	TITLE
100257	BIOLOGY 7
113988	HISTORY 3.1
243631	HISTORY 8
294670	ENGLISH 2
410329	PHILOSOPHY 9

CLASSES (M)



```
@Entity
public class Student
{
        @Id
        @GeneratedValue(strategy = GenerationType.AUTO)
        private Long id;

        String lastName;
        String Year;

        @ManyToMany( cascade = CascadeType.ALL)
        @JoinTable(name = “enrollment”,
            joinColumns = @JoinColumn(name = “student_id”),
            inverseJoinColumns = @JoinColumn(name = “course_id”))
        private List<Course> courses; //A collection type
}
```

```
@Entity
public class Course
{
        @Id
        @GeneratedValue(strategy = GenerationType.AUTO)
        private Long id;

        String title;

        @ManyToMany(mappedBy = “courses”)
        Set<Student> students;
}
```

Input Validation

You can annotate fields in your entities with ***constraints*** for validation.

Here are some possible annotations:

- @NotNull – item parameter must not be null.
- @Max(value=maximum) – item must be <= maximum
- @Min(value=minimum) – item must be >= minimum
- @Size(min = 10, max = 20) – item’s size must be between 10 and 20 (IE—string length)

List of Validation Constraints (javax.validation)

More validation constraints (org.hibernate.validator)

Input Validation

For any form data passed passed in via a th:object in thymeleaf, you can use validation annotations. Since form data comes with a full th:object, you usually use the @Valid constraint. The @Valid constraint all the fields of an object to see if each of THEIR constraints are met.

```
@Controller
public class RegistrationController
{
    @GetMapping(“/signup”)
    String register(@Valid User user)
    {
        ...
    }
}
```

Input Validation

If the validation fails, an “exception” is thrown, which will be converted into a Web error: 400 Bad Request.

However, if you want to handle the errors yourself, you can add a BindingRequest object as the next parameter after the validated one.

```
@Controller

public class RegistrationController

{

    @GetMapping(“/signup”)

    String register(@Valid User user, BindingResult result)

    {

          ...

    }
}
```

Input Validation

If in the previous slide’s controlelr function you call result.hasErrors() it will tell you whether there are errors.

You can also inject errors using the .rejectValue() call:

```
@Controller
public class RegistrationController
{
    @GetMapping(“/signup”)
    String register(@Valid User user, BindingResult result)
    {
        if(user.name.equals(“Quentin”))
        {
            bindingResult.rejectValue(“username,
                “error.user”,
                “The name Quentin is not allowed.”);
              ....
```

Input Validation

On the next template you load, any ‘username’ related errors will cause the following Thymeleaf expression to evaluate to true.

`th:if=”${#fields.hasErrors(‘username’)}”`

and

`th:errors=”*{username}”` can be used to display all errors for a field inside a tag.

`${#fields.errors(‘username’)}` can be used inside a Thymeleaf expression as a list of all errors associated with ‘username’. This is even walkable with th:each.

Path Variables and Request Parameters

To validate path variables and request parameters you will need to add the @Validated annotation to your @Controller.

```
@Controller
@Validated
public class Example {
    @GetMapping("/example")
    String exampleMapping(@RequestParam @Min(0) int option)
    {
        // option is validated with @Min(0). On error,
        // we get a 400 Bad Request web error
        // You cannot use a BindingResult in this case–only
        // works with form data.
        ...
    }
```