DAY 7

Thymeleaf, Spring Boot REST APIs

THYMELEAF

- Thymeleaf is a popular server-side Java template engine for web applications. It enables developers to build dynamic web pages using natural templates with plain HTML, while seamlessly integrating with Spring Framework for server-side processing.

THYMELEAF EXPRESSIONS

Thymeleaf expresssions

- ${...} : Variable expressions.
- *{...} : Selection expressions.
- #{...} : Message (i18n) expressions.
- @... : Link (URL) expressions.
- ~{...} : Fragment expressions.
- __{...} : use an element inside another

ATTRIBUTES

Thyemleaf attributes

- message: `<p th:text="#{msg.welcome}">Welcome everyone!</p>`
- list: `<li th:each="book : ${books}" th:text="${book.title}">En las Orillas del Sar</li>`
- link: `<form th:action="@{/createOrder}">`
- action: `<input type="button" th:value**="#{form.submit}" />`
- path: `<a th:href="@{/admin/users}">`

HOW TH:WORKS

How to write `th:`

HOW **TH:** WORKS

```
<td th:text="${book.title}"></td>
```

<td th:text="${book.title}">

open and close html tag

define th attribute by th:

kind of attribute, in this case **text** but it may be **list, path, if**

attribute is always a **string,** with some variables injected from **@Controller, use $ symbol to note that.**
And after symbol, write the variable name to finish the **expression with brackets {}**

CHEAT SHEET

Feature	Description	Syntax
th:text	Sets the text of an element	`<p th:text="${someValue}">Default Text</p>`
th:if	Conditionally renders an element	`<p th:if="${someCondition}">Visible when condition is true</p>`
th:each	Loops over a collection and renders an element for each item	`<ul><li th:each="item : ${items}" th:text="${item}">Default Text</li></ul>`
th:object	Binds a form to an object and sets its properties	`<form th:object="${user}"><input th:field="*{name}" /></form>`
th:action	Sets the URL for a form's submission	`<form th:action="@{/submit}" method="post">`

CHEAT SHEET

th:href	Sets the URL for an anchor tag	`<a th:href="@{/page}">Link Text</a>`
th:src	Sets the source URL for an image tag	`<img th:src="@{/image.jpg}" alt="Alt Text" />`
th:value	Sets the value of an input field	`<input th:value="${someValue}" />`
th:selected	Conditionally selects an option in a select field	`<select><option th:selected="${isSelected}">Option Text</option></select>`
th:disabled	Conditionally disables an input field	`<input th:disabled="${isDisabled}" />`
th:readonly	Conditionally sets an input field as read-only	`<input th:readonly="${isReadOnly}" />`

th:classappend	Conditionally appends a CSS class to an element	`<div class="default" th:classappend="${additionalClass}"></div>`
th:style	Sets the style attribute of an element	`<div th:style="'background-color:' + ${bgColor} + ';'"></div>`
th:attr	Sets any attribute of an element	`<input th:attr="data-id=${itemId}" />`
th:replace	Replaces an element with another	`<div th:replace="fragments/header :: header"> </div>`
th:include	Includes a fragment of a template	`<div th:include="fragments/footer :: footer"> </div>`
th:unless	Conditionally renders an element when a condition is false	`<p th:unless="${someCondition}">Visible when condition is false</p>`
th:inline	Sets the inline mode of an element	`<script th:inline="javascript">alert([[${message}]]); </script>`
th:textappend	Appends text to an element	`<span th:textappend="${additionalText}">Default Text</span>`
th:with	Sets a local variable in the current context	`<div th:with="varName=${someValue}"></div>`

REST

- Rest stands for Representational State Transfer.

- Let's understand the meaning of each word in the REST acronym.

  - State means data
  - Representational means formats (such as XML, JSON, YAML, HTML, etc)
  - Transfer means carry data between consumer and provider using the HTTP protocol

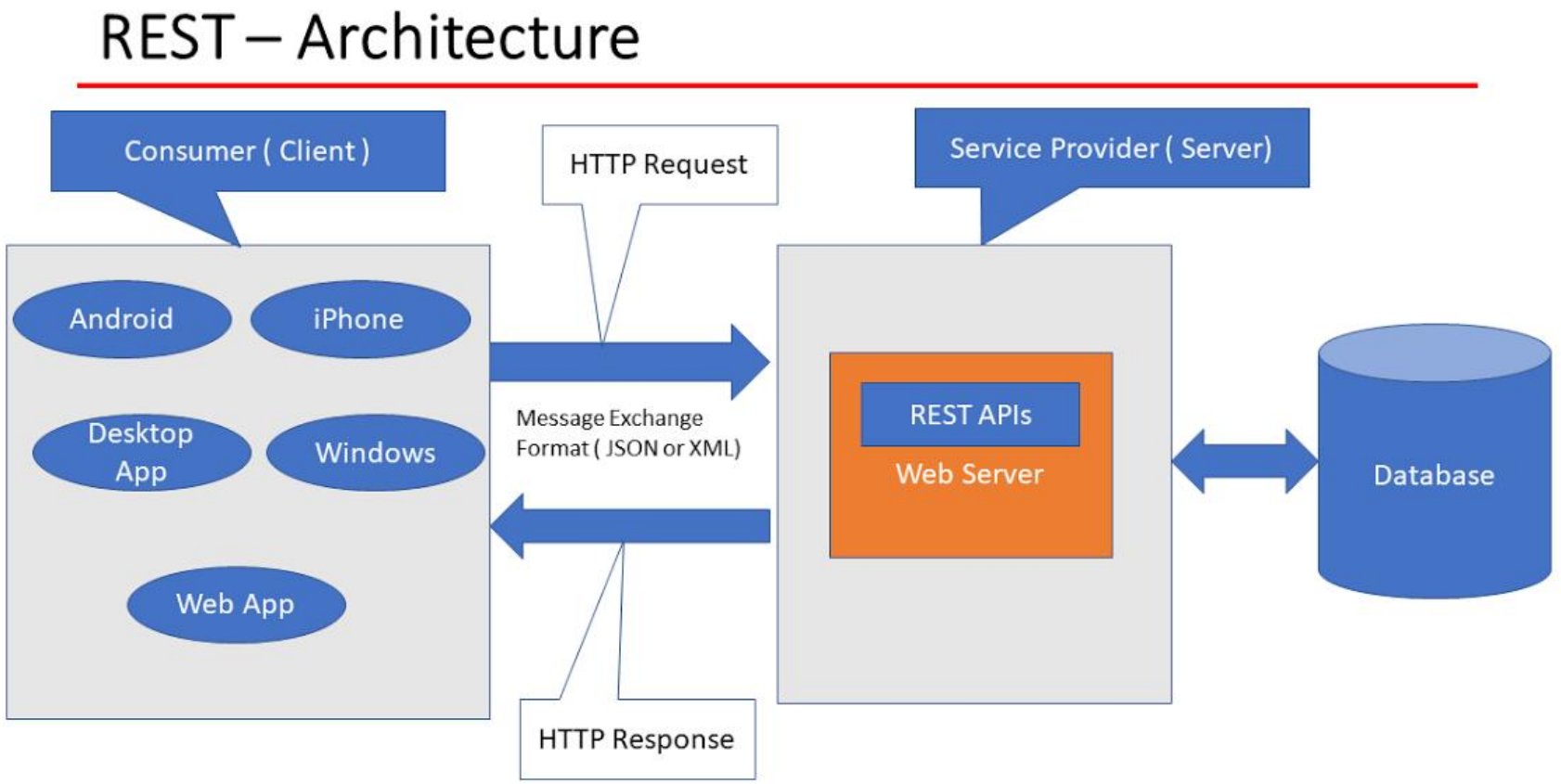
Representational State Transfer

- REST was originally coined by Roy Fielding, who was also the inventor of the HTTP protocol.
- A REST API is an intermediary Application Programming Interface that enables two applications to communicate with each other over HTTP, much like how servers communicate to browsers.
- The REST architectural style has quickly become very popular over the world for designing and architecting applications that can communicate.
- The need for REST APIs increased a lot with the drastic increase of mobile devices. It became logical to build REST APIs and let the web and mobile clients consume the API instead of developing separate applications.

Representational State Transfer

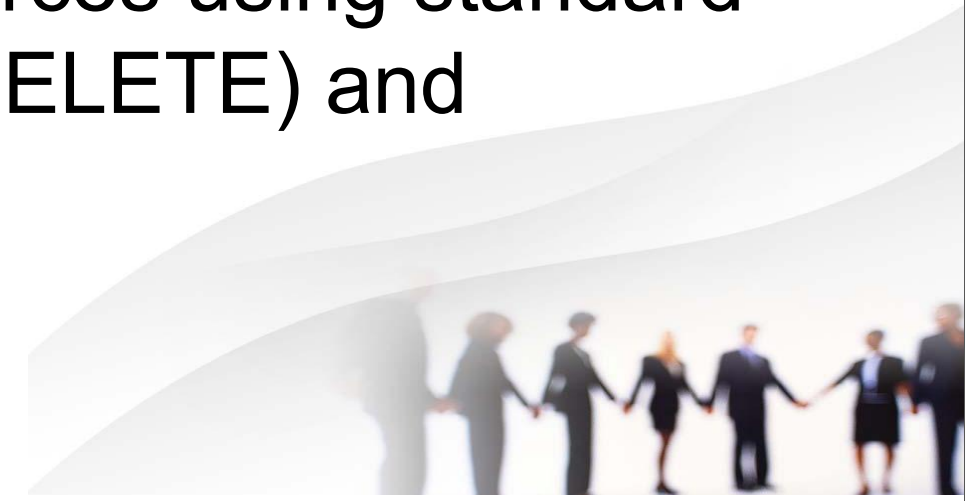
- SOAP(Simple Object Access Protocol) is another web service protocol that has been undertaken by REST
- SOAP exposes components of application logic as a service rather than data.
- SOAP only exchanges data over XML and REST produces the ability to exchange data over a variety of data formats. These include JSON and XML based on the configuration of the server
- SOAP is preferred when robust security is essential as it provides support for web services security(WS-Security), which is a specification defining how security measures are implemented in web services to protect them from external attacks.
- Another advantage of SOAP over REST is its built-in retry logic to compensate for failed requests, unlike REST in which the client has to handle the failed requests by retrying.

REST ARCH


REST – Architecture
Consumer (Client)
HTTP Request
Service Provider (Server)
Android
iPhone
Desktop App
Windows
Web App
Message Exchange Format (JSON or XML)
REST APIs
Web Server
Database
HTTP Response

SPRING BOOT REST API

- A Spring Boot REST API is a web service built using the Spring Boot framework, which adheres to the principles of Representational State Transfer (REST). RESTful APIs enable communication between client and server over HTTP by allowing clients to perform CRUD (Create, Read, Update, Delete) operations on resources using standard HTTP methods (GET, POST, PUT, DELETE) and following RESTful conventions.



COMPONENTS - SPRING BOOT REST

- Spring Boot: Spring Boot is a framework built on top of the Spring Framework, which simplifies the process of building production-ready applications in Java. It provides out-of-the-box configurations, embedded servers, and dependency management, allowing developers to focus on writing business logic rather than boilerplate code.

- RESTful Architecture: A RESTful architecture is an architectural style for designing networked applications. It emphasizes the use of standardized HTTP methods (GET, POST, PUT, DELETE) to perform operations on resources. Each resource is identified by a unique URI, and the representation of resources can be in various formats like JSON, XML, etc.

COMPONENTS - SPRING BOOT REST

- Controller Layer: In a Spring Boot REST API, controllers are responsible for handling incoming HTTP requests and returning appropriate responses. Controllers are typically annotated with @RestController or @Controller annotations, and methods within them are annotated with @GetMapping, @PostMapping, @PutMapping, @DeleteMapping, etc., to map HTTP methods to specific endpoints.

- Data Transfer Objects (DTOs): DTOs are used to represent data transferred between the client and server. They are simple POJOs (Plain Old Java Objects) containing fields that correspond to the data being exchanged. DTOs help in decoupling the internal representation of data from its external representation.

- Service Layer: The service layer contains the business logic of the application. It encapsulates the operations performed on resources and interacts with data repositories (e.g., databases) to fetch or manipulate data. Services are typically invoked by controllers to perform operations requested by clients.

COMPONENTS - SPRING BOOT REST

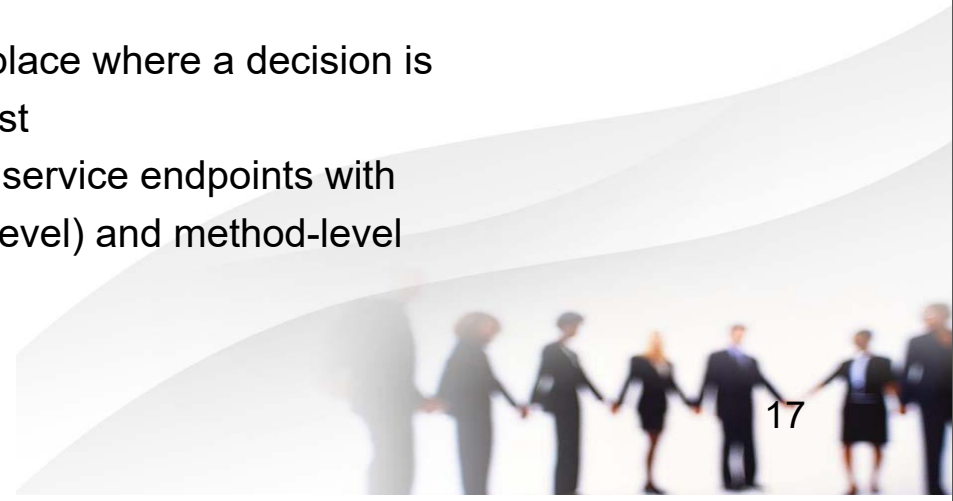
- Data Access Layer: The data access layer is responsible for interacting with the underlying data storage mechanisms, such as databases or external APIs. In Spring Boot applications, this is often achieved using Spring Data repositories or other ORM (Object-Relational Mapping) frameworks.

- Exception Handling: Exception handling is an essential aspect of building robust REST APIs. Spring Boot provides mechanisms for handling exceptions gracefully and returning appropriate error responses to clients. You can use annotations like @ExceptionHandler to handle specific exceptions and customize error responses.

@RequestMapping

- @RequestMapping is used to map URLs such as */viewCarrier.do* onto a methods in the Controller class
- Can be used both at class level and methods levels
- Typically the class-level annotation maps a specific request path (or path pattern) onto a form controller, with additional method-level annotations narrowing the primary mapping for a specific HTTP method request method ("GET", "POST", etc.) or an HTTP request parameter condition.

- Spring 3.1+
  - The ***RequestMappingHandlerMapping*** is the only place where a decision is made about which method should process the request
  - Think of controller methods as a collection of unique service endpoints with mappings for each method derived from type (class level) and method-level @RequestMapping information



Consumable Media Types

You can narrow the primary mapping by specifying a list of consumable media types. The request will be matched only if the *Content-Type* request header matches the specified media type. For example:

```
@Controller
@RequestMapping(value = "/pets", method = RequestMethod.POST, consumes="application/json")
public void addPet(@RequestBody Pet pet, Model model) {
    // implementation omitted
}
```

Note that in the above example, the request data is sent as part of the HTTP request body which is bind to the method parameter using the @RequestBody annotation

Producible Media Types

You can narrow the primary mapping by specifying a list of producible media types. The request will be matched only if the *Accept* request header matches one of these values. Furthermore, use of the *produces* condition ensures the actual content type used to generate the response respects the media types specified in the *produces* condition. For example:

```
@Controller
@RequestMapping(value = "/pets/{petId}", method = RequestMethod.GET, produces="application/json")
@ResponseBody
public Pet getPet(@PathVariable String petId, Model model) {
    // implementation omitted
}
```

Note that in the above example, the response data is sent back to the client as HTTP response (Not to a view) using the @ResponseBody annotation. Spring does the automatic conversion of the returned object to a HTTP Response to a format of JSON or XML – Spring uses Jackson for JSON and JAXB for XML

URI Template Patterns - @PathVariable

- URI Template
  - Is a URI-like string, containing one or more variable names
    http://www.example.com/users/{*userId*} – variable name is *userId*
  - When you substitute values for these variables, the template becomes a URI
    Assigning the value *arun* to the variable yields http://www.example.com/users/*arun*.

Use the **@PathVariable** annotation on a method argument to bind it to the value of a URI template variable

```
@RequestMapping(value="/owners/{ownerId}", method=RequestMethod.GET)
public String findOwner(@PathVariable String ownerId, Model model) {
    Owner owner = ownerService.findOwner(ownerId);
    model.addAttribute("owner", owner);
    return "displayOwner";
}
```

The URI template "/owners/{ownerId}" specifies the variable name ownerId. When this controller handles this request, the value of ownerId is set to the value found in the appropriate part of the URI. For example, when a request comes in for /owners/arun, the value of ownerId is arun

This is the beginning of Spring RESTful services

@RequestParam

- Use the @RequestParam annotation to bind request parameters to a method parameter in your controller

```
@RequestMapping(value = "/carrier.do", method = {RequestMethod.GET, RequestMethod.HEAD})
public ModelAndView editCarrier(
            @RequestParam(value="id", required=false) Integer carrierId,
            @RequestParam(value="action", required=false) String action
)
{
      //
}
```

- The above method can be invoked something like below:

  var actionUrl = "carrier.do?action=" + action + "&id=" + id;

  Note that the request params (action & id) are passed as part of the request URL

- Parameters using this annotation are required by default, but you can specify that a parameter is optional by setting @RequestParam's required attribute to false
- Type conversion is applied automatically if the target method parameter type is not String

@RequestBody

- The @RequestBody method parameter annotation indicates that a method parameter should be bound to the value of the HTTP request body. For example:

```
@RequestMapping(value = "/something", method = RequestMethod.PUT)
public void handle(@RequestBody String body, Writer writer) throws IOException {
    writer.write(body);
}
```

- HttpMessageConverter is responsible for converting from the HTTP request message to an object and converting from an object to the HTTP response body.
- The RequestMappingHandlerAdapter supports the @RequestBody (and @ResponseBody) annotations with the a set of default HttpMessageConverters

- The <mvc:annotation:driven /> configuration in the DispatcherServlet context configuration will enable the RequestMappingHandlerMapping and RequestMappingHandlerAdapter beans.

@ResponseBody

- This annotation can be put on a method and indicates that the return type should be written straight to the HTTP response body (and not placed in a Model, or interpreted as a view name). For example:

```
@RequestMapping(value = SID_CARRIER_LIST_URL, method={RequestMethod.GET})
public @ResponseBody List<SidCarrierInfo> getSidCodeList(
                @RequestParam(value="sidCode", required=true) String sidCode)
{
        LOG.debug("sidCode entered: " + sidCode);
        List<SidCarrierInfo> result = helperService.getSidCodeList(sidCode);
        LOG.debug("found " + result.size() + " sid/carrier objects.");
        return result;
}
```

- The above example will result in the object “SidCarrierInfo” representation (Example: JSON, XML etc..,) being written to the HTTP response stream.
- As with @RequestBody, Spring converts the returned object to a response body by using an HttpMessageConverter.

@ModelAttribute

- An @ModelAttribute on a method argument indicates the argument should be retrieved from the model.
- For the object in the model, the argument's fields should be populated from all request parameters that have matching names. This is known as data binding in

```
@RequestMapping(value = CARRIER_DETAIL_URL, method = {RequestMethod.POST})
public @ResponseBody ModelAndView editCarrier(
            @ModelAttribute(value="carrier") @Valid Carrier editCarrier,
            BindingResult bindingResult)
{
            //
}
```

- This is usually used when the form-backing (or command) object is filled up and submitted from the client form.

@Valid (JSR 303)

- In Spring, you can enable “mvc:annotation-driven” to support JSR303 bean validation via @Valid annotation on the handler method parameter, if any JSR303 validator framework is in the classpath
- Hibernate Validator is the RI for JSR303

- On the model class, add hibernate validator annotations such as @NotNull, @NotEmpty, @Range, @Min, @Max etc…,

```
public class Carrier {

	@NotNull
	@Min(0)
	@Max(9999)
	private int carrierId;

	@NotNull
	@Pattern(regexp="^([Cc]\\d{4})")
	private String carrierCode;

	@NotNull
	@Size(min=1, max=80)
	private String carrierName;

	....
}
```

0 to 9999

@Pattern (regex)
Start with “C” or “c”
Followed by exactly 4 digits

1 to 80 chars

@Valid (JSR 303)

- An @RequestBody method parameter can be annotated with @Valid, in which case it will be validated using the configured Validator instance.
- The validator was initialized using @InitBinder (from previous slides)

```
@RequestMapping(value = CARRIER_DETAIL_URL, method = {RequestMethod.POST})
public @ResponseBody ModelAndView editCarrier(
                @ModelAttribute(value="carrier") @Valid Carrier editCarrier,
                BindingResult bindingResult)
{
        if(!bindingResult.hasErrors())
        {
                // No errors
        }
        else {
                LOG.debug("Carrier has validation error:" + bindingResult.toString());
        }
        return new ModelAndView("carrier/carrierDetail", "carrier", editCarrier);
}
```

- Validator runs the validation code and based on PASS/FAIL, it sets the BindingResult
- BindingResult will now examined in the Controller as shown above for any errors

@Valid (JSR 303) – Error Messages

- There are default error messages that will be displayed for each of the JSR 303 validation annotations.
- This can be overridden in the message resources as key value pairs.
  - Key is @annotationName.object.feildName
    - objectname here refers to the command/model object name (reference) used in the JSP

```
@NotNull
@Size(min=1, max=80)
private String carrierName;

Size.carrier.carrierName=Carrier Name size must be between {1} and {2} characters

@NotNull
private Date startDate;

NotValid.carrier.startDate=Effective Date cannot occur after End Date

@NotNull
@Pattern(regexp="^([Cc]\\d{4})")
private String carrierCode;

Pattern.carrier.carrierCode=Carrier Code must begin with C followed by 4 digits
```

DIFFERENT MAPPINGS

- @GetMapping, @PostMapping, @PutMapping, @DeleteMapping: These annotations are shortcuts for @RequestMapping with method set to GET, POST, PUT, and DELETE, respectively. They are used at the method level to specify the HTTP method for handling the request.

why data transfer objects are needed in Spring Boot REST apis?

- Data Transfer Objects (DTOs) are commonly used in Spring Boot REST APIs for several reasons:

- Decoupling: DTOs help in decoupling the internal representation of data from its external representation. This separation allows for changes in one representation without affecting the other. For example, changes in the database schema or entity structure do not necessarily impact the API contract.

- Data Transformation: DTOs facilitate data transformation between different layers of the application. For instance, an entity object retrieved from the database may contain more information than what needs to be exposed to clients through the API. DTOs allow you to selectively include or exclude certain fields, thus controlling the data that is transferred over the network.