Java

Server-Side APIs

Module 2 - Week 7

Schedule

Week	Topics
Week 0	Module 2 Orientation / PostgreSQL
Week 1	Intro to databases / Ordering, limiting, and grouping
Week 2	SQL joins / Insert, update, and delete / Database design
Week 3	Data Access / Data security
Week 4	DAO testing
Week 5	**Mid-module project**
Week 6	Postman / NPM / Networking and HTTP / Consuming RESTful APIs
Week 7	Server-side APIs
Week 8	Securing APIs
Week 9	**End-of-module project**
Week 10	**Assessment**

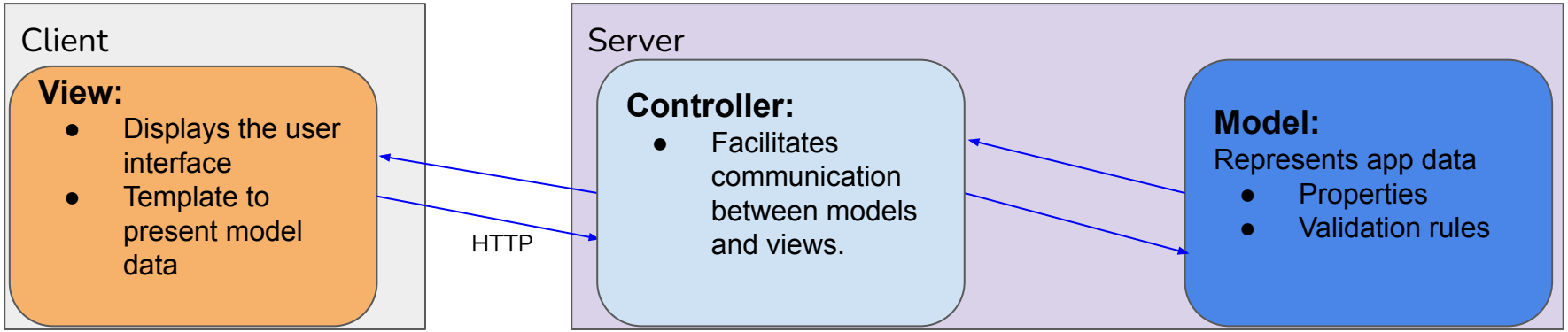
Objectives

- **Describe the MVC pattern and why programmers use it**
- Understand the high-level architectural elements of the REST architecture
    - Dependency injection
    - Model binding
- Implement a RESTful web service with full CRUD features, conforming to REST best practices

MVC

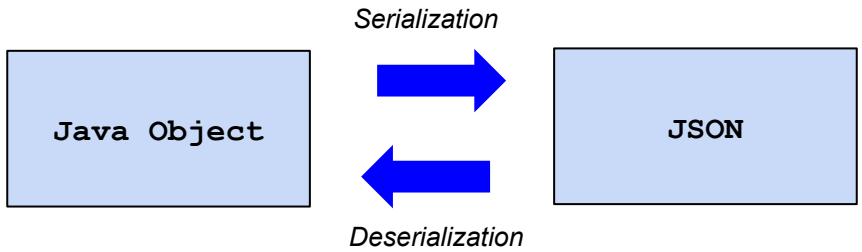
Applications can become quite large. As a result, it often becomes difficult to manage their size and complexity as new features emerge or existing requirements change. To address this, software developers rely on design patterns that assist in keeping the code clean and maintainable. One such pattern is called Model-View-Controller (MVC).

- MVC is a design pattern that separates an application into three main components:
  - Model
  - View
  - Controller
- The MVC pattern promotes loose coupling by helping to create applications so that the logic across web applications can be reused while not allowing any particular part to do too much.



Model Binding

- **Model binding** (or data binding) connects Java objects to non-object content. It has a specific focus on format and data types.
- In the **JDBC DAO** classes, we bind the Java object model properties to database table columns to store Java objects in a database by using the mapRowToObject methods or RowMapper.
- In a **REST API**, the binding is typically **between Java objects and JSON**. Within the Spring framework, the **Jackson** library handles the binding between Java objects and JSON.



REST

- REST (REpresentational State Transfer) is a software architectural pattern that defines a set of constraints and standards for how applications on a network should communicate.
- In other words, REST describes a set of rules that specifies the best practices for how communication on the web should work.

Rules of REST

1. Uniform Interface (URI, URL)
2. Stateless
3. Cacheable
4. Client-Server based
5. Layered System

RESTful

- REST APIs are based on the concept of resources and building addresses (in the form of URLs) and actions (in the form of HTTP methods) on those resources. It also uses HTTP Status Codes to alert the API users about the results of the call
- Resources are the objects defined in the application (i.e. `Hotel`, `Reservation`). You access resources through URLs, which specify the resource and, depending on which HTTP method you use, retrieve and modify them. Use nouns instead of verbs for resource endpoints.
- Name resource collections with plural nouns (e.g. hotels).
- Use nested resource endpoints to show relationships (e.g. hotels/10/reservations).

HTTP Status Codes

200	OK	Everything worked as expected—should have data returned. Common for `GET` requests.
201	Created	New resource was created. Common for `POST` requests.
204	No Content	Everything worked but no data is returned. Common for `DELETE` and `PUT` methods.
400	Bad Request	The request from the front-end had errors. Check the data passed back to see more specifics about the error. Often a data validation problem.
401	Unauthorized	The user isn't allowed to perform this action either because they aren't logged in or because their login information is wrong..
403	Forbidden	The logged in user isn't allowed to perform this action because they don't have permission.
404	Not Found	The given URL doesn't point to a valid resource.
405	Method Not Allowed	The HTTP Method given isn't valid for this URL. This could be because certain resources can't be updated or deleted and don't support the `GET`, `PUT`, `DELETE`, or `POST` methods.
500	Internal Server Error	The API itself has a problem and can't fulfill the request at this time. This could be due to a code issue or because services the API relies on, like databases or application servers, are down.

Controller

The **`@RestController`** annotation tells Spring that this will be a controller class that can accept and return data.

The **`@RequestMapping`** defines a method as being an endpoint.

Note that the same path can have different meaning if there are different HTTP methods for the same path (i.e. GET and POST map to different methods even though the path is the same.

```
@RestController
public class TodoController {

    @RequestMapping(path = "/todo", method = RequestMethod.GET)
    public List<String> getTodos() {
        return todos;
    }

    @RequestMapping(path = "/todo", method = RequestMethod.POST)
    public void addTodo(String task) {
        if (task != null) {
            todos.add(task);
        }
    }

}
```

RequestMapping

The `@RequestMapping` annotation has several arguments that define which handler method responds to a given web request.

- `@RequestMapping(path=)`
  - The `path=` parameter maps a request path to the annotated method.
- `@RequestMapping(method=)`
  - `method=` allows you to define a specific HTTP verb (method) for the request mapping (GET, POST, PUT, DELETE)
  - If not defined, ANY verb will map to the annotated method.

```
@RequestMapping(path = "/todo/{id}/{name}", method = RequestMethod.GET)
public Todo getTodo(@PathVariable Integer id,@PathVariable(required = false) String name)
{
    …
}
```

Path Variables

If you want to find a specific resource by its ID, a popular REST convention is to use the ID as a part of the path:

```
https://localhost:8080/todo/2
```

We can add a placeholder for the id (or other variable) by adding `{id}` in the path where the id value would go and then annotating a param with the same name in our method with the `@PathVariable` annotation.

```
@RequestMapping(path = "/todo/{id}", method = RequestMethod.GET)
public Todo getTodo(@PathVariable Integer id) {
    …
}
```

Request Parameters

There are times when you'll want to pass some information along to your API as part of the request. Imagine that you created a method in your Todo API where someone could retrieve all todos based on the completed status. You might use a query parameter named filter with a value of completed. It would look like this:

```
https://localhost:8080/todo?filter=completed
```

We can use the `@RequestParam` annotation to map the filter query param to a `String` param called filter in our handler method:

```
@RequestMapping("/todo")
public List<Todo> getCompletedTodos(@RequestParam String filter) {
    …
}
```

Request Parameters

We can do this with multiple query params as well:

```
https://localhost:8080/todo
```

```
https://localhost:8080/todo?filter=completed
```

```
https://localhost:8080/todo?filter=completed&limit=10
```

We can mark a @RequestParam as optional by adding the `required=false` modifier:

```
@RequestMapping("/todo")
public List<Todo> getCompletedTodos(
     @RequestParam(required = false) String filter,
     @RequestParam(required = false) Integer limit) {
```

Dependency Injection

Dependency Injection is where instances of classes that are depended on are injected into a new object rather than created by that object. This further decouples the classes from each other and allows the controller to be completely independent from any implementation of the DAO interface.

In Spring, if you want to inject one class into another, you add an annotation to make Spring aware of the class. For the DAOs, you'd add an `@Component` annotation:

```
@Component
public class MemoryHotelDao implements HotelDao {
    ...
}
```

Dependency Injection

Now, the `MemoryHotelDao` class can be injected into other classes, like the controller. To inject the DAO into the controlller, you declare a constructor parameter of the class you depend on in the controller like this:

```
private HotelDao dao;

public HotelController(HotelDao dao) {
    this.dao = dao
}
```

Now, when Spring creates the controller, an instance of the DAO class with the `@Component` annotation is also created and injected into the controller for you.

Dependency Injection

This also makes the controller easier to unit test. When creating a new controller in a unit test, you can pass in a test version of the `HotelDao`:

```
HotelController testController = new HotelController(new TestHotelDao());
```

This allows the `HotelController` to depend on any `HotelDao` without being tied to a single implementation of it.

Data Validation

It's important to handle data validation in back-end code and front-end code. Validation in Java is done using a Java standard called Bean Validations.

Bean Validations are annotations that are added to Java model classes and verify that the data in objects match a certain set of criteria. Here's an example:

```
public class Product {
    @NotBlank( message="Product name cannot be blank" )
    private String name;

    @Positive( message="Product price cannot be negative" )
    private BigDecimal price;

    @Size( min=20, message="Description cannot be less than twenty characters" )
    private String description;

    /*** Getters and Setters ***/
}
```

Data Validation

The validations that you add to your model objects aren't automatically checked for you. To check them in your Controller, you need to add the `@Valid` annotation to the model:

```
@RequestMapping(path = "", method = RequestMethod.POST)
@ResponseStatus(HttpStatus.CREATED)
public void create(@Valid @RequestBody Product product) {
    // data validation
    productsDao.add(product);
}
```

If the validation fails, meaning that the data supplied from the request doesn't pass the validation tests you defined in the model, then the response returns a status code of 400 Bad Request, and the a JSON object containing info about the validation error is returned to the client.

Data Validation

Annotation	Applies To	Description
`@NotNull`	**Any**	The variable can't be `null`.
`@NotEmpty`	`array, List, String`	The variable can't be `null`. Also, if it's a `List` or `array`, it can't be empty. If it's a `String`, it can't be an empty `String`.
`@NotBlank`	`String`	The variable can't be empty or only contain white space characters.
`@Min`	`int, long,`	The variable must have a value greater than the specified minimum. `null` values are skipped.
`@Max`	`int, long`	The variable must have a value less than the specified maximum. `null` values are skipped.
`@DecimalMin`	`BigDecimal, double`	The variable must have a value greater than the specified minimum decimal. `null` values are skipped.
`@Size`	`array, List, String`	The variable length, as an `array, List,` or `String`.

Request Body

When using the POST or PUT methods, we can bind the request body (payload) to a model that matches the payload by using the `@RequestBody` annotation. The code below would take JSON data representing a hotel object and deserialize it into a Java `Hotel` object which is declared as a param type of a param to our handler

```
@RequestMapping( path = "/hotels", method = RequestMethod.POST)
public void addHotel(@RequestBody Hotel newHotel) {
    …
}
```

Note that you may only annotate one handler param with `@RequestBody` since there isn't a way to POST multiple request bodies in a request.

PUT

```
/**
 * Updates a product based on the ID and the request body
 *
 * @param product the updated product
 * @param id the id of the product that is getting updated
 */
@RequestMapping( path = "/products/{id}", method = RequestMethod.PUT )
public void update(@RequestBody Product product, @PathVariable int id) {
    product.setId(id);

    // Update product in underlying datastore
}
```

Delete

Normally, the correct status code is returned by default, but there may also be times when you want to return a different status code than the default. For instance, by default the status code 200 OK is returned for a successful DELETE. However, REST suggests returning status code 204 No Content on a successful DELETE. This is accomplished by adding the annotation:

```
@ResponseStatus(value = HttpStatus.NO_CONTENT)
```

```
@RequestMapping( path = "/products/{id}", method = RequestMethod.DELETE )
@ResponseStatus(value = HttpStatus.NO_CONTENT)
public void delete(@PathVariable int id) {

    // Remove the product from underlying datastore

}
```

Exceptions

There may also be times when you want to return a different status code than the default due to an error. For example, if a user wanted to update the product with an ID of 13 and that ID wasn't in the database, you'd want to return a 404 Not Found status code.

You can do that by creating an Exception class that's linked to that status code with a `@ResponseStatus` annotation. Then you can throw a new instance of this exception from your controller method when the invalid product ID is detected.

```
@ResponseStatus(value = HttpStatus.NOT_FOUND)
public class ProductNotFoundException extends Exception {
    …
}
```