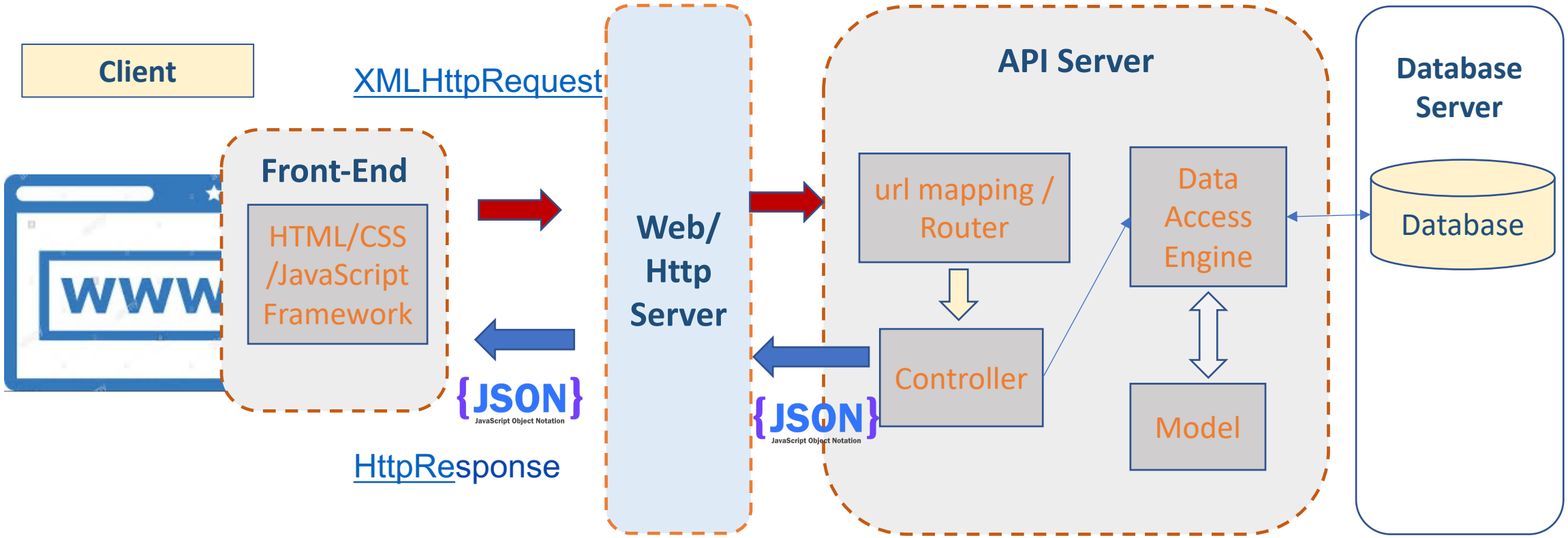
What is RESTful API ?

Learning by Doing

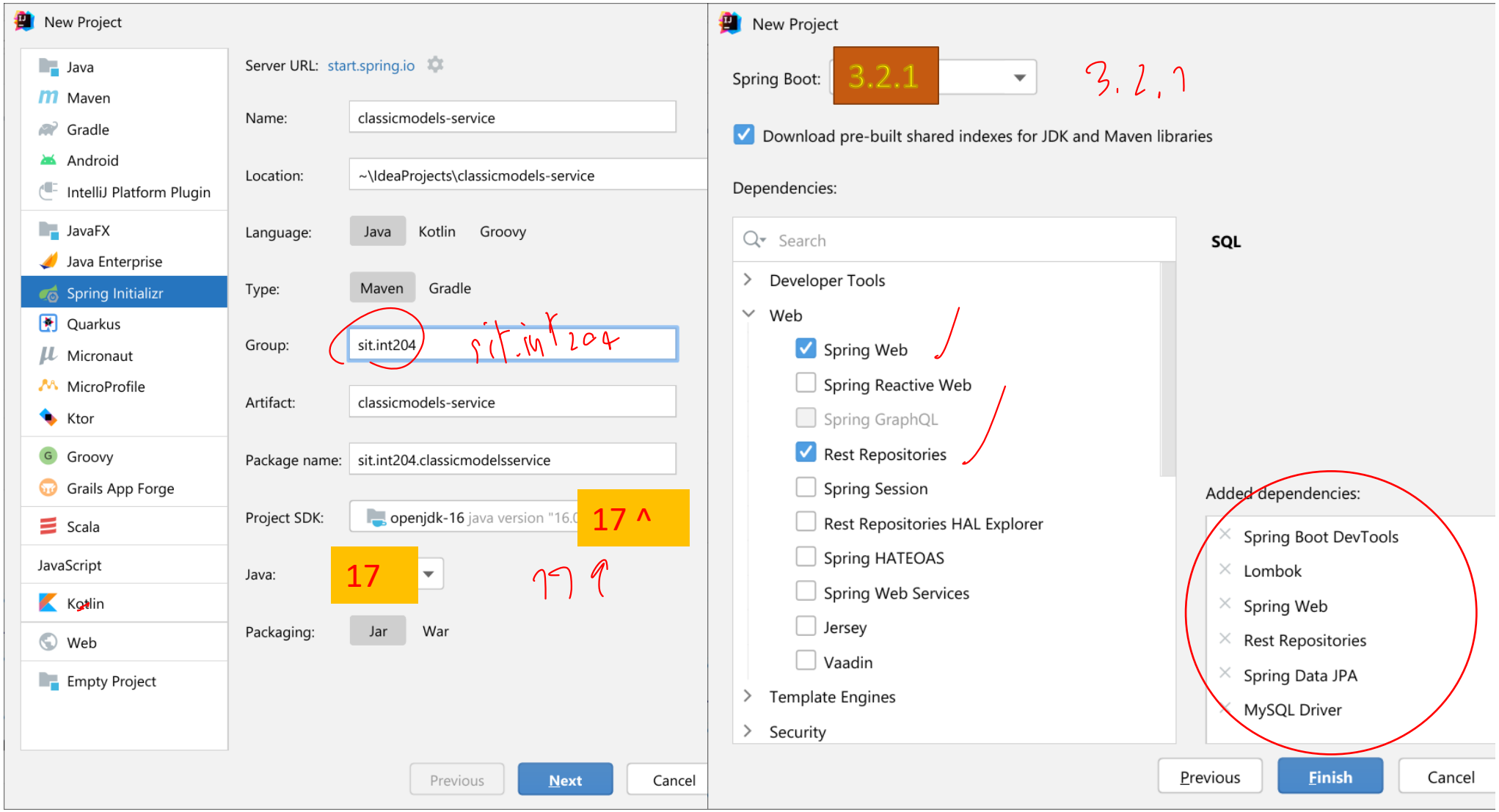
Made with Goodnotes

MVC Web Application Architectures (spa)



Made with Goodnotes

Step 1: Initializing a Spring Boot Project



Made with Goodnotes

Step 2: Connecting Spring Boot to the Database

ceController.java | application.properties | Office.java

อยู่ใน project

```
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.password=143900
spring.datasource.username=root
spring.datasource.url=jdbc:mysql://localhost:3306/classicmodels
spring.jpa.hibernate.ddl-auto=none
spring.jpa.hibernate.naming.physical-strategy=org.hibernate.boot.model.nam
```

```
org.hibernate.boot.model.naming.PhysicalNamingStrategy
```

ถ้าไม่กำหนด spring.jpa.hibernate.naming.physical-strategy

จะใช้ convention ดังนี้

การระบุ `@Column(name)` ใน `enitity clase` ต้องพิมพ์เป็นตัวเล็กหมด หรือถ้าระบุคอลัมน์เป็น `camel case` ชื่อฟิลด์ ใน ตารางต้องแยกคำด้วย ขีดล่าง `( _ )`

Made with Goodnotes

Step 3: Creating an Office Model (1)

```
@Getter  @Setter
@Entity @Table(name = "offices")
public class Office {
    @Id
    @Column(name = "officeCode", nullable = false, length = 10)
    private String officeCode;
    @Column(name = "city", nullable = false, length = 50)
    private String city;
    @Column(name = "phone", nullable = false, length = 50)
    private String phone;
    @Column(name = "addressLine1", nullable = false, length = 50)
    private String addressLine1;
```

Made with Goodnotes

Step 3: Creating an Office Model (2)

```
    @Column(name = "addressLine2", length = 50)
    private String addressLine2;
    @Column(name = "state", length = 50)
    private String state;
    @Column(name = "country", nullable = false, length = 50)
    private String country;
    @Column(name = "postalCode", nullable = false, length = 15)
    private String postalCode;
    @Column(name = "territory", nullable = false, length = 10)
    private String territory;

    @JsonIgnore
    @OneToMany(mappedBy = "office")
    private Set<Employee> employees = new LinkedHashSet<>();
```

Made with Goodnotes

Step 3: Creating an Employee Model (1)

```
@Getter @Setter
@Entity @Table(name = "employees")
public class Employee {
    @Id
    @Column(name = "employeeNumber", nullable = false)
    private Integer id;

    @JsonIgnore
    @ManyToOne
    @JoinColumn(name = "office")
    private Office office;

    @Column(name = "lastName", nullable = false, length = 50)
```

Made with Goodnotes

Step 3: Creating an Employee Model (2)

```
private String lastName;
    @Column(name = "firstName", nullable = false, length = 50)
    private String firstName;
    @Column(name = "extension", nullable = false, length = 10)
    private String extension;
    @Column(name = "email", nullable = false, length = 100)
    private String email;

    @JsonIgnore
    @ManyToOne
    @JoinColumn(name = "reportsTo")
    private Employee employees;

    @Column(name = "jobTitle", nullable = false, length = 50)
    private String jobTitle;
```

Made with Goodnotes

Step 4: Creating Repository Interface

```
public interface OfficeRepository extends JpaRepository<Office, String> {

}
```

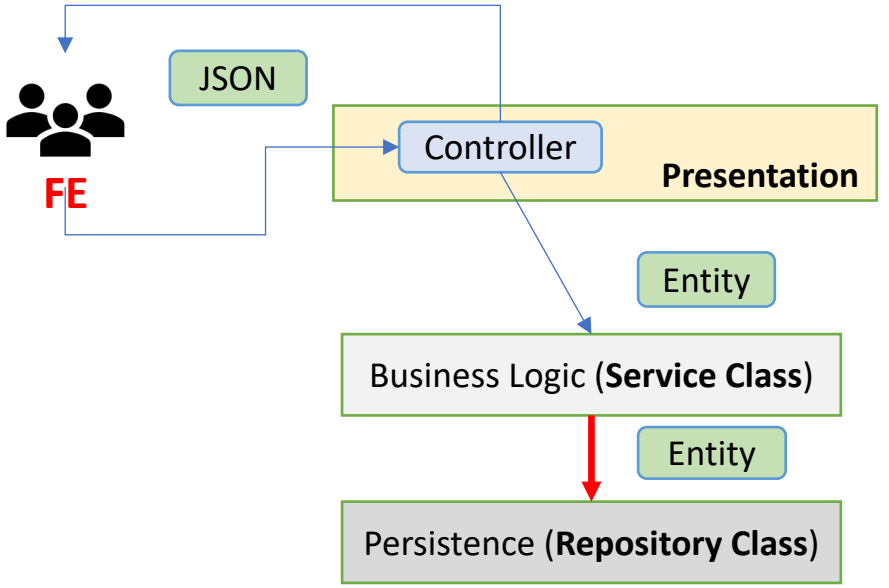
```
public interface EmployeeRepository extends JpaRepository<Employee, Integer>
{

}
```

```
public interface CustomerRepository extends JpaRepository<Customer, Integer>
{

}
```

Made with Goodnotes

Step 5: Creating Service Class

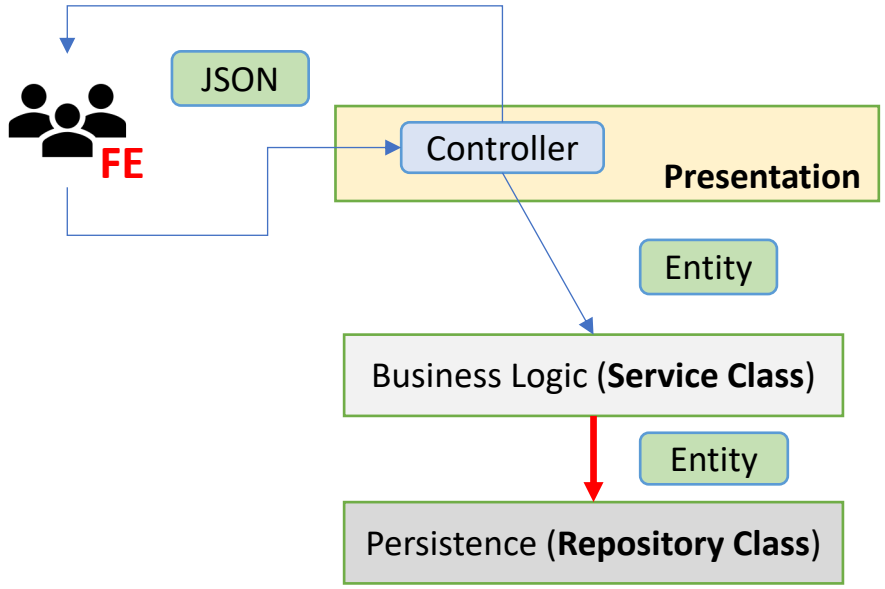
```
@Service
public class OfficeService {
    @Autowired
    private OfficeRepository repository;

    public List<Office> getAllOffice() {
        return repository.findAll();
    }

    public Office getOffice(String officeCode) {
        return repository.findById(officeCode).orElseThrow(
                () -> new HttpClientErrorException(HttpStatus.NOT_FOUND,
                        "Office Id " + officeCode + " DOES NOT EXIST !!!") {
                }
        );
    }


    @Transactional
    public Office createNewOffice(Office office) {
        return repository.save(office);
    }
```



Made with Goodnotes

Step 5: Creating Service Class (cont.)

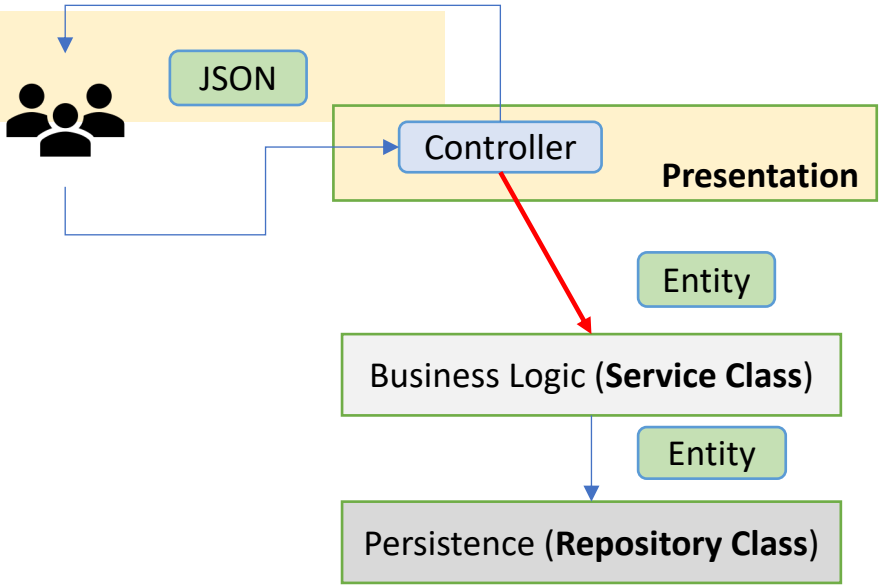
```
@Transactional
public void removeOffice(String officeCode) {
    Office office = repository.findById(officeCode).orElseThrow(
        () -> new HttpClientErrorException(HttpStatus.NOT_FOUND, "Office Id " + officeCode + " DOES NOT EXIST !!!")
    );
    repository.delete(office);
  }

  @Transactional
  public Office updateOffice(String officeCode, Office office) {
    if(office.getOfficeCode()!=null && !office.getOfficeCode().trim().isEmpty()) {
      if (!office.getOfficeCode().equals(officeCode)) {
        throw new HttpClientErrorException(HttpStatus.BAD_REQUEST,
            "Conflict Office code  !!! (" + officeCode +
            " vs " + office.getOfficeCode() + ")");
      }
    }
    Office existingOffice = repository.findById(officeCode).orElseThrow(
        () -> new HttpClientErrorException(HttpStatus.NOT_FOUND,
            "Office Id " + officeCode + " DOES NOT EXIST !!!"));
    return repository.save(office);
  }
}
```



Made with Goodnotes

Step 6: Creating Controller

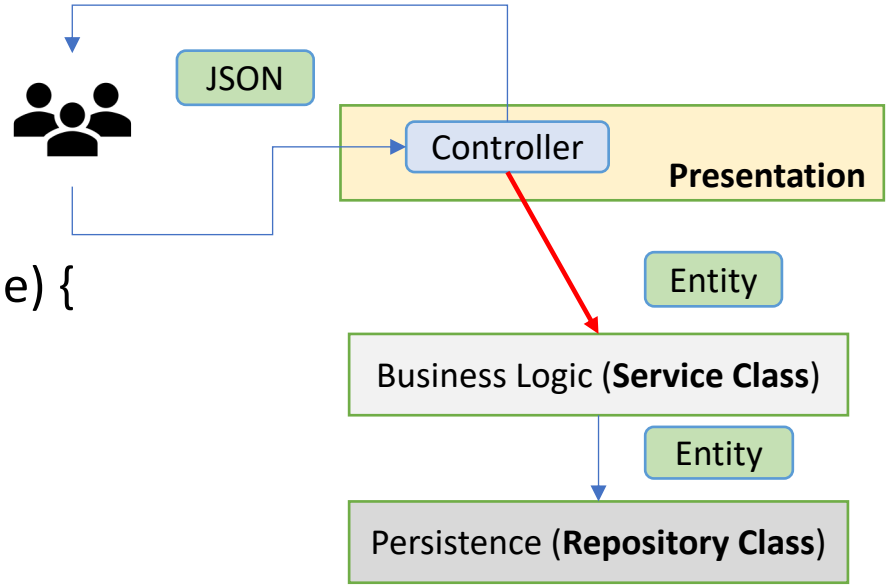
```
@RestController
@RequestMapping("/api/offices")
public class OfficeController {
    @Autowired
    private OfficeService service;

    @GetMapping("")
    public List<Office> getAllOffices() {
        return service.getAllOffice();
    }

    @GetMapping("/{officeCode}")
    public Office getOfficeById(@PathVariable String officeCode) {
        return service.getOffice(officeCode);
    }


    @PostMapping("")
    public Office addNewOffice(@RequestBody Office office) {
         return service.createNewOffice(office);
    }
```



Made with Goodnotes

Step 6: Creating Controller (cont.)

```
    @PutMapping("/{officeCode}")
    public Office updateOffice(@RequestBody Office office, @PathVariable String officeCode) {
        return service.updateOffice(officeCode, office);
    }


    @DeleteMapping("/{officeCode}")
     public void removeOffice(@PathVariable String officeCode) {
         service.removeOffice(officeCode);
     }
}
```



Made with Goodnotes

Step 6: Testing the APIs (GET)

GET localhost:8080/api/offices

Params Authorization Headers (7) Body Pre-request Script

Body Cookies Headers (5) Test Results

Pretty Raw Preview Visualize JSON


```
[
    {
        "id": "1",
        "city": "San Francisco",
        "phone": "+1 650 219 4782",
        "addressLine1": "100 Market Street",
        "addressLine2": "Suite 300",
        "state": "CA",
        "country": "USA",
        "postalCode": "94080",
        "territory": "NA"
    },
    {
        "id": "2",
        "city": "Boston",
        "phone": "+1 215 837 0825",
        "addressLine1": "1550 Court Place",
```


GET localhost:8080/api/offices/7

Params Authorization Headers (7) Body Pre-rec

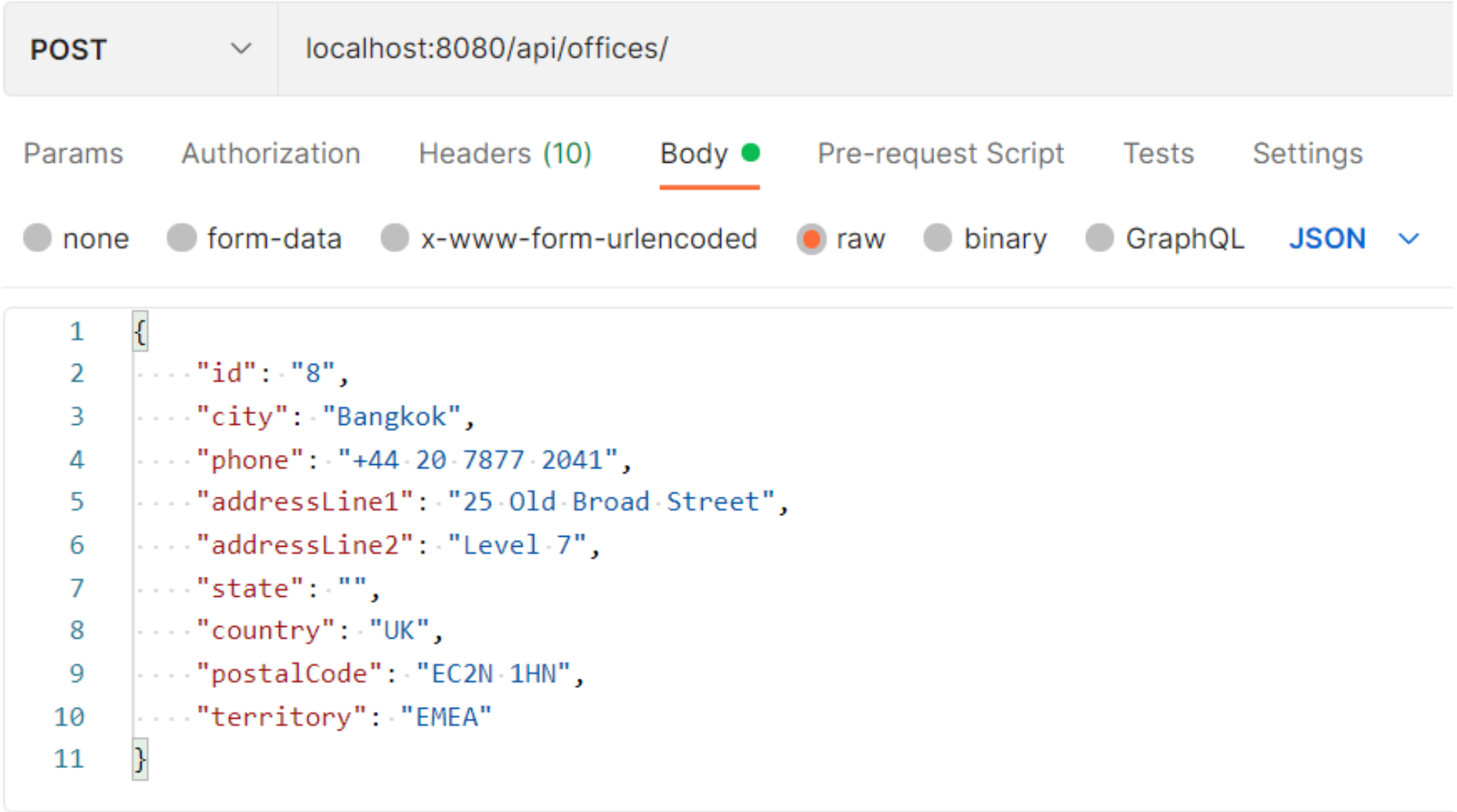
Body Cookies Headers (5) Test Results

Pretty Raw Preview Visualize JSON


```
{
    "id": "7",
    "city": "London",
    "phone": "+44 20 7877 2041",
    "addressLine1": "25 Old Broad Street",
    "addressLine2": "Level 7",
    "state": null,
    "country": "UK",
    "postalCode": "EC2N 1HN",
    "territory": "EMEA"
}
```

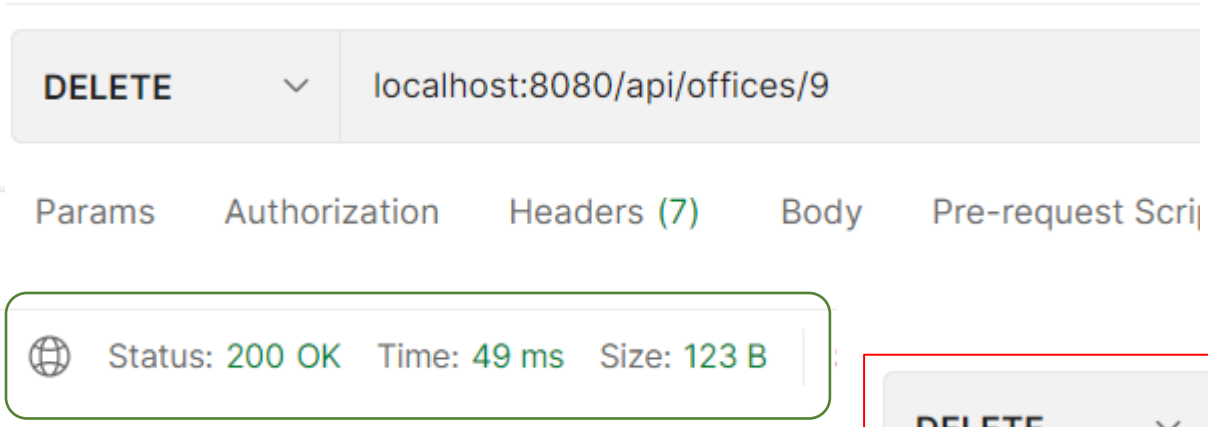

Made with Goodnotes

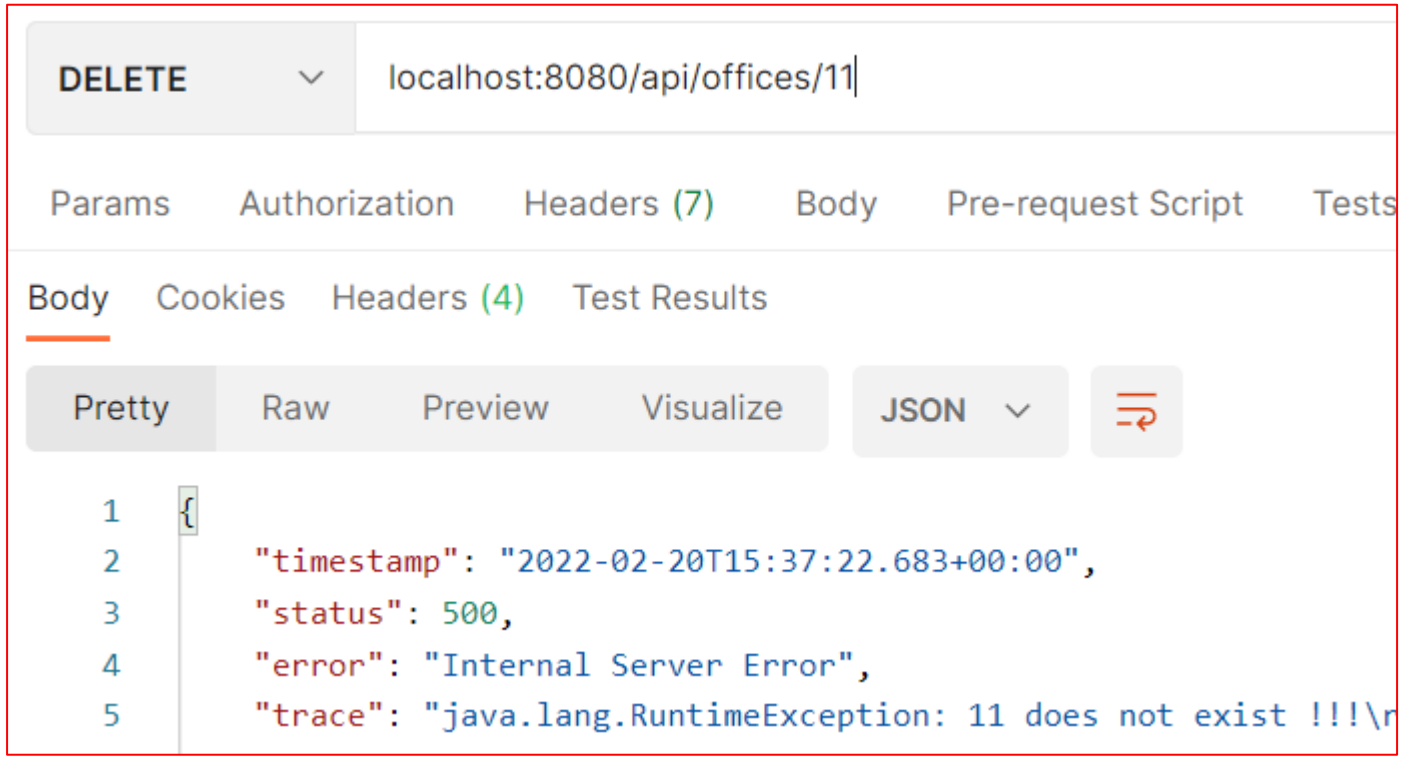
Step 7: Testing the APIs (POST)



Made with Goodnotes

Step 8: Testing the APIs (DELETE)





Made with Goodnotes

Step 8: Testing the APIs (PUT)

Made with Goodnotes

Made with Goodnotes

JSON : JavaSrcipt Object Notation

- A lightweight data-interchange format.
- Easy for humans to read and write.
- Easy for machines to parse and generate.
- Based on a subset of the JavaScript Programming Language Standard ECMA-262 3rd Edition - December 1999.
- JSON is a text format that is completely language independent but uses conventions that are familiar to programmers of the C-family of languages, including C, C++, C#, Java, JavaScript, Perl, Python, and many others.

https://www.json.org/json-en.html

https://education.launchcode.org/js-independent-track/chapters/fetch-json/index.html

Made with Goodnotes

RESTful Resource

URI	HTTP Verb	Description
api/offices	GET	Get all office
api/offices/1	GET	Get an office with id = 1
api/offices/1/employees	GET	Get all employee for office id = 1
api/offices	POST	Add new office
api/offices/1	PUT	Update an office with id = 1
api/offices/1	DELETE	Delete an office with id = 1

Made with Goodnotes

REST API (also known as RESTful API)

- REST stands for **RE**presentational **S**tate **T**ransfer and was created by computer scientist Roy Fielding.
- An application programming interface (API or web API) that conforms to the constraints of REST architectural style and allows for interaction with RESTful web services.
- In REST architecture, a REST Server simply provides access to resources and REST client accesses and modifies the **resources**.
- Each **resource** is **identified** by URIs/ global IDs.
- REST uses various representation to represent a resource like text, JSON, XML. JSON is the most popular one.

Rest API ทำหน้าที่
client เข้ามาใช้
แก้ไข resource access ด้วย
URIs / globalIDs

show status

Made with Goodnotes

Rules of REST API

- There are certain rules which should be kept in mind while creating REST **API endpoints**.
  - REST is based on the resource or **noun** instead of action or verb based. It means that a URI of a REST API should always end with a noun. Example: **/api/users** is a good example.
  - HTTP verbs are used to identify the action. Some of the HTTP verbs are – GET, PUT, POST, DELETE, PATCH.
  - A web application should be organized into resources like users and then uses HTTP verbs like – GET, PUT, POST, DELETE to modify those resources. And as a developer it should be clear that what needs to be done just by looking at the endpoint and HTTP method used.

ห้ามใช้ Verb / ใช้ Noun named endpoint..

Made with Goodnotes

URI + HTTP verb

RESTful Resource Example

URI	HTTP verb	Description
api/users	GET	Get all users
api/users/new	GET	Show form for adding new user
api/users	POST	Add a user
api/users/1	PUT	Update a user with id = 1
api/users/1/edit	GET	Show edit form for user with id = 1
api/users/1	DELETE	Delete a user with id = 1
api/users/1	GET	Get a user with id = 1

Always use plurals in URL to keep an API URI consistent throughout the application.
Send a proper HTTP code to indicate a success or error status.

Made with Goodnotes

RESTFul Principles and Constraints

- RESTFul Client-Server
  - Server will have a RESTful web service which would provide the required functionality to the client.
  - The client send's a request to the web service on the server. The server would either reject the request or comply and provide an adequate response to the client.
- Stateless
  - Statelessness mandates that each request from the client to the server must contain all of the information necessary to understand and complete the request.
  - The server cannot take advantage of any previously stored context information on the server.
  - For this reason, the client application must entirely keep the session state.

Made with Goodnotes

RESTFul Principles and Constraints (2)

- Interface/Uniform Contract
  - This is the underlying technique of how RESTful web services should work. RESTful basically works on the HTTP web layer and uses the below key verbs to work with resources on the server.
    - POST – To create a resource on the server
    - GET – To retrieve a resource from the server
    - PUT – To change the state of a resource or to update it
    - DELETE – To remove or delete a resource from the server
- Code on demand (optional)
  - REST also allows client functionality to extend by downloading and executing code in the form of applets or scripts.
  - Servers can provide part of features delivered to the client in the form of code, and the client only needs to execute the code.

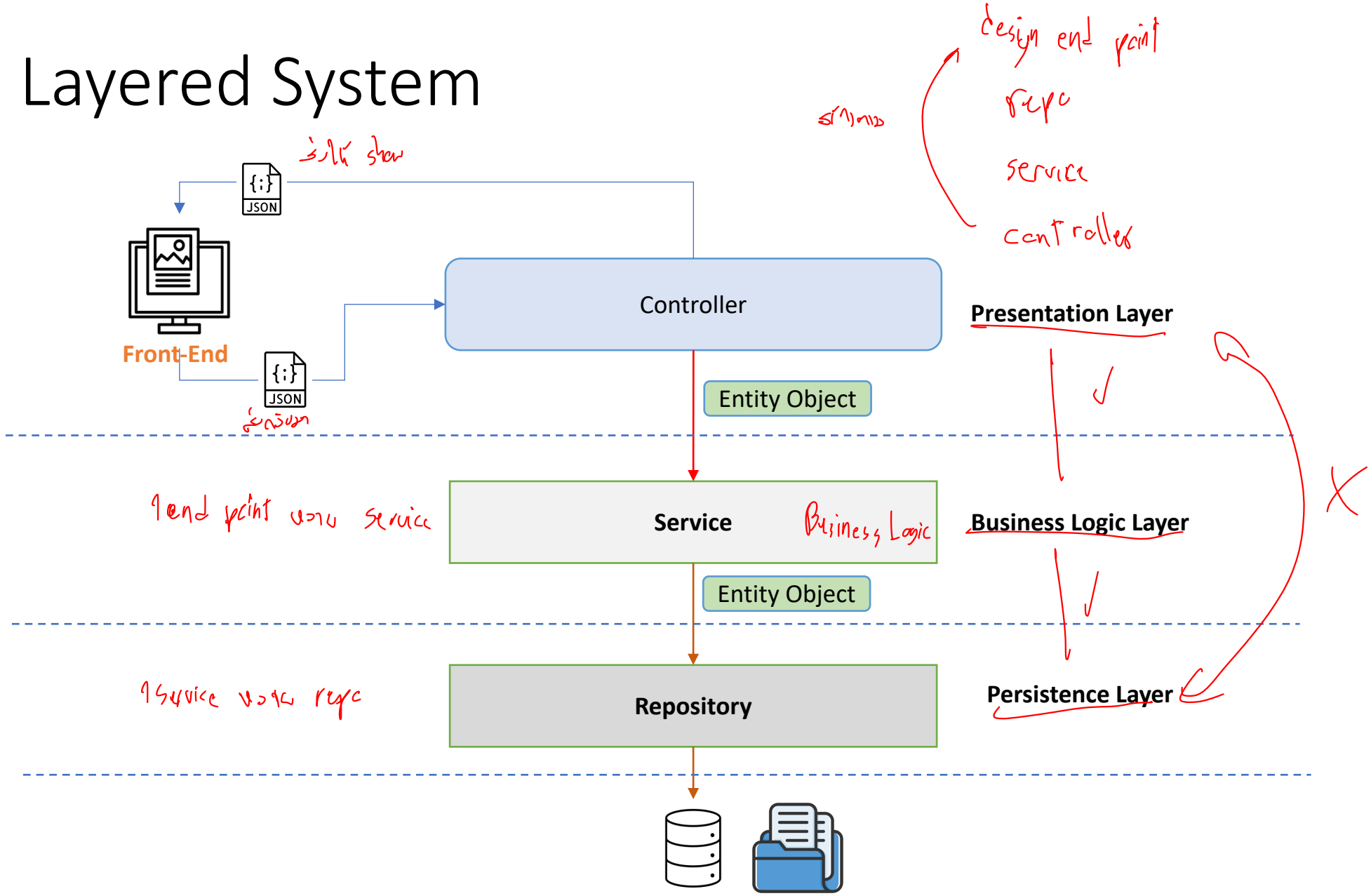
Made with Goodnotes

RESTFul Principles and Constraints (3)

- Cacheable
  - The cacheable constraint requires that a response should implicitly or explicitly label itself as cacheable or non-cacheable.
  - If the response is cacheable, the client application gets the right to reuse the response data later for equivalent requests and a specified period.
- Layered system
  - The layered system style allows an architecture to be composed of hierarchical layers by constraining component behavior.
  - For example, in a layered system, each component cannot see beyond the immediate layer they are interacting with.

Made with Goodnotes

Layered System



Made with Goodnotes

Controller

```
@RestController
@RequestMapping("/api/offices")
public class OfficeController {
   @Autowired
   private OfficeService service;

   @GetMapping("")
   public List<Office> getAllOffices() {
      return service.getAllOffice();
   }

   @GetMapping("/{officeCode}")
   public Office getOfficeById(@PathVariable String officeCode) {
      return service.getOffice(officeCode);
   }


   @PostMapping("")
   public Office addNewOffice(@RequestBody Office office) {
       return service.createNewOffice(office);
   }
```

In Spring's approach to building RESTful web services, HTTP requests are handled by a controller. These components are identified by the @RestController annotation, the data returned by each method will be written straight into the response body instead of rendering a template.

You can use the @RequestMapping annotation to map requests to controllers methods. It has various attributes to match by URL, HTTP method, request parameters, headers, and media types. You can use it at the class level to express shared mappings or at the method level to narrow down to a specific endpoint mapping.

An annotation used in Spring Boot to enable **automatic dependency injection**. It allows the Spring container to provide an instance of a required dependency when a bean is created. This annotation can be used on fields, constructors, and methods to have Spring provide the dependencies automatically

Made with Goodnotes

What is a Postman

- Postman is an API platform for building and using APIs. Postman simplifies each step of the API lifecycle and streamlines collaboration so you can create better APIs—faster.

Made with Goodnotes

Creating a Spring Boot REST API Project

- Step 1: Initializing a Spring Boot Project
- Step 2: Connecting Spring Boot to the Database
- Step 3: Creating a User Model
- Step 4: Creating Repository Interface (Persistence Layer)
- Step 5: Creating Service Classes (Business Layer)
- **Step 6: Creating a Rest Controller** (Presentation Layer)
- Step 7: Compile, Build and Run
- **Step 8: Testing the APIs** 

Made with Goodnotes

Spring Boot Overview
+
Introduction to
Auto Configuration & Spring Data JPA

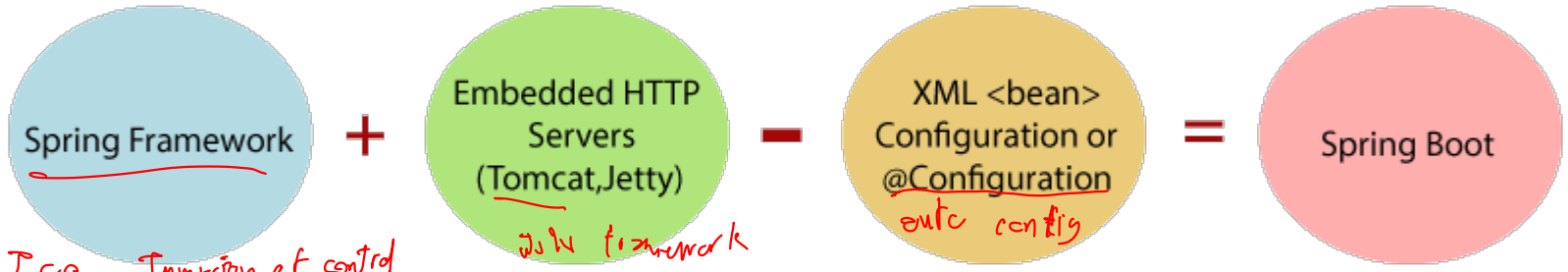
By

Pichet Limvajiranan

Made with Goodnotes

What is Spring Boot?

- Spring Boot is a project that is built on the top of the Spring Framework.
- It provides an easier and faster way to set up, configure, and run both simple and web-based applications.
- It allows us to build a stand-alone application with minimal or zero configurations.
- It is better to use if we want to develop a simple Spring-based application or RESTful services.



Made with Goodnotes

Spring Boot Features

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

Made with Goodnotes

Why should we use Spring Boot Framework?

We should use Spring Boot Framework because:

- The dependency injection approach is used in Spring Boot.
- It contains powerful database transaction management capabilities.
- It simplifies integration with other Java frameworks like JPA/Hibernate ORM, Struts, etc.
- It reduces the cost and development time of the application.

Made with Goodnotes

Spring Boot: Auto Configuration

- The problem with Spring and Spring MVC is the amount of configuration that is needed

```
<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="prefix"><value>/WEB-INF/views/</value></property>
    <property name="suffix"><value>.jsp</value> </property>
</bean>

<mvc:resources mapping="/webjars/**" location="/webjars/"/>
```

Spring framework

- Spring Boot solves this problem through a combination of **Auto Configuration** and **Starter Projects**.
  - Spring Boot looks at Frameworks available on the CLASSPATH then Existing configuration for the application.
  - Based on these, Spring Boot provides basic configuration needed to configure the application with these frameworks.
  - This is called Auto Configuration.

auto config

Made with Goodnotes

Spring Boot: Starter Projects

- Starters are a set of convenient dependency descriptors that you can include in your application.
- You get a one-stop-shop for all the Spring and related technology that you need, without having to hunt through sample code and copy paste loads of dependency descriptors.
- example starter - Spring Boot Starter Web.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

Made with Goodnotes

Spring Boot Starter Project Options

- spring-boot-starter-web-services - SOAP Web Services
- **spring-boot-starter-web - Web & RESTful applications**
- spring-boot-starter-test - Unit testing and Integration Testing
- spring-boot-starter-jdbc - Traditional JDBC
- spring-boot-starter-hateoas - Add HATEOAS features to your services
- **spring-boot-starter-security - Authentication and Authorization using Spring Security**
- **spring-boot-starter-data-jpa - Spring Data JPA with Hibernate**
- spring-boot-starter-cache - Enabling Spring Framework's caching support
- spring-boot-starter-data-rest - Expose Simple REST Services using Spring Data REST

Made with Goodnotes

Auto Configuration & Property Default Value example

1) Add com.h2database dependency to classicmodels-service project & reload maven
2) Comment all properties in application.properties

```
#spring.datasource.driver-class-name=com.mysql.
#spring.datasource.username=root
#spring.datasource.password=143900
#spring.datasource.url=jdbc:mysql://localhost:
```

```
<dependency>
   <groupId>com.h2database</groupId>
   <artifactId>h2</artifactId>
   <version>2.2.224</version>
   <scope>runtime</scope>
</dependency>
```

3) Run/Test all end-point with postman
   - /api/offices POST
   - /api/offices GET
   - /api/offices/{id} GET
   - /api/offices/{id} PUT
   - /api/offices/{id} DELETE

Made with Goodnotes

Creating Spring Boot Projects

- Using Spring Initializr
  - A great web to bootstrap your Spring Boot projects.
  - https://start.spring.io
- Using the Spring Tool Suite (STS)
  - The Spring Tool Suite (STS: https://spring.io/tools/sts) is an extension of the Eclipse IDE with lots of Spring framework related plugins.
- Using IDE Bundled tool.

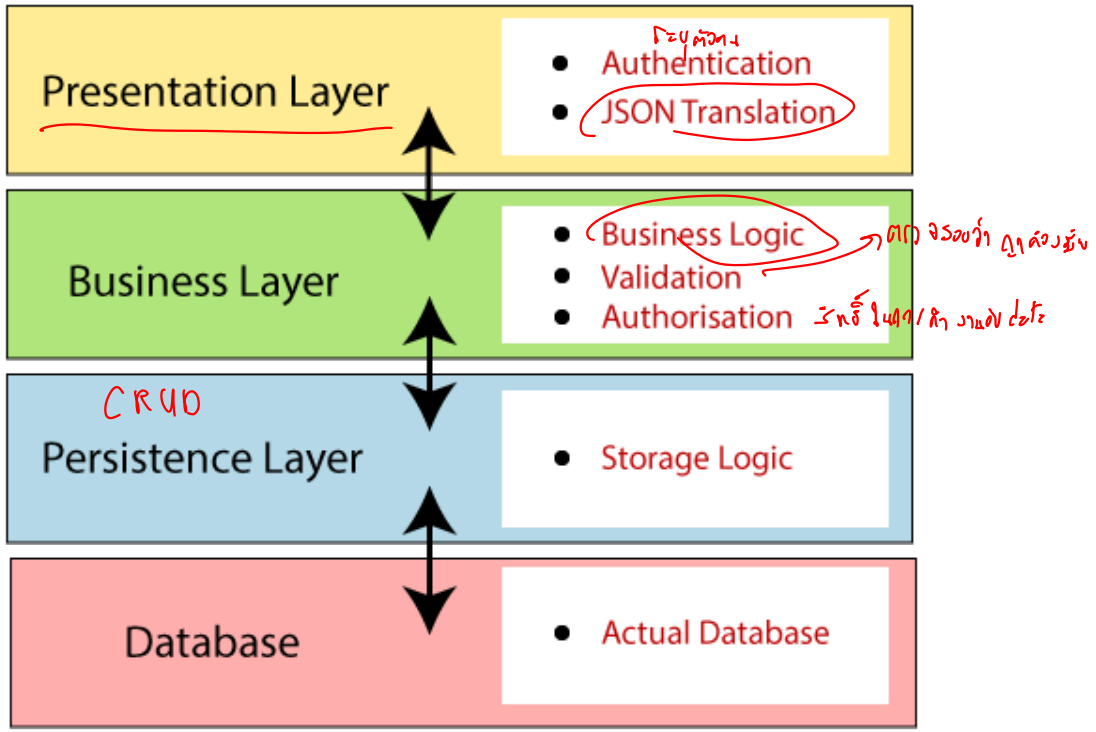
Maven Wrapper:

```
mvnw dependency:tree
mvnw spring-boot:run
```

Made with Goodnotes

Spring Boot Architecture

- Spring Boot follows a layered architecture in which each layer communicates with the layer directly below or above (hierarchical structure) it.
  - Presentation Layer
  - Business Layer
  - Persistence Layer
  - Database Layer



Made with Goodnotes

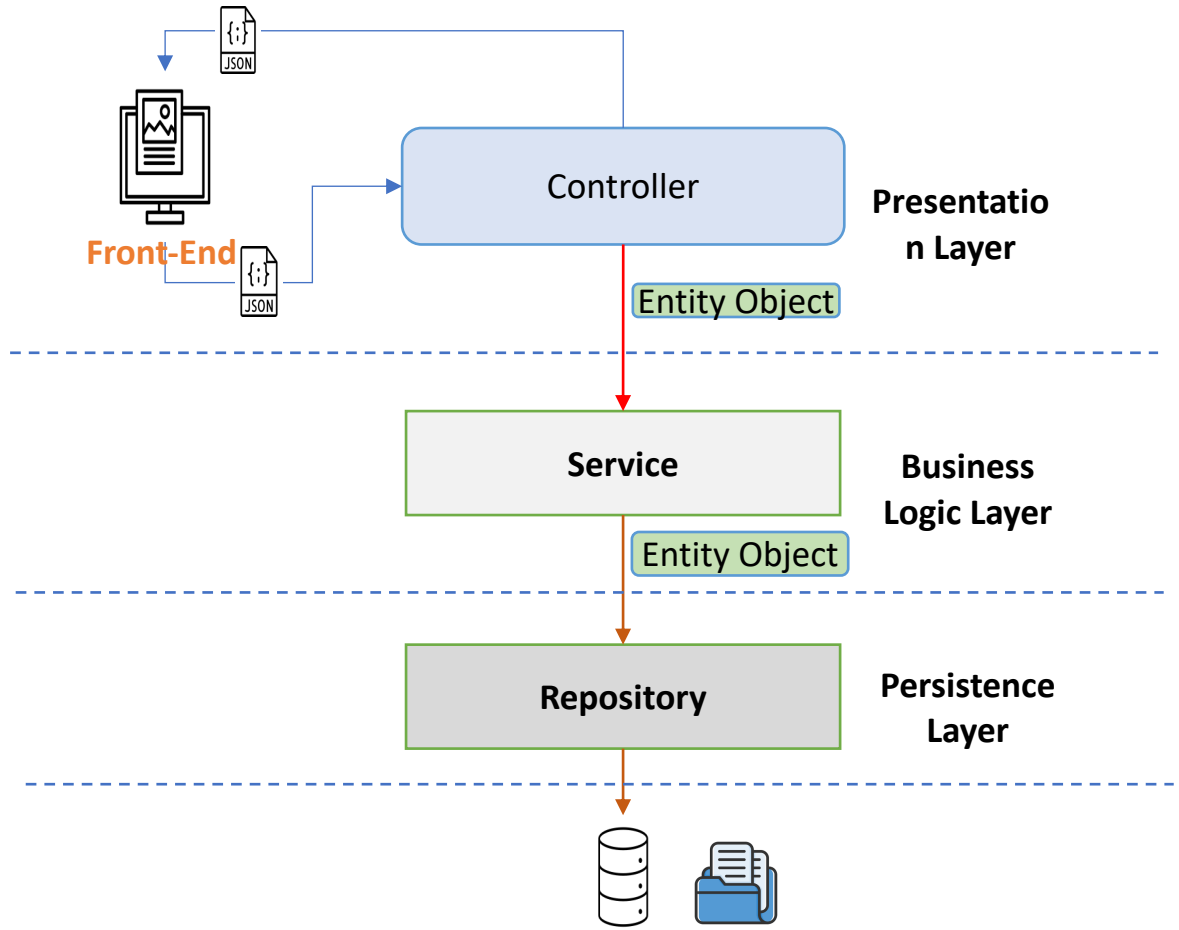
Spring Boot Layers

- Presentation Layer:
  - Handles the HTTP requests, translates the JSON parameter to object, and authenticates the request and transfer it to the business layer. In short, it consists of views i.e., frontend part.
- Business Layer:
  - Handles all the business logic. It consists of service classes and uses services provided by data access layers. It also performs authorization and validation.
- Persistence Layer:
  - Contains all the storage logic and translates business objects from and to database rows.
- Database Layer:
  - Perform CRUD (create, retrieve, update, delete) operations.

Made with Goodnotes

Spring Boot Flow Architecture

- Spring Boot uses all the modules of Spring-like Spring MVC, Spring Data, etc.
- Creates a data access layer and performs CRUD operation.
- The client makes the HTTP requests (GET or POST).
- The request goes to the controller, and the controller maps that request and handles it. After that, it calls the service logic if required.
- In the service layer, all the business logic performs. It performs the logic on the data that is mapped to JPA with model classes.
- A HTTP Response is returned to the user if no error occurred.



Made with Goodnotes

Spring Framework Annotations

- Basically, there are 6 types of annotation available in the whole spring framework.

HTTP protocol → Presentation Layer

1. Spring Web Annotations
2. Spring Core Annotations
3. Spring Boot Annotations
4. Spring Scheduling Annotations
5. Spring Data Annotations
6. Spring Bean Annotations

Made with Goodnotes

1) Spring Web Annotations

- Present in the org.springframework.web.bind.annotation
- Some of the annotations that are available in this category are:
  - **@RequestMapping**
  - **@RequestBody**
  - **@PathVariable**
  - @RequestParam
  - Response Handling Annotations
    - @ResponseBody @ExceptionHandler @ResponseStatus
  - @Controller
  - **@RestController**
  - @ModelAttribute
  - @CrossOrigin

Made with Goodnotes

Spring Web Annotation example

```
@RestController
@RequestMapping("/api/offices")
public class OfficeController {
    :
    :
   @GetMapping("/{officeCode}")
   public Office getOfficeById(@PathVariable String officeCode) {
      return service.getOffice(officeCode);
   }


   @PostMapping("")
   public Office addNewOffice(@RequestBody Office office) {
       return service.createNewOffice(office);
  }
```

Made with Goodnotes

Practice

- Create API service for calculate student grade as following requirements:
  - input

```
{ "id" : 1001,
  "name" : "Somchai",
   "score" : 78
}
```

  - output

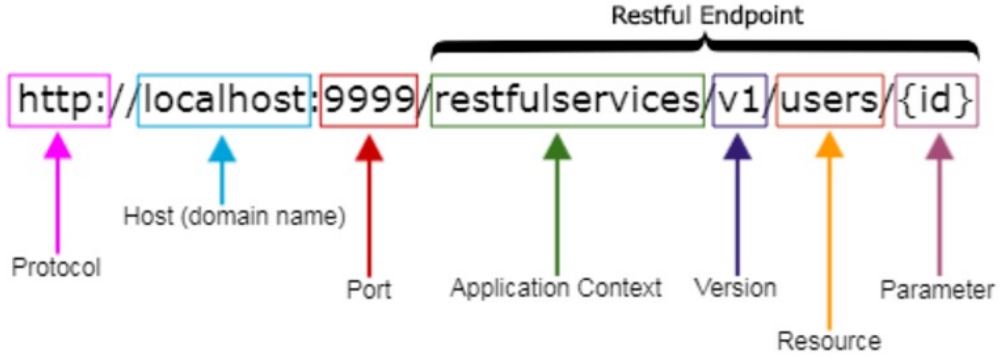
```
{ "id" : 1001,
  "name" : "Somchai",
   "score" : 78,
   "grade" : "B"
}
```

TO DO:
- Specify URI & Method
- Create Entity
- Create Service
- Create Controller & method

Business rule :
- Score is 80 and up, A
- Score 70 to 79, B
- Score 60 to 69, C
- Score 50 to 59, D
- Score is 49 and lower, F

Made with Goodnotes

REST API URI Naming Conventions and Best Practices



- Singleton and Collection Resources

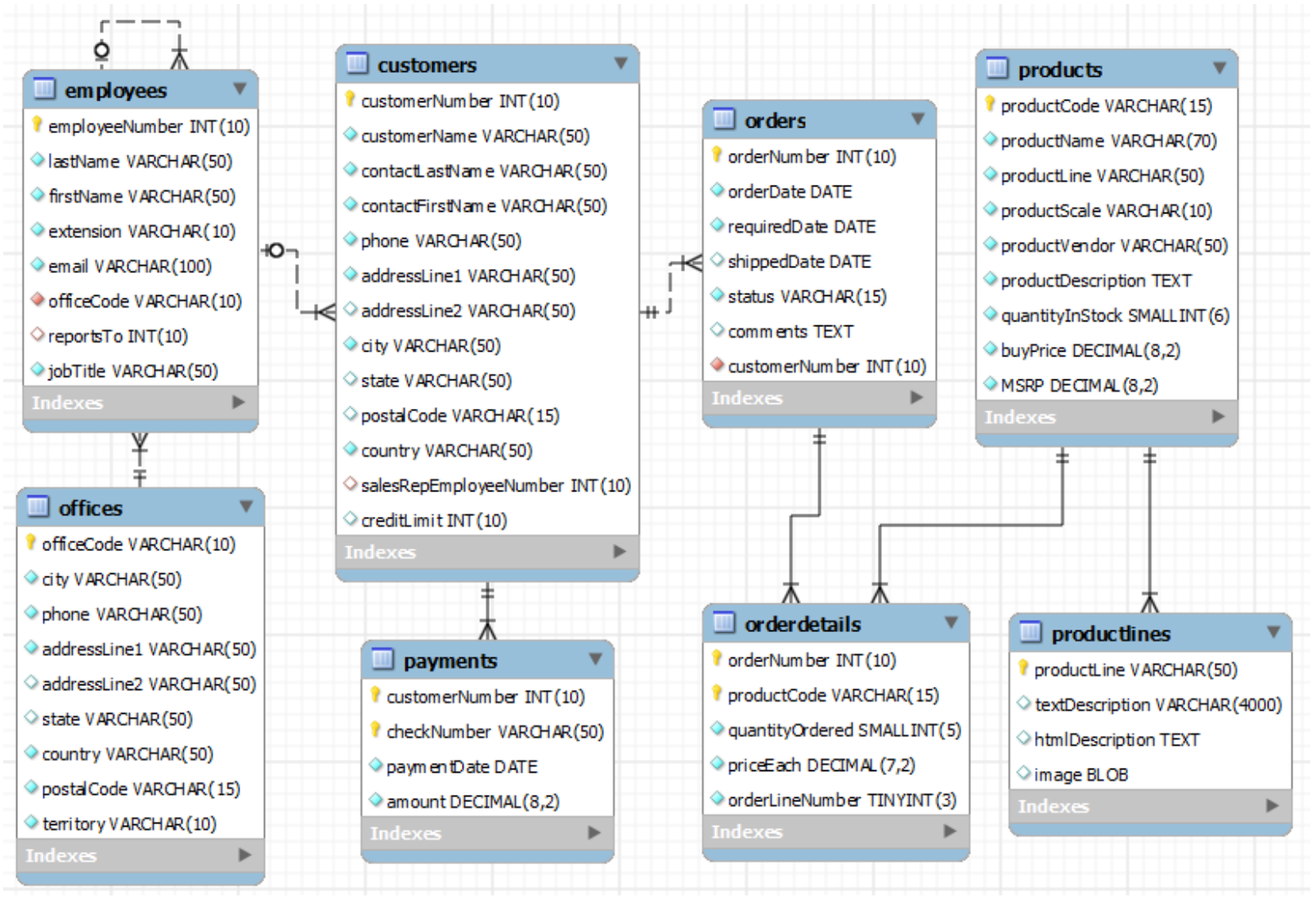
```
/customers                  // is a collection resource
/customers/{id}             // is a singleton resource
```

- Sub-collection Resources

```
/customers/{id}/orders                  // is a sub-collection resource
```

- Best Practices
  - https://medium.com/@nadinCodeHat/rest-api-naming-conventions-and-best-practices-1c4e781eb6a5
  - https://restfulapi.net/resource-naming

Made with Goodnotes



Excercise

Design REST Resources representation for the Classicmodels

Made with Goodnotes

Spring Core annotations

- Spring Core annotations are present in the 2 packages
  - org.springframework.beans.factory.annotation
  - org.springframework.context.annotation
- We can divide them into two broad categories:
  - DI-Related Annotations
    - **@Autowired** @Qualifier @Primary
    - @Bean @Lazy @Required
    - @Value @Scope
    - @Lookup, etc.
  - Context Configuration Annotations
    - @Profile @Import
    - @ImportResource @PropertySource, etc.

Made with Goodnotes

@Autowired

- We use @Autowired to mark the dependency that will be injected by the Spring container.
- Applied to the fields, **setter** methods, and **constructors**. It injects object dependency implicitly.

```
public class OfficeService {
    @Autowired
    private OfficeRepository repository;
```

```
public class OfficeService {
    private final OfficeRepository repository;
    @Autowired
    public OfficeService(OfficeRepository repository) {
        this.repository = repository;
    }
```

```
public class OfficeService {
    private final OfficeRepository repository;
    @Autowired
    public void setRepository(OfficeRepository repository) {
        this.repository = repository;
    }
```

Made with Goodnotes

Spring Boot Annotations

- @SpringBootApplication Combination of three annotations
  - @EnableAutoConfiguration
  - @ComponentScan
  - @Configuration.

```
@SpringBootApplication

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

Made with Goodnotes

Spring Bean Annotations

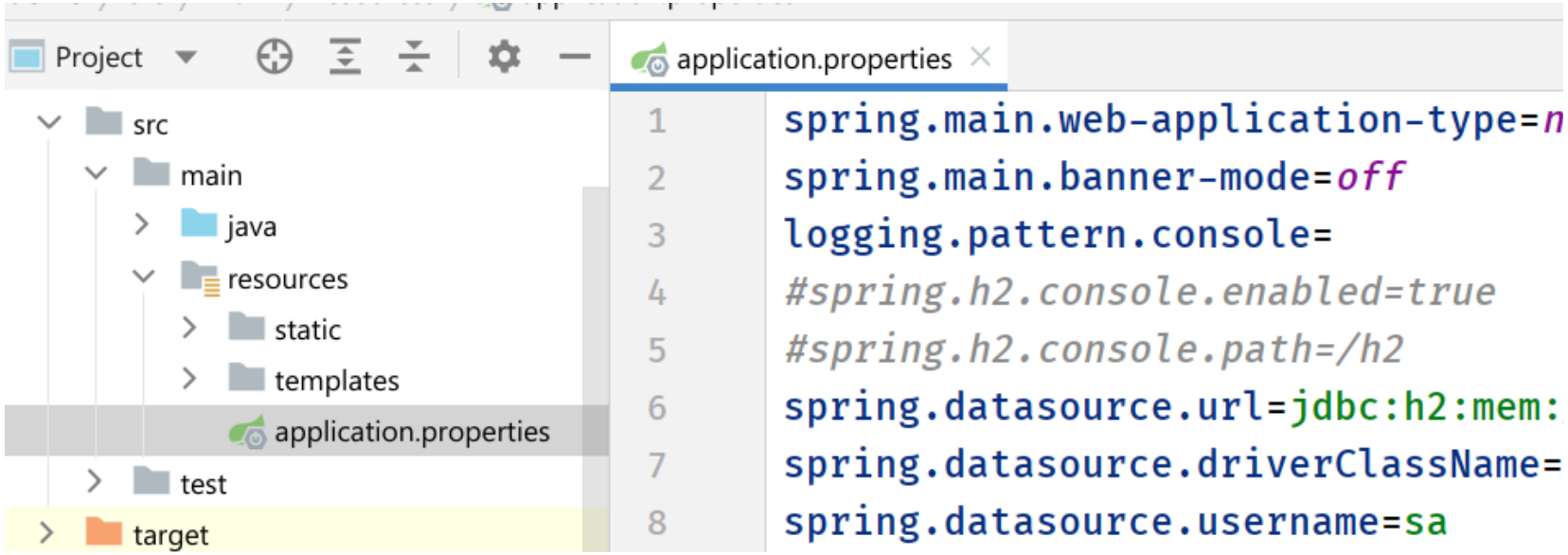
- Some of the annotations that are available in this category are:
  - @ComponentScan
  - @Configuration
  - Stereotype Annotations
    - @Component
      - **@Service**
      - **@Repository**
      - **@Controller**

@Service: We specify a class with @Service to indicate that they're holding the business logic. Besides being used in the service layer, there isn't any other special use for this annotation. The utility classes can be marked as Service classes.
@Repository: We specify a class with @Repository to indicate that they're dealing with **CRUD operations**, usually, it's used with DAO (Data Access Object) or Repository implementations that deal with database tables.
@Controller: We specify a class with @Controller to indicate that they're front controllers and responsible to handle user requests and return the appropriate response. It is mostly used with REST Web Services.

Made with Goodnotes

Spring Boot Application Properties

- Spring Boot Framework comes with a built-in mechanism for application configuration using a file called application.properties.
- It is located inside the src/main/resources folder.
- The properties have default values.
- We can set a property(s) for the Spring Boot application.



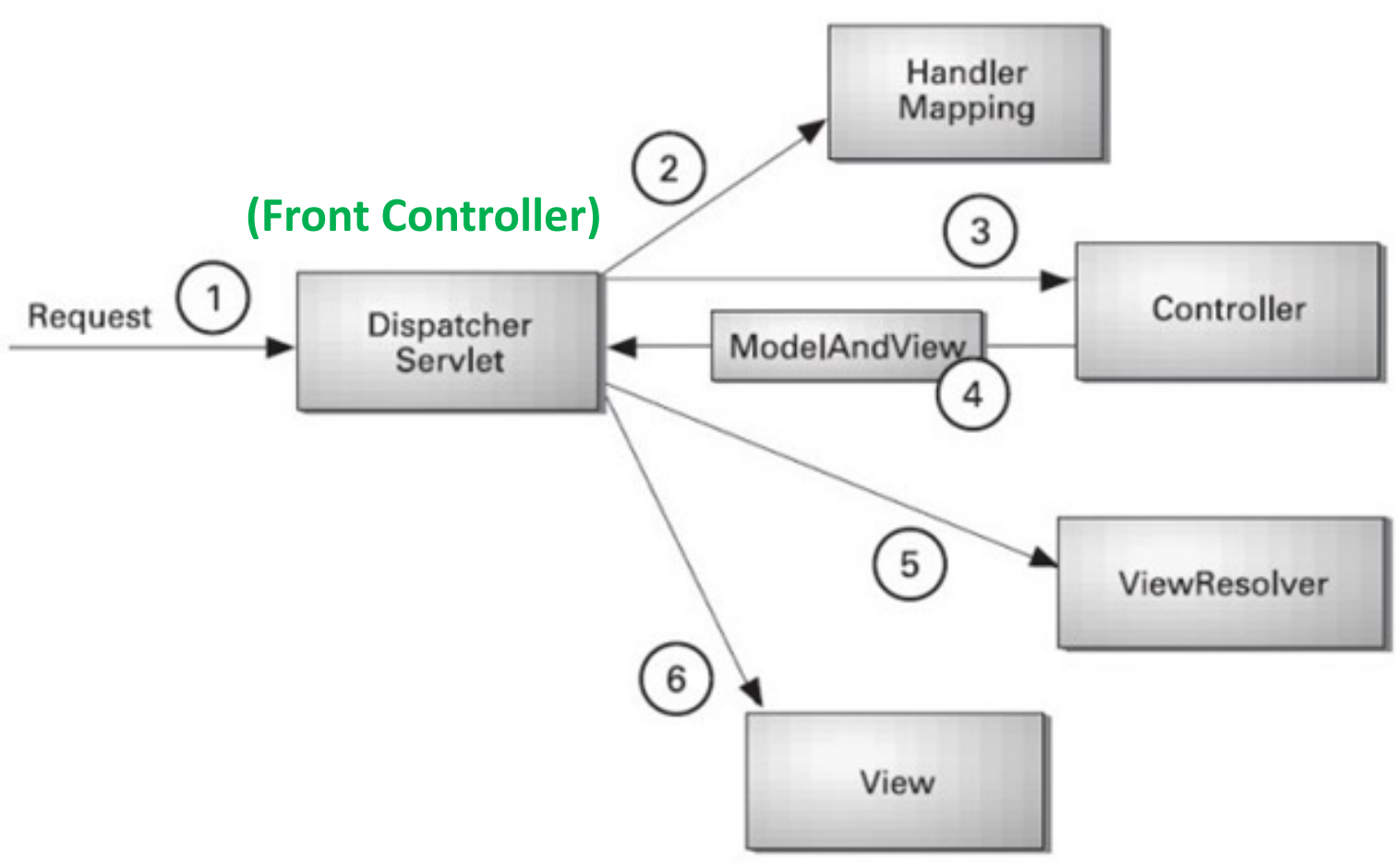
Made with Goodnotes

Spring Web MVC

- The Spring Web MVC framework provides Model-View-Controller (MVC) architecture and ready components that can be used to develop flexible and loosely coupled web applications.
- The MVC pattern results in separating the different aspects of the application (input logic, business logic, and UI logic), while providing a loose coupling between these elements.
- A Spring MVC provides an elegant solution to use MVC in spring framework by the help of DispatcherServlet.
    - In Spring Web MVC, the **DispatcherServlet** class works as the **front controller**. It is responsible to manage the flow of the Spring MVC application.

Made with Goodnotes

The DispatcherServlet and Flow of Spring Web MVC



Made with Goodnotes

Defining a Controller

- The DispatcherServlet delegates the request to the controllers to execute the functionality specific to it.
- The **@Controller** annotation indicates that a particular class serves the role of a controller.
- The @**RequestMapping** @**GetMapping** @**PostMapping** annotation is used to map a URL to either an entire class or a particular handler method.

```
@Controller
public class HelloController {
   @RequestMapping("/hello")
   public String printHello(ModelMap model) {
      model.addAttribute("message", "Hello Spring MVC Framework!");
      return "hello";
   }
}
```

Made with Goodnotes

Spring Boot Controller example

```java
@Controller
public class AppController {
    @Autowired
    private final StudentRepository studentRepository;

    @RequestMapping("/home")
    public String home() {
        return "home";
    }
    @GetMapping("/student-listing")
    public String students(Model model) {
        model.addAttribute("students", studentRepository.findAll());
        return "student-list";
    }
    @GetMapping("/student-list-plain-text")
    public ResponseEntity<String> students_list(Model model) {
        return new ResponseEntity<>(studentRepository.findAll()
                .toString(), HttpStatus.OK);
    }
```

Made with Goodnotes

Spring View Technology

- The Spring web framework is built around the MVC (Model-View-Controller) pattern, which makes it easier to separate concerns in an application.
- This allows for the possibility to use different view technologies, from the well established JSP technology to a variety of template engines.
  - Java Server Pages
  - Thymeleaf
  - FreeMarker
  - Groovy Markup Template Engine

Made with Goodnotes

Thymeleaf Template Engine example

```
<!DOCTYPE html>
<html lang="en“  xmlns=http://www.w3.org/1999/xhtml
          xmlns:th="http://www.thymeleaf.org">
<body>
<div class="container p4 m4">
    <h2>Student List:</h2><hr>
    <div class="row">
        <div class="col-2">Student Id</div>
        <div class="col-4">Name</div>
        <div class="col-2">GPAX</div>
    </div>
    <div class="row" th:each="student : ${students}">
        <div class="col-2" th:text="${student.id}"/>
        <div class="col-4" th:text="${student.name}"/>
        <div class="col-2" th:text="${student.gpax}"/>
    </div>
</div>
```

Made with Goodnotes

Spring Data JPA

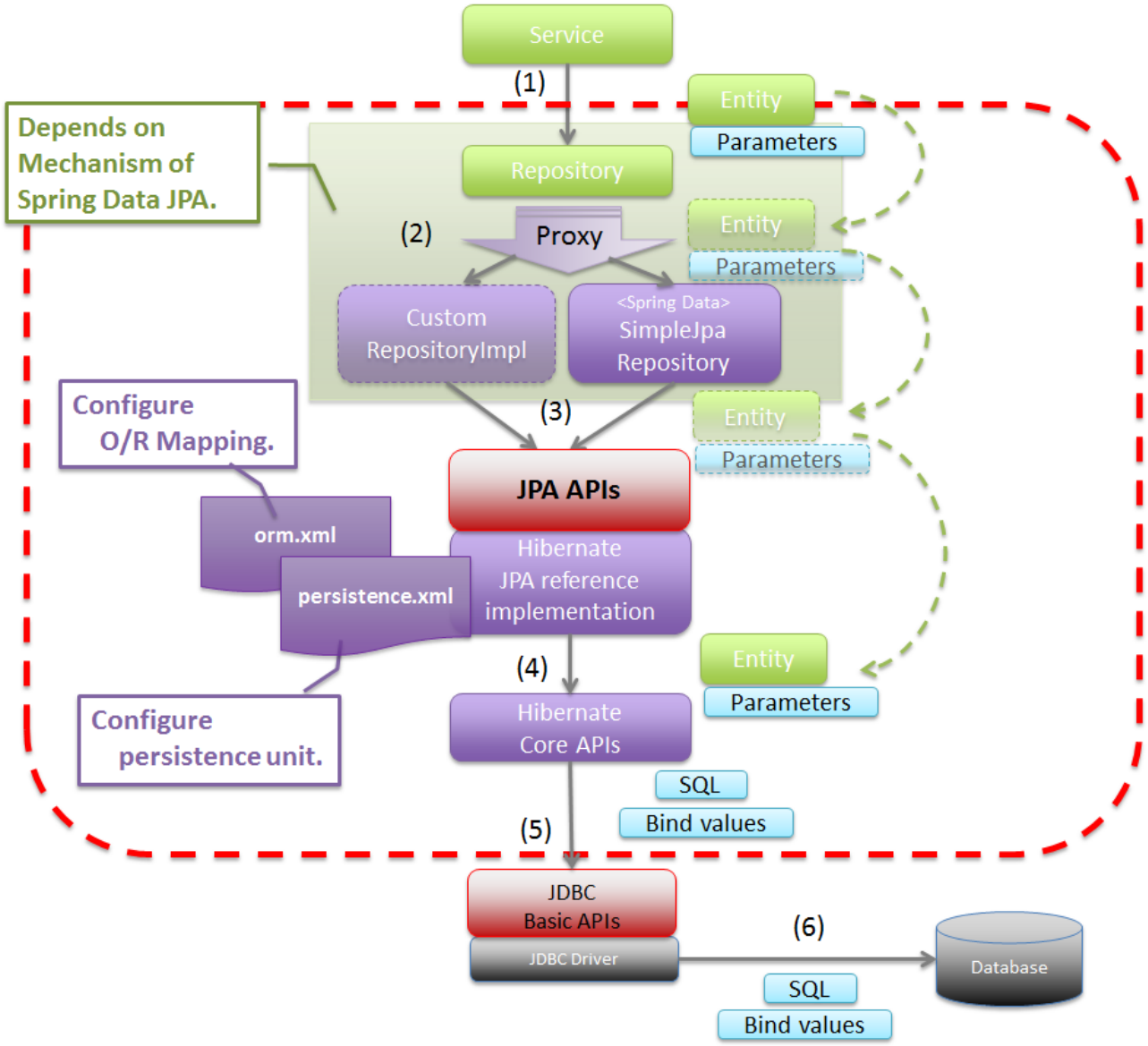
- Managing data between java classes or objects and the relational database is a very cumbersome and tricky task.
- The DAO (Data Access Object) layer usually contains a lot of **boilerplate code** that should be simplified in order to reduce the number of lines of code and make the code reusable.
- Spring Data JPA:
  - This provides spring data repository interfaces which are implemented to create JPA repositories.
  - Spring Data JPA provides a solution to reduce a lot of boilerplate code.
  - Spring Data JPA provides an **out-of-the-box** implementation for all the required CRUD operations for the JPA entity so we don't have to write the same boilerplate code **again and again**.

```
public class CustomerRepository {
    private static final int PAGE_SIZE = 10;
    private EntityManager getEntityManager()
    public List<Customer> findAll() {..}
    public void save(Customer c) {..}
    public Product find(Integer cid) {..}
}
```

```
public class ProductRepository {
    private static final int PAGE_SIZE = 50;
    private EntityManager getEntityManager()
    public List<Product> findAll() {..}
    public void save(Product p) {..}
    public Product find(String pid) {..}
}
```

Made with Goodnotes

Basic Spring Data JPA Flow



Made with Goodnotes

JPA Repository Example

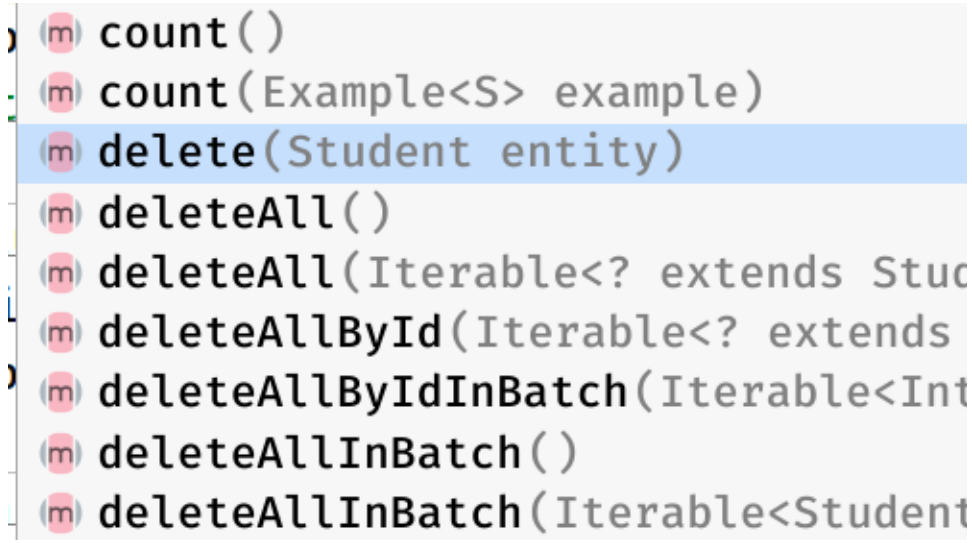
```java
@Getter @Setter @NoArgsConstructor
@AllArgsConstructor @ToString
@Entity
public class Student {
    @Id
    private Integer id;
    private String name;
    private Double gpax;
}
```

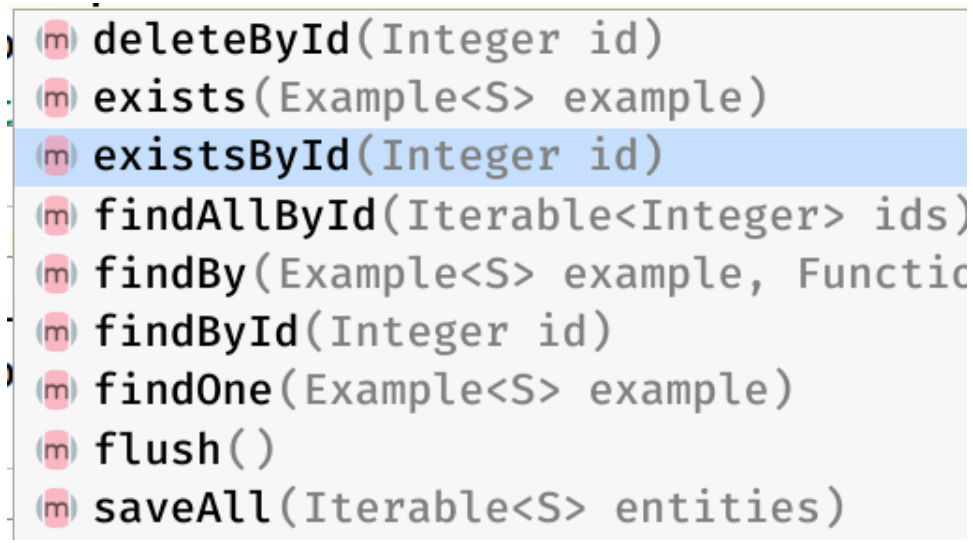
```java
import org.springframework.data.jpa.repository.JpaRepository;
import sit.int204.demo.entities.Student;

public interface StudentRepository extends JpaRepository<Student, Integer> {
    List<Student> findByNameContainsOrGpaxBetweenOrderByGpaxDesc(
            String name, double low, double high);
}
```

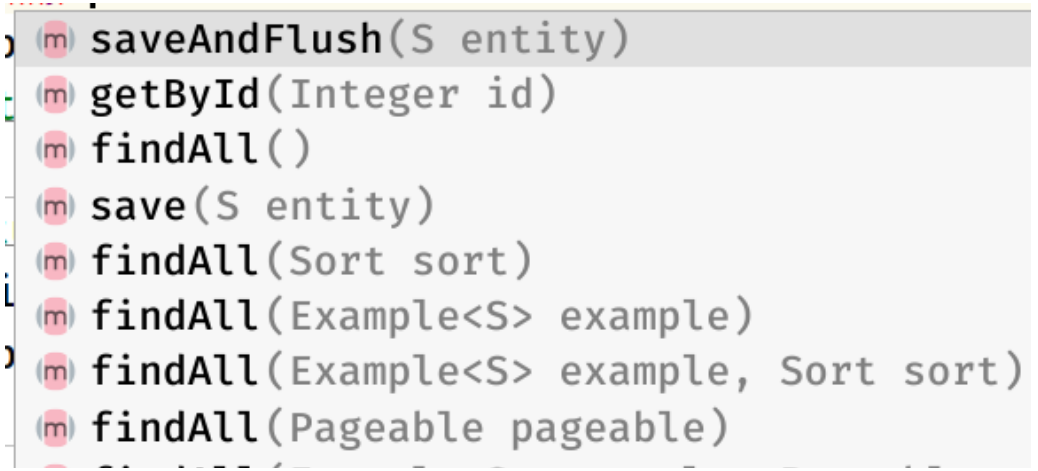
Made with Goodnotes

Jpa Repository default methods

```
public class AppController {
    @Autowired
    private final StudentRepository
studentRepository;
```







Made with Goodnotes

Spring Boot
+
Introduction to
Spring Framework Annotations & Spring Data JPA

By

Pichet Limvajiranan

Made with Goodnotes

Spring Framework Annotations

- Basically, there are 6 types of annotation available in the whole spring framework.

Presentation layer → spring web

1. Spring Web Annotations
2. Spring Core Annotations
3. Spring Boot Annotations
4. Spring Scheduling Annotations
5. Spring Data Annotations
6. Spring Bean Annotations

Made with Goodnotes

1) Spring Web Annotations

- Present in the org.springframework.web.bind.annotation
- Some of the annotations that are available in this category are:
  - **@RequestMapping**
  - **@RequestBody**
  - **@PathVariable**
  - @RequestParam
  - Response Handling Annotations
    - @ResponseBody @ExceptionHandler @ResponseStatus
  - @Controller
  - **@RestController**
  - @ModelAttribute
  - @CrossOrigin

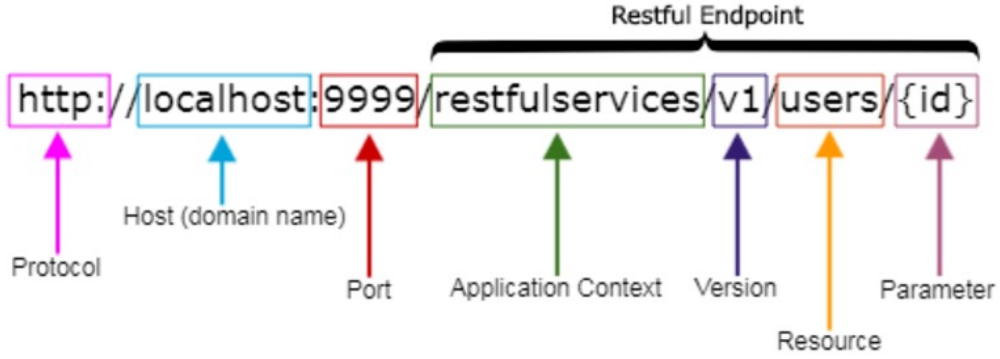
Made with Goodnotes

Spring Web Annotation example

```
@RestController
@RequestMapping("/api/offices")
public class OfficeController {
    :
    :
   @GetMapping("/{officeCode}")
   public Office getOfficeById(@PathVariable String officeCode) {
      return service.getOffice(officeCode);
   }


   @PostMapping("")
   public Office addNewOffice(@RequestBody Office office) {
       return service.createNewOffice(office);
  }
```

Made with Goodnotes

REST API URI Naming Conventions and Best Practices



- Singleton and Collection Resources

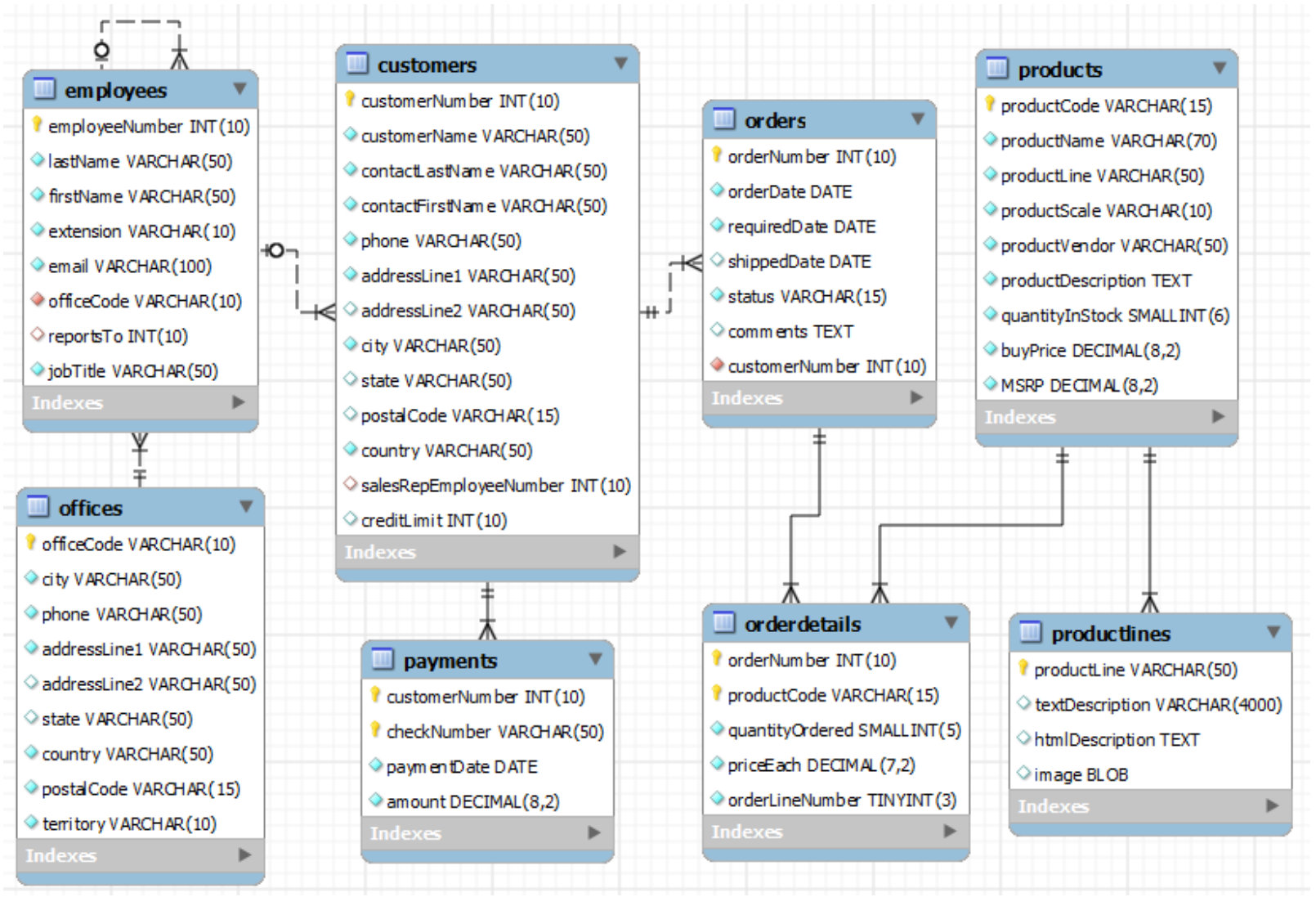
```
/customers                  // is a collection resource
/customers/{id}             // is a singleton resource
```

- Sub-collection Resources

```
/customers/{id}/orders                  // is a sub-collection resource
```

- Best Practices
  - https://medium.com/@nadinCodeHat/rest-api-naming-conventions-and-best-practices-1c4e781eb6a5
  - https://restfulapi.net/resource-naming

Made with Goodnotes



Excercise

Design REST Resources representation for the Classicmodels

Made with Goodnotes

2) Spring Core annotations

- Spring Core annotations are present in the 2 packages
  - org.springframework.beans.factory.annotation
  - org.springframework.context.annotation
- We can divide them into two broad categories:
  - **DI-Related Annotations**
    - **@Autowired** @Qualifier @Primary
    - @Bean @Lazy @Required
    - @Value @Scope
    - @Lookup, etc.
  - Context Configuration Annotations
    - @Profile @Import
    - @ImportResource @PropertySource, etc.

Made with Goodnotes

@Autowired

- We use @Autowired to mark the dependency that will be injected by the Spring container.
- Applied to the fields, setter methods, and constructors. It injects object dependency implicitly.

```
public class OfficeService {
    @Autowired
    private OfficeRepository repository;
```

```
public class OfficeService {
    private final OfficeRepository repository;
    @Autowired
    public OfficeService(OfficeRepository repository) {
        this.repository = repository;
    }
```

```
public class OfficeService {
    private final OfficeRepository repository;
    @Autowired
    public void setRepository(OfficeRepository repository) {
        this.repository = repository;
    }
```

Made with Goodnotes

3) Spring Boot Annotations

- @SpringBootApplication Combination of three annotations
  - @EnableAutoConfiguration
  - @ComponentScan
  - @Configuration.

```
@SpringBootApplication
public class ClassicmodelServiceApplication {
    public static void main(String[] args) {
        SpringApplication.run(ClassicmodelServiceApplication.class, args);
    }
}
```

Made with Goodnotes

5) Spring Data Annotations

Spring Data provides an abstraction over data storage.

- Common Spring Data Annotations
  - **@Transactional**
  - @NoRepositoryBean
  - @Param
  - **@Id**
  - @Transient
  - @CreatedBy, @LastModifiedBy, @CreatedDate, @LastModifiedDate

- Spring Data JPA Annotations
  - **@Query**
  - @Procedure
  - @Lock
  - @Modifying
  - @EnableJpaRepositories
- Spring Data Mongo Annotations
  - @Document
  - @Field
  - @Query
  - @EnableMongoRepositories

Made with Goodnotes

6) Spring Bean Annotations

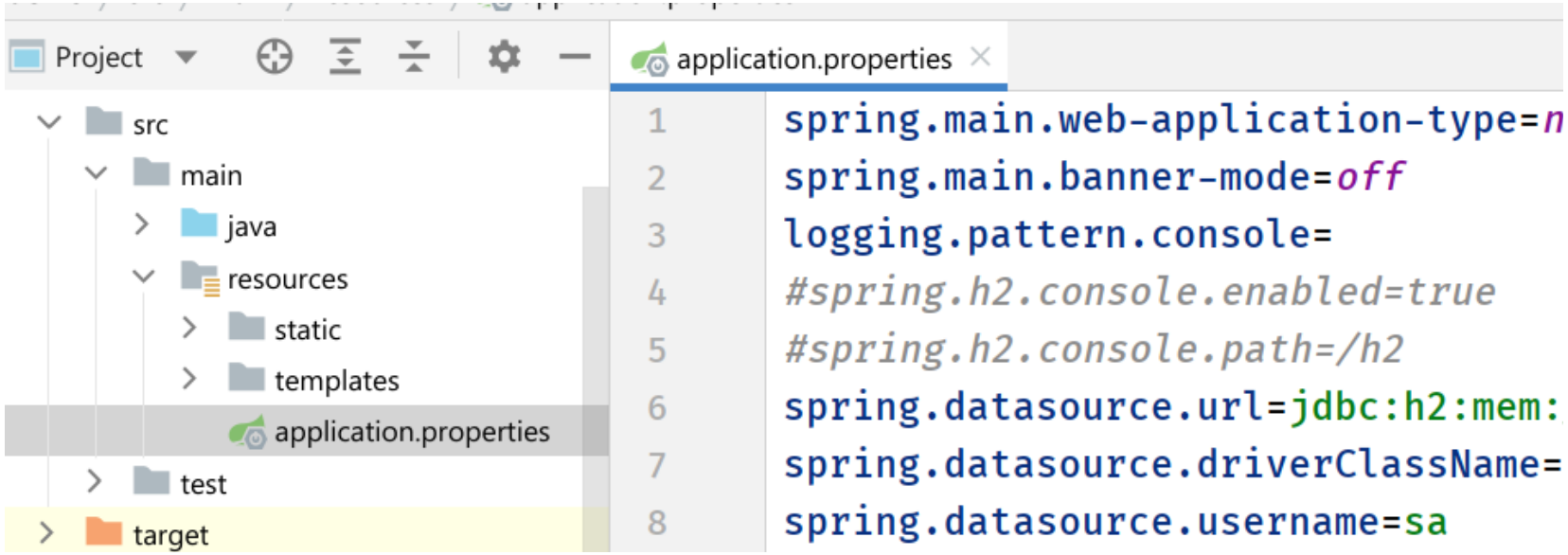
- Some of the annotations that are available in this category are:
  - @ComponentScan
  - @Configuration
  - Stereotype Annotations
    - @Component
      - **@Service**
      - **@Repository**
      - **@Controller**

@Service: We specify a class with @Service to indicate that they're holding the business logic. Besides being used in the service layer, there isn't any other special use for this annotation. The utility classes can be marked as Service classes.
@Repository: We specify a class with @Repository to indicate that they're dealing with **CRUD operations**, usually, it's used with DAO (Data Access Object) or Repository implementations that deal with database tables.
@Controller: We specify a class with @Controller to indicate that they're front controllers and responsible to handle user requests and return the appropriate response. It is mostly used with REST Web Services.

Made with Goodnotes

Spring Boot Application Properties

- Spring Boot Framework comes with a built-in mechanism for application configuration using a file called application.properties.
- It is located inside the src/main/resources folder.
- The properties have default values.
- We can set a property(s) for the Spring Boot application.



Made with Goodnotes

Spring Boot Application Properties - Examples

- spring.main.banner-mode
  - CONSOLE Print the banner to System.out.
  - LOG Print the banner to the log file.
  - OFF Disable printing of the banner.
- logging.level.<logger-name>=<level>
  - where level is one of TRACE, DEBUG, INFO, WARN, ERROR, FATAL, or OFF.
  - Example
    - logging.level.root=warn
    - logging.level.org.springframework.web=off
    - logging.level.org.hibernate=error
- server.error.include-stacktrace
  - ALWAYS Always add stacktrace information.
  - NEVER Never add stacktrace information.
  - ON_PARAM Add stacktrace attribute when the appropriate request parameter is not "false".

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

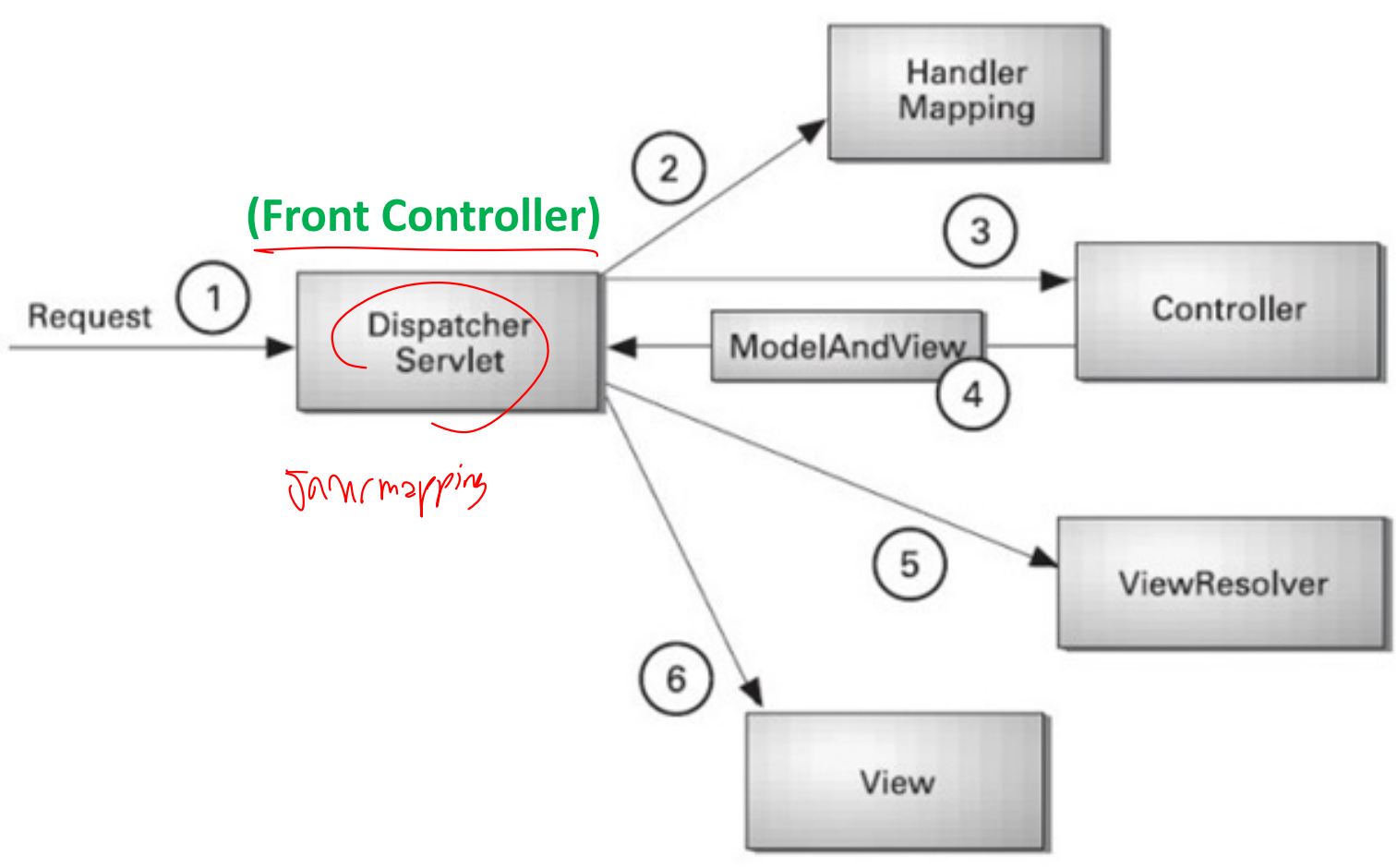
Made with Goodnotes

Spring Web MVC

- The Spring Web MVC framework provides Model-View-Controller (MVC) architecture and ready components that can be used to develop flexible and loosely coupled web applications.
- The MVC pattern results in separating the different aspects of the application (input logic, business logic, and UI logic), while providing a loose coupling between these elements.
- A Spring MVC provides an elegant solution to use MVC in spring framework by the help of DispatcherServlet.
    - In Spring Web MVC, the **DispatcherServlet** class works as the **front controller**. It is responsible to manage the flow of the Spring MVC application.

Made with Goodnotes

The DispatcherServlet and Flow of Spring Web MVC



Made with Goodnotes

Defining a Controller

- The DispatcherServlet delegates the request to the controllers to execute the functionality specific to it.
- The **@Controller** annotation indicates that a particular class serves the role of a controller.
- The @**RequestMapping** @**GetMapping** @**PostMapping** annotation is used to map a URL to either an entire class or a particular handler method.

```
@Controller
public class HelloController {
   @RequestMapping("/hello")
   public String printHello(ModelMap model) {
      model.addAttribute("message", "Hello Spring MVC Framework!");
      return "hello";
   }
}
```

Made with Goodnotes

Spring Boot Controller example

```java
@Controller
public class AppController {
    @Autowired
    private final StudentRepository studentRepository;

    @RequestMapping("/home")
    public String home() {
        return "home";
    }
    @GetMapping("/student-listing")
    public String students(Model model) {
        model.addAttribute("students", studentRepository.findAll());
        return "student-list";
    }
    @GetMapping("/student-list-plain-text")
    public ResponseEntity<String> students_list(Model model) {
        return new ResponseEntity<>(studentRepository.findAll()
                .toString(), HttpStatus.OK);
    }
```

Made with Goodnotes

Spring View Technology

- The Spring web framework is built around the MVC (Model-View-Controller) pattern, which makes it easier to separate concerns in an application.
- This allows for the possibility to use different view technologies, from the well established JSP technology to a variety of template engines.
  - Java Server Pages
  - Thymeleaf
  - FreeMarker
  - Groovy Markup Template Engine

Made with Goodnotes

Thymeleaf Template Engine example

```
<!DOCTYPE html>
<html lang="en“  xmlns=http://www.w3.org/1999/xhtml
          xmlns:th="http://www.thymeleaf.org">
<body>
<div class="container p4 m4">
    <h2>Student List:</h2><hr>
    <div class="row">
        <div class="col-2">Student Id</div>
        <div class="col-4">Name</div>
        <div class="col-2">GPAX</div>
    </div>
    <div class="row" th:each="student : ${students}">
        <div class="col-2" th:text="${student.id}"/>
        <div class="col-4" th:text="${student.name}"/>
        <div class="col-2" th:text="${student.gpax}"/>
    </div>
</div>
```

Made with Goodnotes

Spring Data JPA

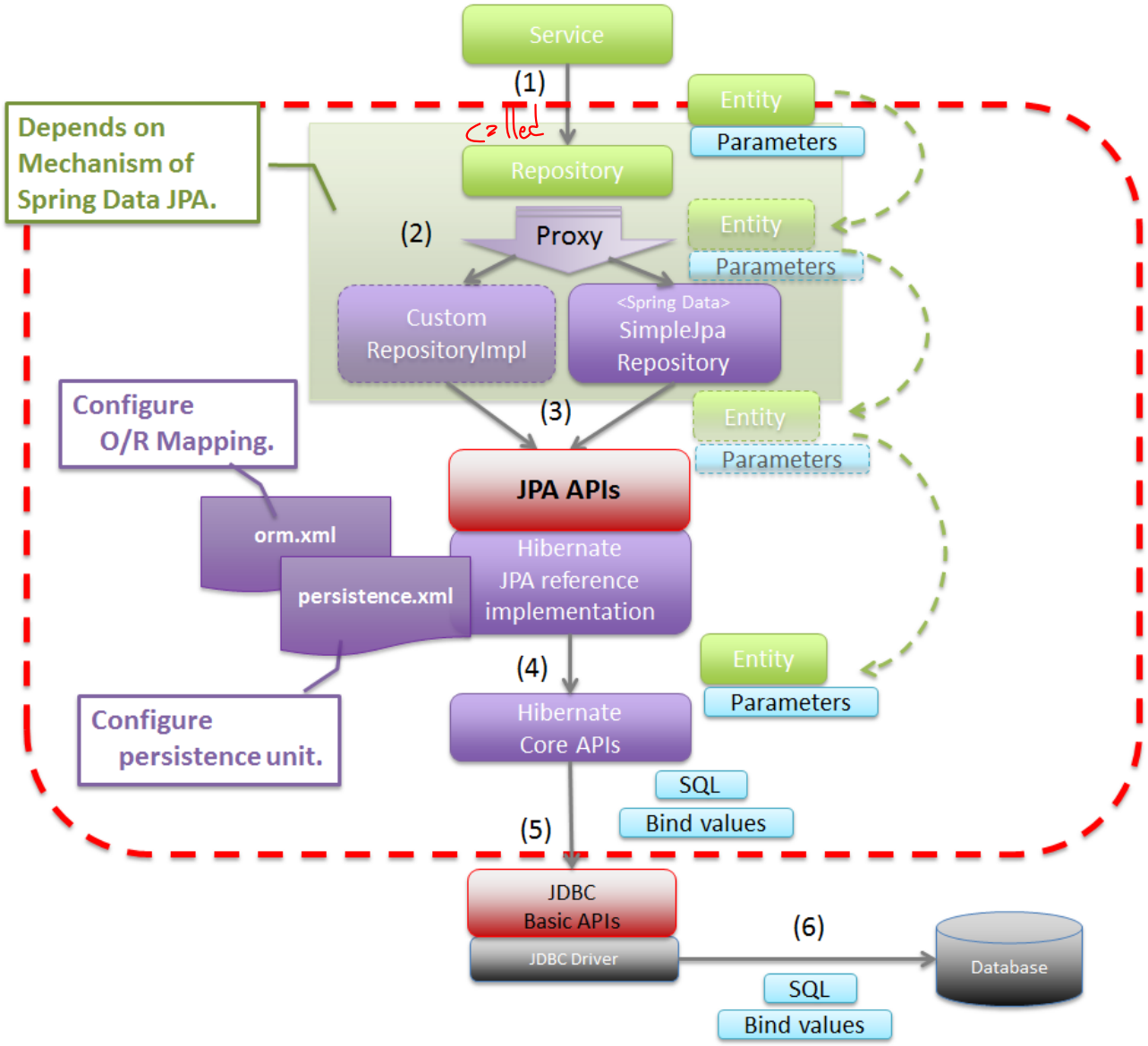
- Managing data between java classes or objects and the relational database is a very cumbersome and tricky task.
- The DAO (Data Access Object) layer usually contains a lot of **boilerplate code** that should be simplified in order to reduce the number of lines of code and make the code reusable.
- Spring Data JPA:
  - This provides spring data repository interfaces which are implemented to create JPA repositories.
  - Spring Data JPA provides a solution to reduce a lot of boilerplate code.
  - Spring Data JPA provides an **out-of-the-box** implementation for all the required CRUD operations for the JPA entity so we don't have to write the same boilerplate code **again and again**.

```
public class CustomerRepository {
    private static final int PAGE_SIZE = 10;
    private EntityManager getEntityManager()
    public List<Customer> findAll() {..}
    public void save(Customer c) {..}
    public Product find(Integer cid) {..}
}
```

```
public class ProductRepository {
    private static final int PAGE_SIZE = 50;
    private EntityManager getEntityManager()
    public List<Product> findAll() {..}
    public void save(Product p) {..}
    public Product find(String pid) {..}
}
```

Made with Goodnotes

Basic Spring Data JPA Flow



Made with Goodnotes

JPA Repository Example

```java
@Getter @Setter @NoArgsConstructor
@AllArgsConstructor @ToString
@Entity
public class Student {
    @Id
    private Integer id;
    private String name;
    private Double gpax;
}
```

```java
import org.springframework.data.jpa.repository.JpaRepository;
import sit.int204.demo.entities.Student;

public interface StudentRepository extends JpaRepository<Student, Integer> {
    List<Student> findByNameContainsOrGpaxBetweenOrderByGpaxDesc(
            String name, double low, double high);
}
```

class

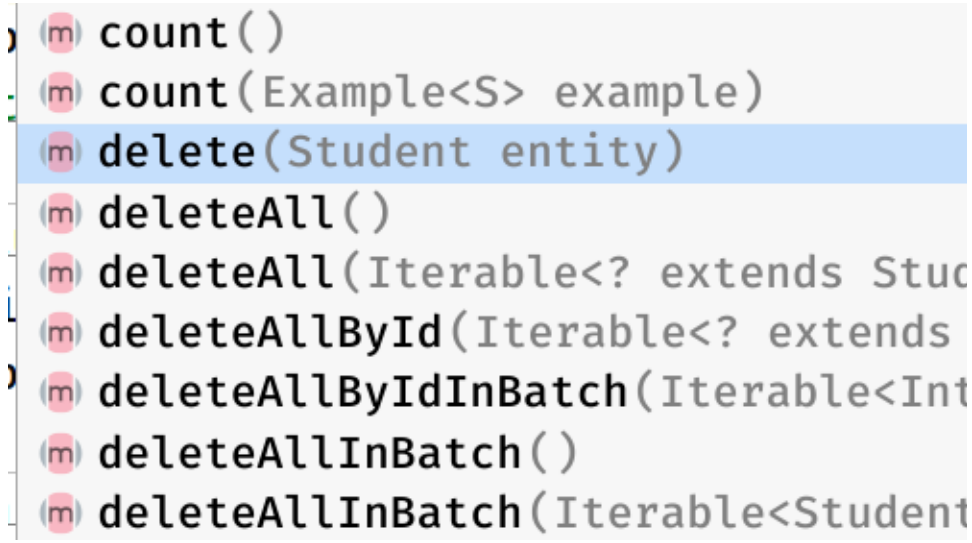
pk

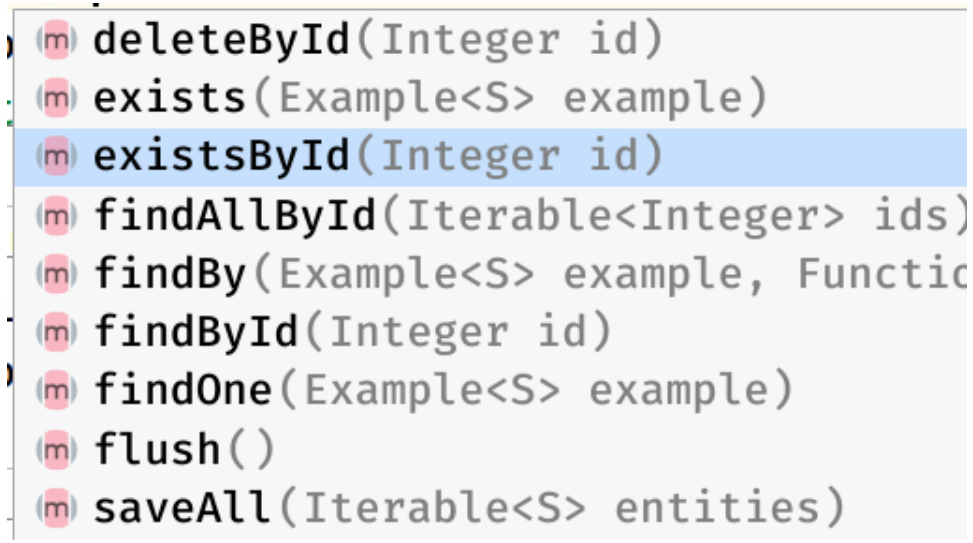
Query methods

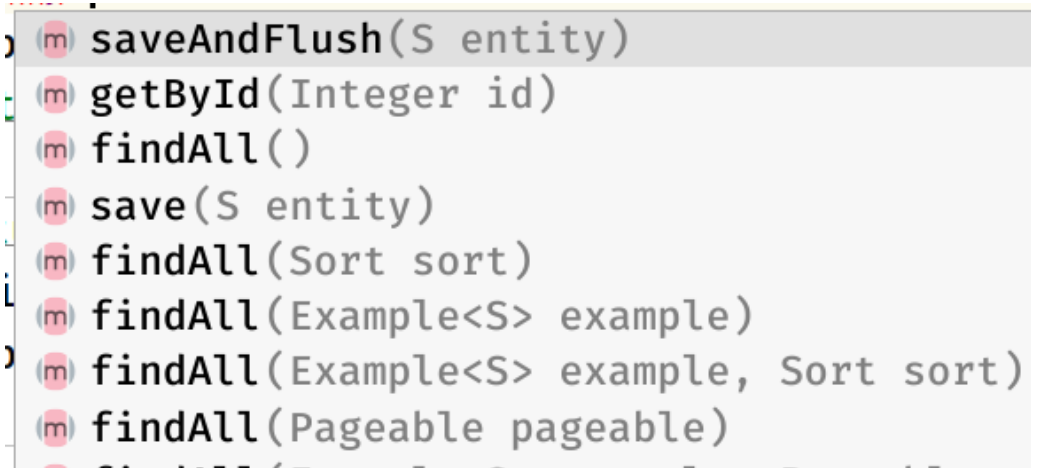
Made with Goodnotes

Jpa Repository default methods

```
public class AppController {
    @Autowired
    private final StudentRepository
studentRepository;
```







Made with Goodnotes

Example: Data Jpa - count()

OfficeController

```
@GetMapping("/count")
public Count getOfficesCount() {
   return new Count(service.getOfficeCount());
}
```

OfficeService

```
public Long getOfficeCount() {
   return repository.count();
}
```

```
@Getter
@Setter
class Count {
   private long count;
   public Count(long n) {
      this.count = n;
   }
}
```

Made with Goodnotes

Exercise: Create REST API service for resources below

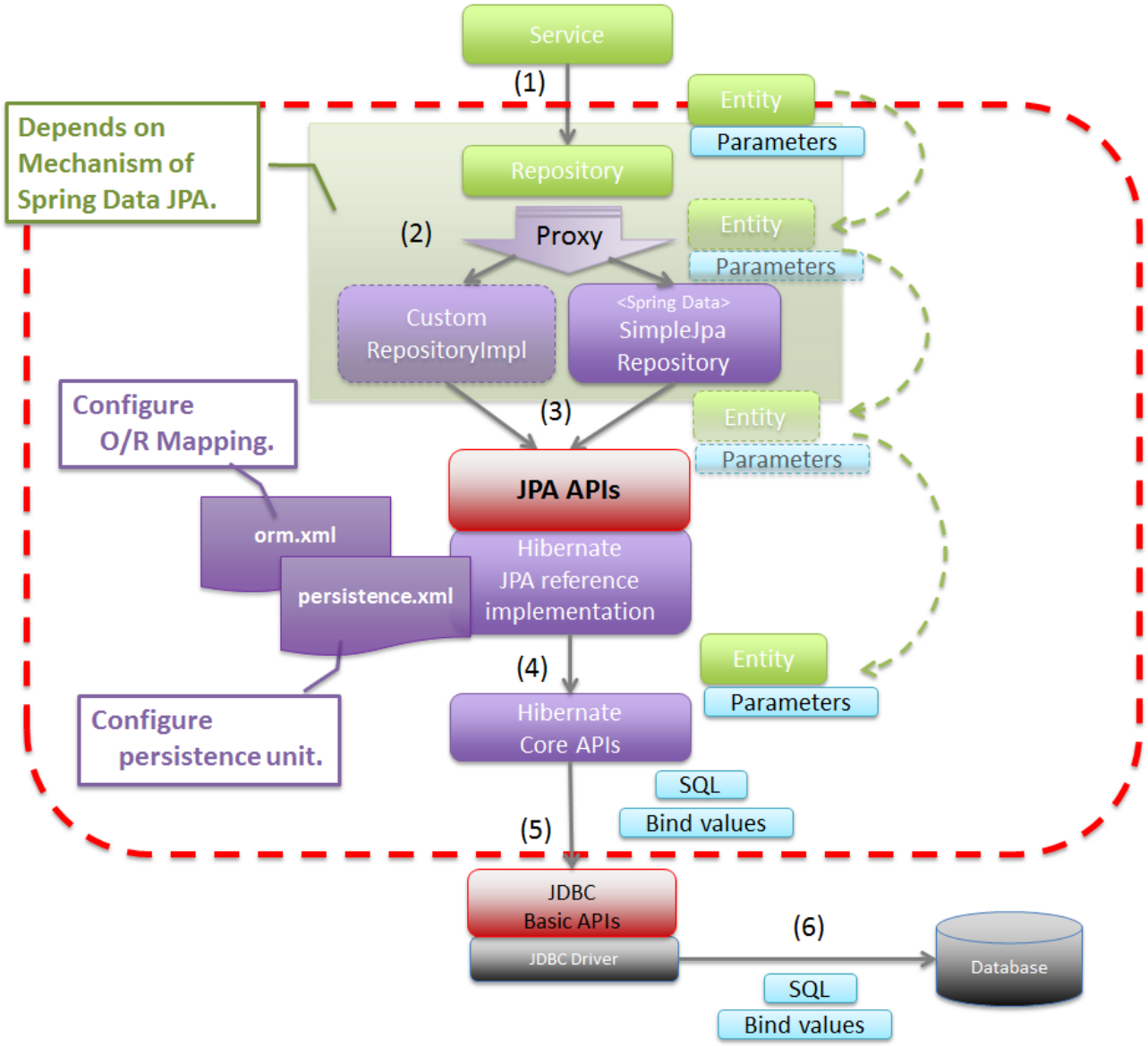
URI	HTTP Verb	Description
/customers	**GET**	**Get all customers**
/customers/{id}	**GET**	**Get a customer with id**
/customers/{id}/orders	**GET**	**Get all orders for customer id**
/customers	POST	Add new customers
/customers/{id}	PUT	Update a customers with id
/customers/{id}	DELETE	Delete a customers with id

Made with Goodnotes

Spring Data JPA
Query Creation

Made with Goodnotes

Basic Spring Data JPA Flow

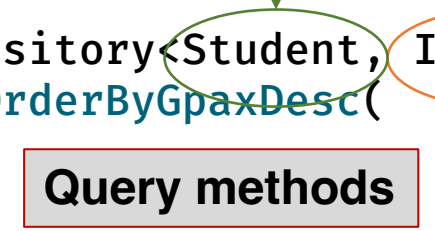


Made with Goodnotes

JPA Repository Example

```java
@Getter @Setter @NoArgsConstructor
@AllArgsConstructor @ToString
@Entity
public class Student {
    @Id
    private Integer id;
    private String name;
    private Double gpax;
}
```

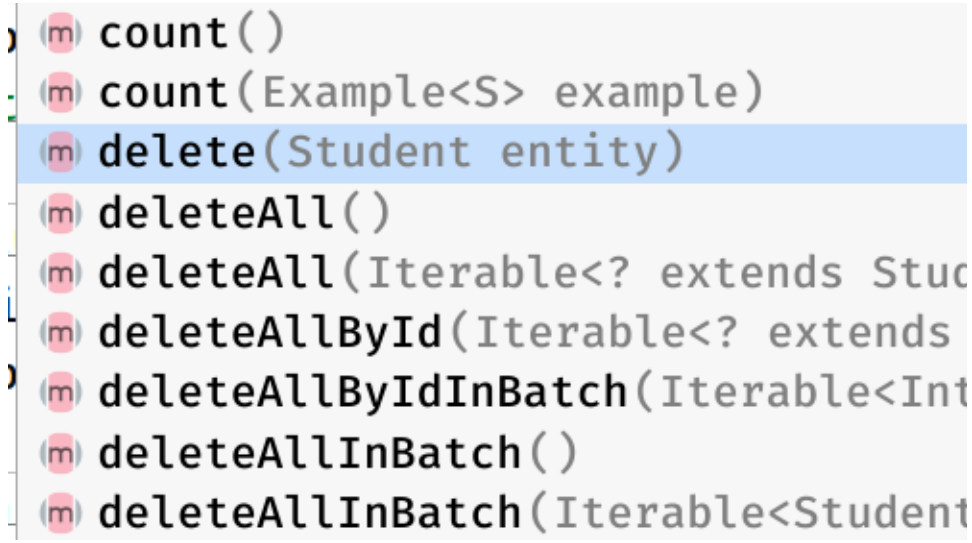
```java
import org.springframework.data.jpa.repository.JpaRepository;
import sit.int204.demo.entities.Student;

public interface StudentRepository extends JpaRepository<Student, Integer> {
    List<Student> findByNameContainsOrGpaxBetweenOrderByGpaxDesc(
            String name, double low, double high);
}
```

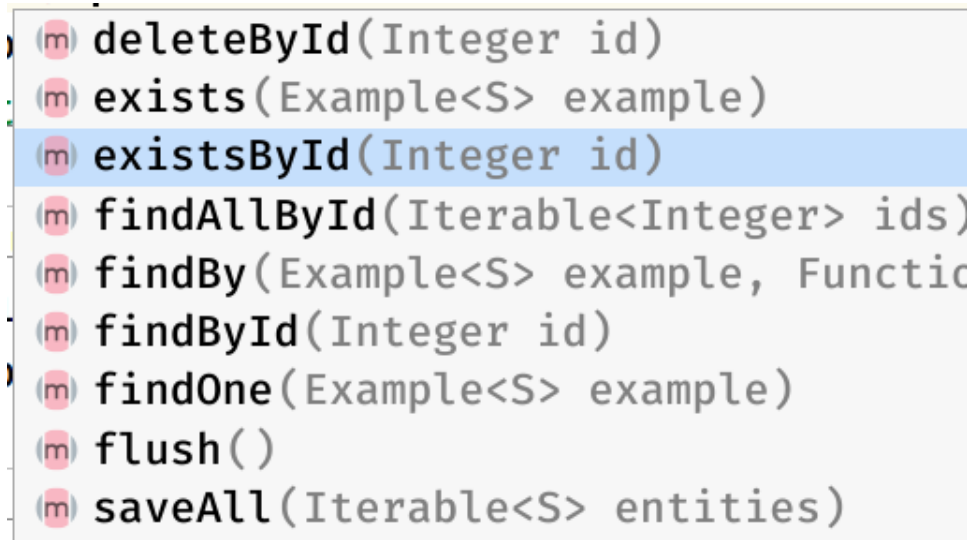


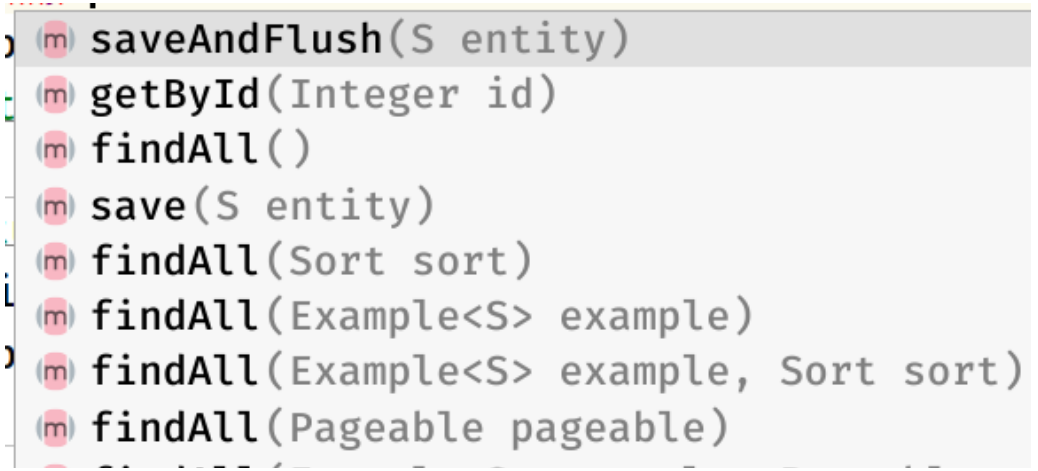
Made with Goodnotes

Jpa Repository default methods

```
public class AppController {
    @Autowired
    private final StudentRepository
studentRepository;
```







Made with Goodnotes

Examples & Exercises: saveAll(Iterable<S> entities)

```java
@Entity
@Data
@AllArgsConstructor
@NoArgsConstructor
public class Customer {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String firstName;
    private String lastName;
}
```

```java
public interface CustomerRepository extends JpaRepository<Customer, Long> { }
```

```java
@Service
public class CustomerService {
     @Autowired CustomerRepository customerRepository;
     public List<Customer> insertCustomers(List<Customer> customers) {
        return customerRepository.saveAll(customers);
     }
}
```

Made with Goodnotes

Query Creation

- Generally, the query creation mechanism for JPA works as described in “Query Methods”. The following example shows what a JPA query method translates into:
- Example: Query creation from method names

```
public interface UserRepository extends Repository<User, Long> {
  List<User> findByEmailAddressAndLastname(String emailAddress, String lastname);
}
```

- We create a query using the JPA criteria API from this, but, essentially, this translates into the following query:

  select u from User u where u.emailAddress = ?1 and u.lastname = ?2.

- Spring Data JPA does a property check and traverses nested properties, as described in “Property Expressions”.

Made with Goodnotes

Supported keywords inside method names

Keyword	Sample	JPQL snippet
Distinct	findDistinctByLastnameAndFirstname	select distinct … where x.lastname = ?1 and x.firstname = ?2
And	findByLastnameAndFirstname	… where x.lastname = ?1 and x.firstname = ?2
Or	findByLastnameOrFirstname	… where x.lastname = ?1 or x.firstname = ?2
Is, Equals	findByFirstname,findByFirstnameIs,findByFirstnameEquals	… where x.firstname = ?1
Between	findByStartDateBetween	… where x.startDate between ?1 and ?2
LessThan	findByAgeLessThan	… where x.age < ?1
LessThanEqual	findByAgeLessThanEqual	… where x.age <= ?1

Made with Goodnotes

Supported keywords inside method names (2)

Keyword	Sample	JPQL snippet
GreaterThan	findByAgeGreaterThan	… where x.age > ?1
GreaterThanEqual	findByAgeGreaterThanEqual	… where x.age >= ?1
After	findByStartDateAfter	… where x.startDate > ?1
Before	findByStartDateBefore	… where x.startDate < ?1
IsNull, Null	findByAge(Is)Null	… where x.age is null
IsNotNull, NotNull	findByAge(Is)NotNull	… where x.age not null
Like	findByFirstnameLike	… where x.firstname like ?1
NotLike	findByFirstnameNotLike	… where x.firstname not like ?1

Made with Goodnotes

Supported keywords inside method names (3)

Keyword	Sample	JPQL snippet
StartingWith	findByFirstnameStartingWith	… where x.firstname like ?1 (parameter bound with appended %)
EndingWith	findByFirstnameEndingWith	… where x.firstname like ?1 (parameter bound with prepended %)
Containing	findByFirstnameContaining	… where x.firstname like ?1 (parameter bound wrapped in %)
OrderBy	findByAgeOrderByLastnameDesc	… where x.age = ?1 order by x.lastname desc
NotIn	findByAgeNotIn(Collection<Age> ages)	… where x.age not in ?1
True	findByActiveTrue()	… where x.active = true
False	findByActiveFalse()	… where x.active = false

Made with Goodnotes

Query Method Example

```
public interface CustomerRepository extends JpaRepository<Customer, Integer> {
    public List<Customer> findAllByCustomerNameContaining(String name);
    public List<Customer> findAllByCityContainsOrderByCountry(String name);
    public List<Customer> findAllByCreditLimitBetween(Double lower, Double upper);
    public List<Customer> findAllByCustomerNameBetween(String lower, String upper);
}
```

Made with Goodnotes

Exercise 1:

- Create REST API service for Products as end-points below

URI	HTTP verb	Description
/products	GET	Get all products filter by price between and product name contains
/products/product-line/{id}	GET	Get products by product line
/products/{id}	GET	Get product by product id
/products	POST	Add new product
/products/{id}	PUT	Update product

Made with Goodnotes

JPA Named Queries

- Using named queries to declare queries for entities is a valid approach and works fine for a small number of queries.
- As the queries themselves are tied to the Java method that runs them, you can actually bind them directly by using the Spring Data JPA @Query annotation rather than annotating them to the domain class.
- This frees the domain class from persistence specific information and co-locates the query to the repository interface.

```
public interface UserRepository extends JpaRepository<User, Long> {
   @Query('select u from User u where u.emailAddress = ?1")
   User findByEmailAddress(String emailAddress);
}
```

Made with Goodnotes

Native Queries

- The @Query annotation allows for running native queries by setting the nativeQuery flag to true, as shown in the following example:

- Declare a native query at the query method using @Query

```
public interface UserRepository extends JpaRepository<User, Long> {
    @Query(value = "SELECT * FROM USERS WHERE EMAIL_ADDRESS = ?1", nativeQuery = true)
    User findByEmailAddress(String emailAddress);
}
```

Made with Goodnotes

Exercise 2:

- Using Name Query instead Query Method for REST service in exercise 1

URI	HTTP verb	Description
/products	GET	Get all products filter by price between and product name contains
/products/line/{product-line}	GET	Get products by product line
/products/{id}	GET	Get product by product id
/products	POST	Add new product
/products/{id}	PUT	Update product

Made with Goodnotes

Spring Data REST: Pagination and Sorting

- The PagingAndSortingRepository is an extension of CrudRepository to provide additional methods to retrieve entities using the pagination and sorting abstraction. It implicitly provides two methods:
  - **Page<T> findAll(Pageable pageable)**
    returns a Page of entities meeting the paging restriction provided in the Pageable object.

    ```
    Pageable firstPageTwoElements = PageRequest.of(0, 2); Pageable
    secondPageFiveElements = PageRequest.of(1, 5);
    ```

  - **Iterable<T> findAll(Sort sort)**
    returns all entities sorted by the given options. No paging is applied here.

    ```
    Sort sortedByName = Sort.by("name");
    ```

  - Pagination & Sorting

    ```
    Pageable sortedByPriceDescNameAsc = PageRequest.of(0, 5,
    Sort.by("price").descending().and(Sort.by("name")));
    ```

Made with Goodnotes

Spring Data Sort and Order

- The Sort class provides sorting options for database queries with more flexibility in choosing single/multiple sort columns and directions (ascending/descending).
  - we use by(), descending(), and() methods to create Sort object and pass it to Repository.findAll()
- You can sort results by Sort and Order object with one or more specified variables.
- Sorting can be done in ascending or descending order.

```
@Service
 :
 :
public List<Customer> getAllCustomers(String sortBy) {
   return repository.findAll(Sort.Direction.DESC, Sort.by(sortBy));
}
```

Made with Goodnotes

Sort & Order object example

```
// order by 'published' column - ascending
List<Tutorial> tutorials = tutorialRepository.findAll(Sort.by("published"));

// order by 'published' column, descending
tutorialRepository.findAll(Sort.by("published").descending());

// order by 'published' column - descending, then order by 'title' - ascending
tutorialRepository.findAll(Sort.by("published").descending().and(Sort.by("title")));
```

```
List<Sort.Order> orders = new ArrayList();
Sort.Order order1 = new Sort.Order(Sort.Direction.DESC, "published");
orders.add(order1);
Sort.Order order2 = new Sort.Order(Sort.Direction.ASC, "title");
orders.add(order2);

List<Tutorial> tutorials = tutorialRepository.findAll(Sort.by(orders));
```

Made with Goodnotes

JpaRepository with Pagination

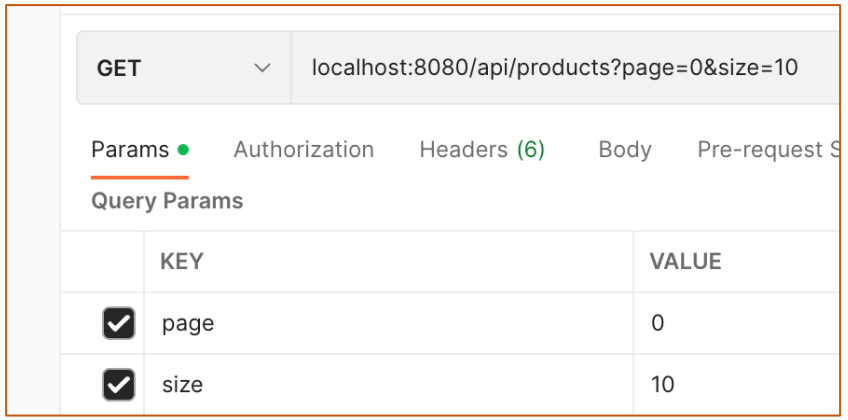
- findAll(Pageable pageable): returns a Page of entities meeting the paging condition provided by Pageable object.
- Pagination can be added by creation of PageRequest object which is implementation of Pageable interface.
- Similar to sorting adding pagination depends from type of Repository extended by our interface.

```
@Service
  :
public Page<Customer> getAllCustomers(int page, int pageSize) {
        Pageable pageable = PageRequest.of(page, pageSize);
        return repository.findAll(pageable);
}
```

Made with Goodnotes

Accepting Page and Sort Parameters

- Generally, paging and sorting parameters are optional and thus part of the request URL as query parameters. If any API supports paging and sorting, ALWAYS provide default values to these parameters – to be used when the client does not choose to specify any paging or sorting preferences.
- Example:




```
@GetMapping("")
    public List<Customer> getAllCustomers(
            @RequestParam(defaultValue = "id") String sortBy,
            @RequestParam(defaultValue = "0") Integer page,
            @RequestParam(defaultValue = "10") Integer pageSize) {
             Page<Customer> customers = service.findAll(sortBy, page, pageSize);
            return customers.getContent();
        }
```

Made with Goodnotes

Page<T> Object


```
{
  "content": [
    {
      "id": 323,
      "customerName": "Down Under Souveniers, Inc",
      "contactLastName": "Graham",
      "contactFirstName": "Mike",
      "phone": "+64 9 312 5555",
      "addressLine1": "162-164 Grafton Road",
      "addressLine2": "Level 2",
      :
  ]
  "pageable": {
    "sort": {
      "empty": false,
      "sorted": true,
      "unsorted": false
    },
      "offset": 5,
      "pageSize": 5,
      "pageNumber": 1,
      "unpaged": false,
      "paged": true
  },
  "last": false,
  "totalPages": 25,
  "totalElements": 122,
  "size": 5,
  "number": 1,
  "sort": {
    "empty": false,
    "sorted": true,
    "unsorted": false
  },
  "numberOfElements": 5,
  "first": false,
  "empty": false
}
```


Made with Goodnotes

Service - Paging & Sorting

```
@Service
public class CustomerService {
   @Autowired
   private CustomerRepository repository;

    public Page<Customer> getAllCustomers(
            String sortBy, int page, int pageSize) {
      Pageable pageble = PageRequest.of(page, pageSize);
      Page<Customer> customers = repository.findAll(pageble);
      return customers;
   }
}
```

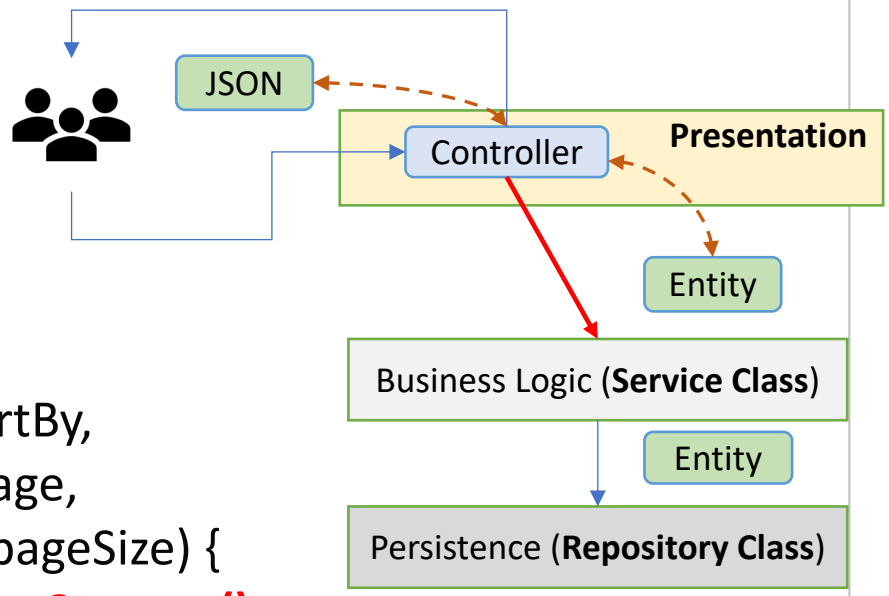
```
{
 "content " : {
      { },
      { },
      { }
  },
  "page": {
    "size": 20,
    "totalElements": 11,
    "totalPages": 1,
    "number": 0
  }
}
```

Made with Goodnotes

Controller - Paging & Sorting

```
@RestController
@RequestMapping("/api/customers")
public class CustomerController {
   @Autowired
   private CustomerService service;

   @GetMapping("")
   public List<Customer> getAllCustomers(
         @RequestParam(defaultValue = "id") String sortBy,
         @RequestParam(defaultValue = "0") Integer page,
         @RequestParam(defaultValue = "10") Integer pageSize) {
         return service.findAll(sortBy, page, pageSize).getContent();
   }
}
```

localhost:port/api/customers?sortBy=id&page=0&pageSize=10



Made with Goodnotes

Spring RESTful API Pagination & Sorting

By

Pichet Limvajiranan

Made with Goodnotes

Spring Data REST: Pagination and Sorting

- The PagingAndSortingRepository is an extension of CrudRepository to provide additional methods to retrieve entities using the pagination and sorting abstraction. It implicitly provides two methods:
  - **Page<T> findAll(Pageable pageable)**
    returns a Page of entities meeting the paging restriction provided in the Pageable object.

```
Pageable firstPageTwoElements = PageRequest.of(0, 2); Pageable
secondPageFiveElements = PageRequest.of(1, 5);
```

  - **Iterable<T> findAll(Sort sort)**
    returns all entities sorted by the given options. No paging is applied here.

```
Sort sortedByName = Sort.by("name");
```

  - Pagination & Sorting

```
Pageable sortedByPriceDescNameAsc = PageRequest.of(0, 5,
Sort.by("price").descending().and(Sort.by("name")));
```

Made with Goodnotes

Spring Data Sort and Order

- The Sort class provides sorting options for database queries with more flexibility in choosing single/multiple sort columns and directions (ascending/descending).
  - we use by(), descending(), and() methods to create Sort object and pass it to Repository.findAll()
- You can sort results by Sort and Order object with one or more specified variables.
- Sorting can be done in ascending or descending order.

```
@Service
 :
 :
public List<Customer> getAllCustomers(String sortBy) {
   return repository.findAll(Sort.Direction.DESC, Sort.by(sortBy));
}
```

Made with Goodnotes

Sort & Order object example

```
// order by 'published' column - ascending
List<Tutorial> tutorials = tutorialRepository.findAll(Sort.by("published"));

// order by 'published' column, descending
tutorialRepository.findAll(Sort.by("published").descending());

// order by 'published' column - descending, then order by 'title' - ascending
tutorialRepository.findAll(Sort.by("published").descending().and(Sort.by("title")));
```

```
List<Sort.Order> orders = new ArrayList();
Sort.Order order1 = new Sort.Order(Sort.Direction.DESC, "published");
orders.add(order1);
Sort.Order order2 = new Sort.Order(Sort.Direction.ASC, "title");
orders.add(order2);

List<Tutorial> tutorials = tutorialRepository.findAll(Sort.by(orders));
```

Made with Goodnotes

Exercise 1:

- Create REST API service for Products as end-points below

URI	HTTP verb	Description
/products	GET	Get all products filter by price between and product name contains sorting as request specify

Made with Goodnotes

```
public interface ProductRepository extends JpaRepository<Product, String> {
   List<Product> getProductsByPriceBetweenAndProductNameContains(
         Double lower, Double upper, String partOfName, Sort sort);
   Product findFirstByOrderByPriceDesc();
}
```

```
@Service
public class ProductService {
   @Autowired
   ProductRepository repository;
   public List<Product> getAllProducts(Double lower, Double upper,
                              String partOfName, String sortBy, String direction) {
      if (upper==0 && lower==0) { upper = repository.findFirstByOrderByPriceDesc().getPrice(); }
      if(sortBy.isEmpty()) { sortBy = "productCode" ; }
      Sort.Order sortOrder = new Sort.Order((direction.equalsIgnoreCase("asc")?
            Sort.Direction.ASC : Sort.Direction.DESC), sortBy);
      return repository.getProductsByPriceBetweenAndProductNameContains(
            lower, upper, partOfName, Sort.by(sortOrder));
   }
}
```

Made with Goodnotes

```java
@RestController
@RequestMapping("/products")
public class ProductController {
    @Autowired
    ProductService service;
    @GetMapping("")
    public List<Product> getAllProducts(
            @RequestParam(defaultValue = "") String partOfProductName,
            @RequestParam(defaultValue = "0") Double lower,
            @RequestParam(defaultValue = "0") Double upper,
            @RequestParam(defaultValue = "") String sortBy,
            @RequestParam(defaultValue = "ASC") String sortDirection
    ) {
        return service.getAllProducts(lower, upper, partOfProductName, sortBy, sortDirection);
    }
}
```

Made with Goodnotes

JpaRepository with Pagination

- findAll(Pageable pageable): returns a Page of entities meeting the paging condition provided by Pageable object.
- Pagination can be added by creation of PageRequest object which is implementation of Pageable interface.
- Similar to sorting adding pagination depends from type of Repository extended by our interface.

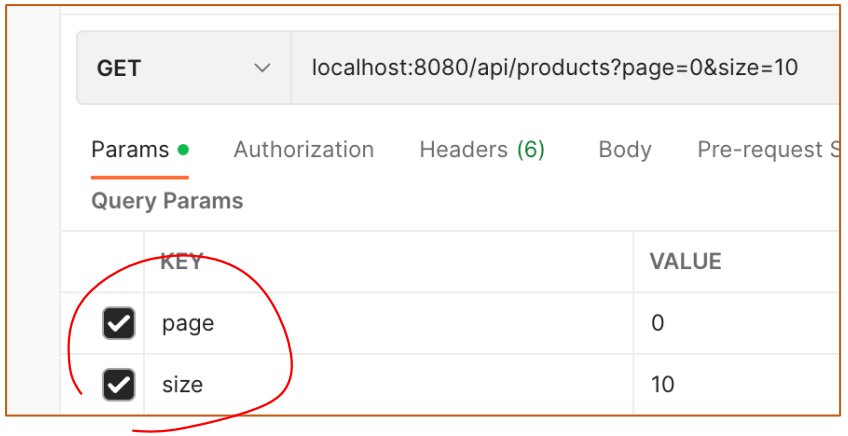
```
@Service
  :
public Page<Customer> getAllCustomers(int page, int pageSize) {
        Pageable pageable = PageRequest.of(page, pageSize);
        return repository.findAll(pageable);
}
```

Made with Goodnotes

Accepting Page and Sort Parameters

- Generally, paging and sorting parameters are optional and thus part of the request URL as query parameters. If any API supports paging and sorting, ALWAYS provide default values to these parameters – to be used when the client does not choose to specify any paging or sorting preferences.
- Example:

```
@GetMapping("")
  public List<Customer> getAllCustomers(
        @RequestParam(defaultValue = "id") String sortBy,
        @RequestParam(defaultValue = "0") Integer page,
        @RequestParam(defaultValue = "10") Integer pageSize) {
         Page<Customer> customers = service.findAll(sortBy, page, pageSize);
        return customers.getContent();
  }
```



Made with Goodnotes

Page<T> Object


```
{
  "content": [
    {
      "id": 323,
      "customerName": "Down Under Souveniers, Inc",
      "contactLastName": "Graham",
      "contactFirstName": "Mike",
      "phone": "+64 9 312 5555",
      "addressLine1": "162-164 Grafton Road",
      "addressLine2": "Level 2",
      :
  ]
  "pageable": {
    "sort": {
      "empty": false,
      "sorted": true,
      "unsorted": false
    },
      "offset": 5,
      "pageSize": 5,
      "pageNumber": 1,
      "unpaged": false,
      "paged": true
  },
  "last": false,
  "totalPages": 25,
  "totalElements": 122,
  "size": 5,
  "number": 1,
  "sort": {
    "empty": false,
    "sorted": true,
    "unsorted": false
  },
  "numberOfElements": 5,
  "first": false,
  "empty": false
}
```


Made with Goodnotes

Service - Paging & Sorting

```
@Service
public class CustomerService {
   @Autowired
   private CustomerRepository repository;

    public Page<Customer> getAllCustomers(
            String sortBy, int page, int pageSize) {
      Pageable pageble = PageRequest.of(page, pageSize);
      Page<Customer> customers = repository.findAll(pageble);
      return customers;
   }
}
```

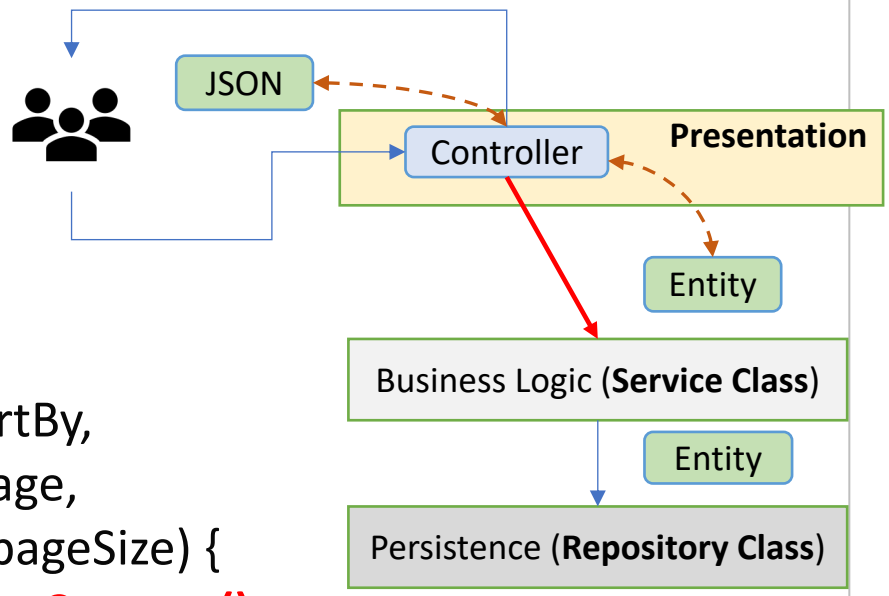
```
{
 "content " : {
      { },
      { },
      { }
  },
  "page": {
    "size": 20,
    "totalElements": 11,
    "totalPages": 1,
    "number": 0
  }
}
```

Made with Goodnotes

Controller - Paging & Sorting

```
@RestController
@RequestMapping("/api/customers")
public class CustomerController {
   @Autowired
   private CustomerService service;

   @GetMapping("")
   public List<Customer> getAllCustomers(
         @RequestParam(defaultValue = "id") String sortBy,
         @RequestParam(defaultValue = "0") Integer page,
         @RequestParam(defaultValue = "10") Integer pageSize) {
         return service.findAll(sortBy, page, pageSize).getContent();
   }
}
```

localhost:port/api/customers?sortBy=id&page=0&pageSize=10



Made with Goodnotes

Exercise 2:

- Create REST API service for Products as end-points below

URI	HTTP verb	Description
/products	GET	Get all products filter by price between and product name contains sorting as request specify with pagination

Made with Goodnotes

Add New Query Method to support pagination

```
public interface ProductRepository extends JpaRepository<Product, String> {
    List<Product> getProductsByPriceBetweenAndProductNameContains(
            Double lower, Double upper, String partOfName, Sort sort);
    Product findFirstByOrderByPriceDesc();

    Page<Product> getProductsByPriceBetweenAndProductNameContains(
            Double lower, Double upper, String partOfName, Pageable pageable);
}
```

Made with Goodnotes

Overload service method to support pagination

```
public Page<Product> getAllProducts(Double lower, Double upper,
                        String partOfName, String sortBy, String direction,
                        int pageNo, int pageSize) {
   if (upper == 0 && lower == 0) {
      upper = repository.findFirstByOrderByPriceDesc().getPrice();
   }
   if (sortBy.isEmpty()) {
      sortBy = "productCode";
   }
   Sort.Order sortOrder = new Sort.Order(
         (direction.equalsIgnoreCase("asc") ? Sort.Direction.ASC : Sort.Direction.DESC), sortBy);
   Pageable pageable = PageRequest.of(pageNo, pageSize, Sort.by(sortOrder));
   return repository.getProductsByPriceBetweenAndProductNameContains(lower, upper, partOfName, pageable);
}
```

Made with Goodnotes

Controller: Create new end-point/Modify method to support pagination

```
@RestController
@RequestMapping("/products")
public class ProductController {
   @Autowired
   ProductService service;
   @GetMapping("")
   public Page<Product> getAllProducts(
         @RequestParam(defaultValue = "") String partOfProductName,
         @RequestParam(defaultValue = "0") Double lower,
         @RequestParam(defaultValue = "0") Double upper,
         @RequestParam(defaultValue = "") String sortBy,
         @RequestParam(defaultValue = "ASC") String sortDirection,
         @RequestParam(defaultValue = ”0") int pageNo, @RequestParam(defaultValue = "10") int pageSize
   ) {
      return service.getAllProducts(lower, upper, partOfProductName, sortBy, sortDirection, pageNo, pageSize);
   }
}
```

Made with Goodnotes

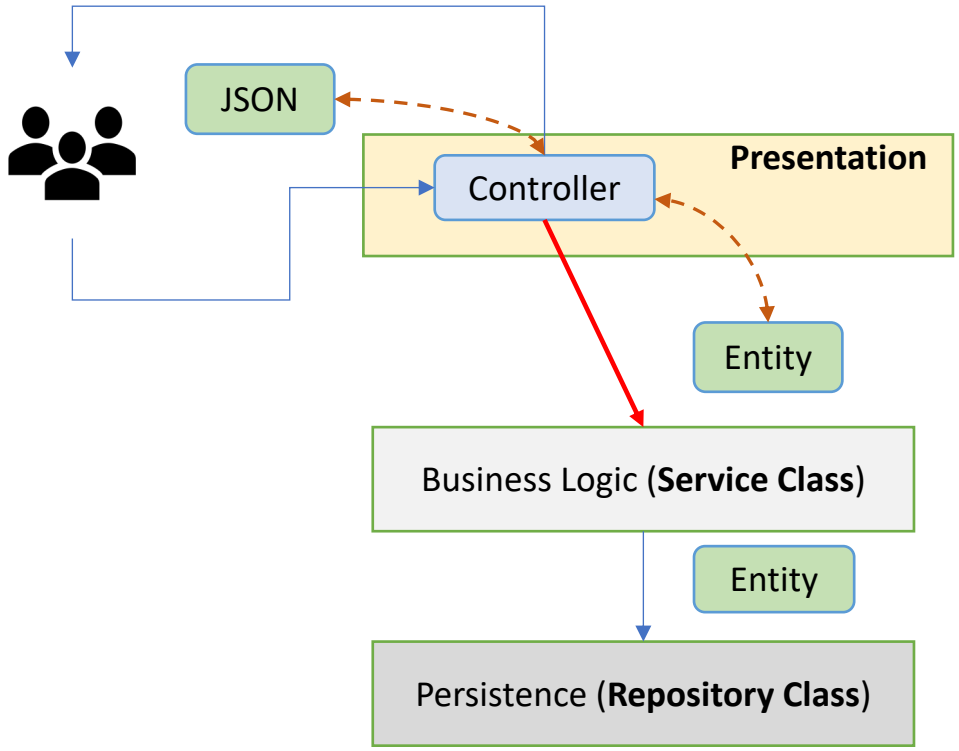
Spring RESTful API DTO

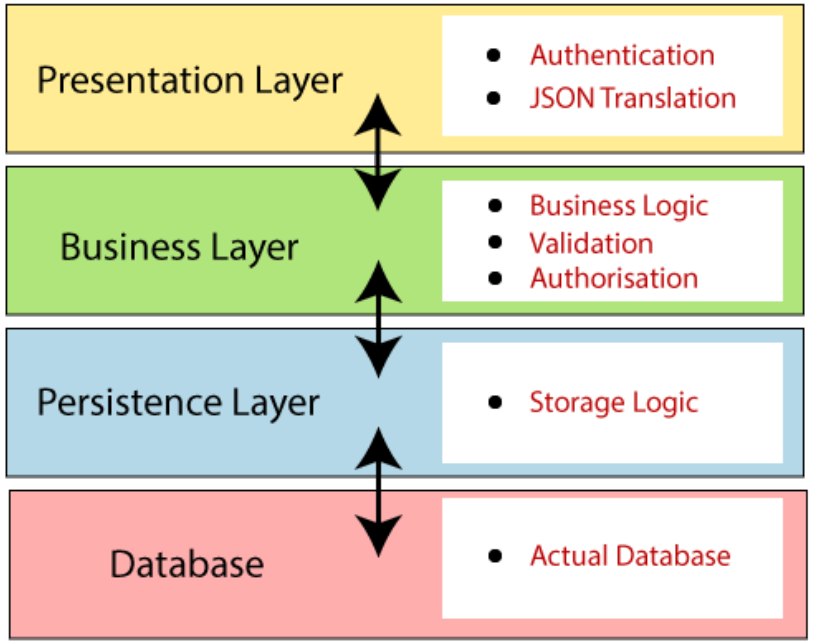
By

Pichet Limvajiranan

Made with Goodnotes

Spring Boot Layer Architectures

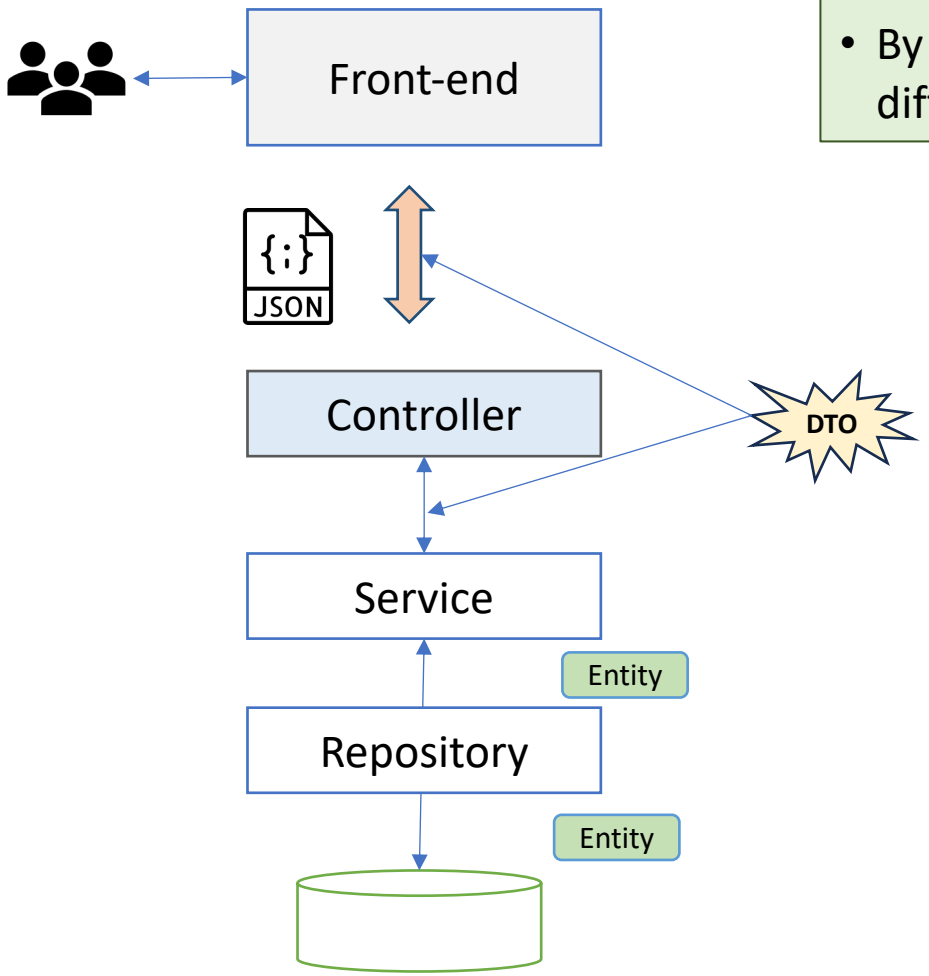


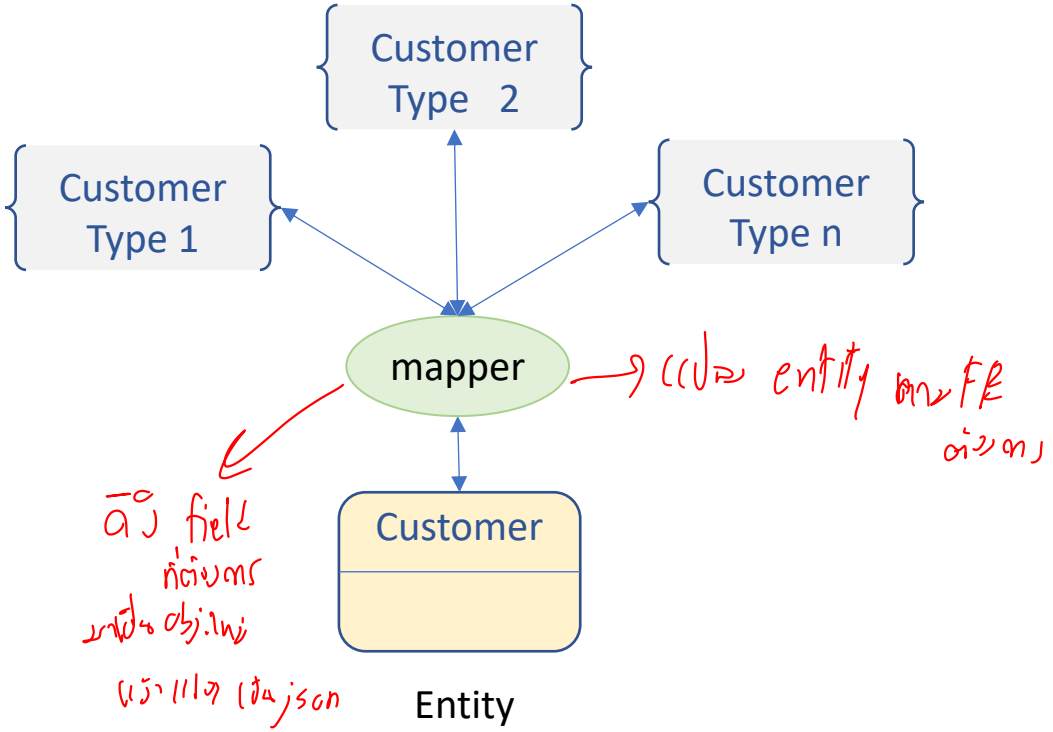


Made with Goodnotes

DTO: Data Transfer Object

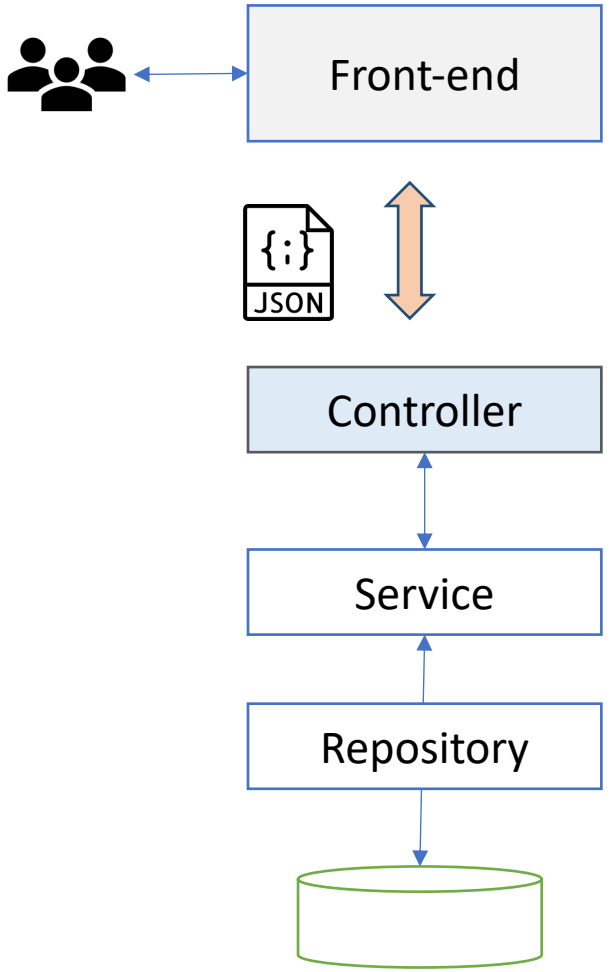
- DTOs normally are created as POJOs.
- The data is mapped from the domain models to the DTOs.
- By using DTOs, we can provide as many different versions (with different structures) of our entities as we want.



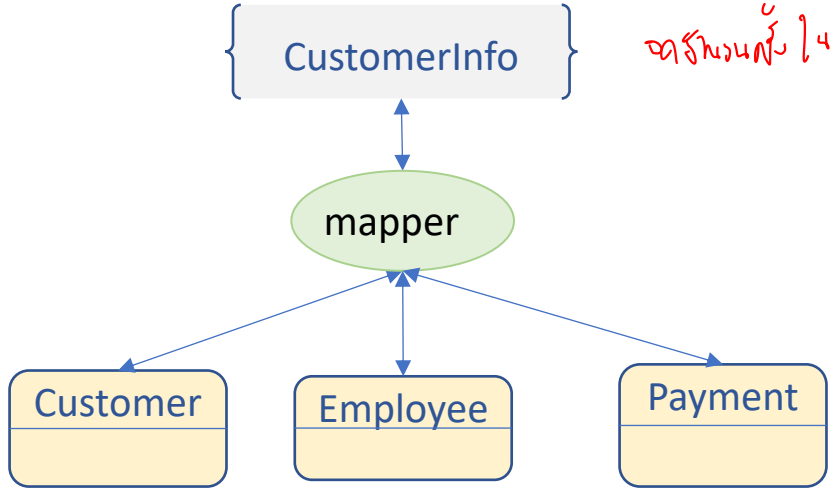


Made with Goodnotes

DTO: Data Transfer Object



DTO is a design pattern conceived to reduce the number of calls when working with remote interfaces.



Another advantage of using DTOs on RESTful APIs is that they can help hiding implementation details of domain objects (aka. entities). Exposing entities through endpoints can become a security issue if we do not carefully handle what properties can be changed through what operations.

Made with Goodnotes

Service Layer

```
@Service
public class CustomerService {
   @Autowired
   private CustomerRepository repository;
   public SimpleCustomerDTO getSimpleCustomerById(Integer id) {
      return repository.findById(id)
            .map(customer -> convertEntityToDto(customer))
            .orElseThrow(()->new ResponseStatusException(
                  HttpStatus.NOT_FOUND,  id +" Does Not Exist !!!" ));
   }
   private SimpleCustomerDTO convertEntityToDto(Customer customer) {
      SimpleCustomerDTO simpleCustomerDTO = new SimpleCustomerDTO();
      simpleCustomerDTO.setCustomerName(customer.getCustomerName());
        :
      simpleCustomerDTO.setSalesPerson(customer.getSalesRepEmployee().getFirstName()
                  + ' '+ customer.getSalesRepEmployee().getLastName());
      return simpleCustomerDTO;
   }
}
```

```
@Getter @Setter
public class SimpleCustomerDTO {
   private String customerName;
   private String phone;
   private String city;
   private String country;
   private String salesPerson;
}
```

```
@GetMapping("/{id}")
public SimpleCustomerDTO getCustomerById (@PathVariable Integer id) {
   return customerService.getSimpleCustomerById(id);
}
```

Made with Goodnotes

Model Mapper Library

- To **avoid** having to write **cumbersome/boilerplate code** to map DTOs into entities and vice-versa, we are going to use a library called **ModelMapper**.
- The goal of ModelMapper is to make object mapping easy by automatically determining how one object model maps to another.
- This library is quite powerful and accepts a whole bunch of configurations to streamline the mapping process, but it also favors convention over configuration by providing a default behavior that fits most cases.

```
<dependency>
   <groupId>org.modelmapper</groupId>
   <artifactId>modelmapper</artifactId>
   <version>3.1.1</version>
</dependency>
```

Made with Goodnotes

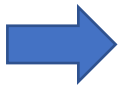
ModelMapper: How it works?

- ModelMapper consists of two separate processes
  - **The matching process** 
    - Identifying eligible properties, transforming and tokenizing their names.
      - AccessLevels and NamingConventions (Type Mapping).
        - Methods are eligible based on configured
        - Eligible **methods take precedence over fields** with the same transformed property name.
        - **Only** source methods with **zero parameters** and **a non-void** return type are **eligible.**
  - The mapping process
    - Matched property values are converted from a source to destination object.
    - If a TypeMap exists for the source and destination types, mapping will occur according to the Mappings defined in the TypeMap.
    - Else if a Converter exists that is capable of converting the source object to the destination type, mapping will occur using the Converter.

Made with Goodnotes

Customer to SimpleCutomerDTO

```java
@Entity
public class Customer {
    @Id
    private Integer id;
    private String customerName;
        :
        :
    private String postalCode;
    private String country;
    private BigDecimal creditLimit;
    @JsonIgnore
    @ManyToOne(fetch = FetchType.EAGER)
    @JoinColumn(name = "salesRepEmployeeNumber")
    private Employee salesRepEmployee;
    @OneToMany(mappedBy = "customer")
    private Set<Payment> payments = new LinkedHashSet<>();
    @OneToMany(mappedBy = "customerNumber")
    private Set<Order> orders = new LinkedHashSet<>();
}
```

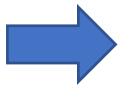


```java
@Getter
@Setter
public class SimpleCustomerDTO {
    private String customerName;
    private String phone;
    private String city;
    private String country;
}
```

Made with Goodnotes

Eligible methods

```
@Entity
public class Customer {
   @Id
   private Integer id;
   private String customerName;
      :
      :
   private String postalCode;
   private String country;
   private BigDecimal creditLimit;
   @JsonIgnore
   @ManyToOne(fetch = FetchType.EAGER)
   @JoinColumn(name = "salesRepEmployeeNumber")
   private Employee salesRepEmployee;
   @OneToMany(mappedBy = "customer")
   private Set<Payment> payments = new LinkedHashSet<>();
   @OneToMany(mappedBy = "customerNumber")
   private Set<Order> orders = new LinkedHashSet<>();
}
```



```
@Getter
@Setter
public class SimpleCustomerDTO {
   private String customerName;
   private String phone;
   private String city;
   private String country;

   public String getCountry() {
     return “Something”;
   }

}
```

Made with Goodnotes

Using ModelMapper

```
modelMapper.map(entityObject, DTOClass.class);
```

```
@Autowired
private CustomerService service;
@GetMapping("/{id}")
public SimpleCustomerDTO getCustomerById (@PathVariable Integer id) {
   return service.getSimpleCustomerById(id);
}
```

```
@Service
public class CustomerService {
  @Autowired   private CustomerRepository repository;
  @Autowired   private ModelMapper modelMapper;
   public SimpleCustomerDTO getCustomer(int customerId) {
       Customer customer = repository.findById(customerId)
         .orElseThrow(()->new ResponseStatusException(
                HttpStatus.NOT_FOUND, customerId+ " does not exist !!!"));
       return modelMapper.map(customer, SimpleCustomerDTO.class);
   }
```

Made with Goodnotes

Deep/Nested Mappings

```java
@Setter @Getter
public class SimpleEmployeeDTO {
   private String lastName;
   private String firstName;
}
```

```java
public class SimpleCustomerDTO {
   private String customerName;
   private String phone;
   private String city;
   private String country;
   private SimpleEmployeeDTO salesRepEmployee;
}
```

```json
{
    "customerName": "Atelier graphique",
    "phone": "40.32.2555",
    "city": "Nantes",
    "country": "France",
    "salesRepEmployee": {
        "lastName": "Hernandez",
        "firstName": "Gerard"
    }
}
```

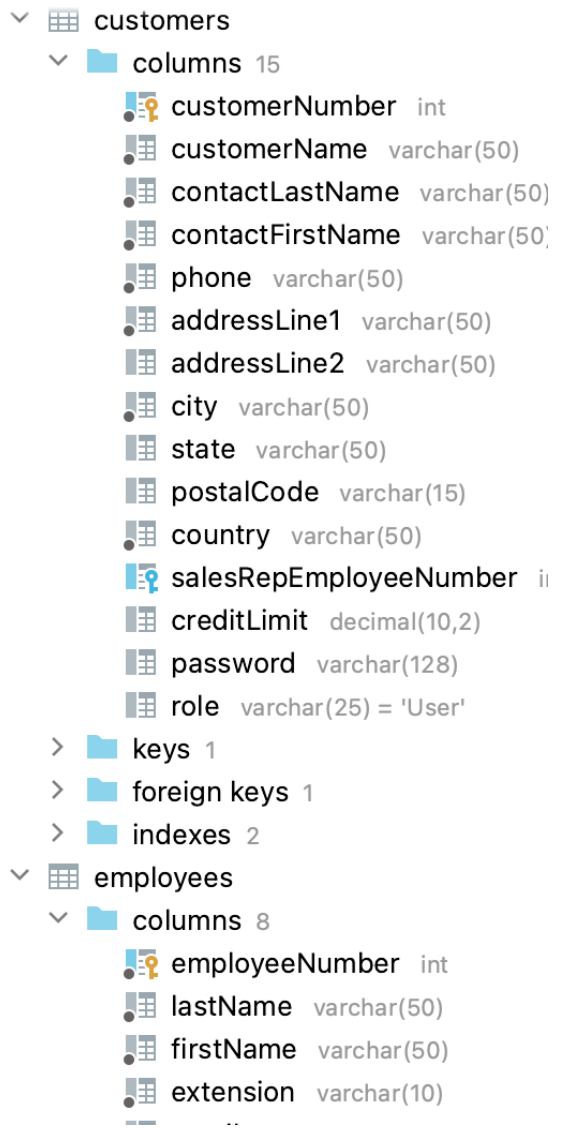
Made with Goodnotes

Deep Mappings (2)

```
@Setter @Getter
public class SimpleEmployeeDTO {
    private String lastName;
    private String firstName;
}
```

```
public class SimpleCustomerDTO {
    private String customerName;
    private String phone;
    private String city;
    private String country;
    private String salesRepEmployeeFirstName;
    private String salesRepEmployeeLastName;
}
```

```
{
    "customerName": "Atelier graphique",
    "phone": "40.32.2555",
    "city": "Nantes",
    "country": "France",
    "salesRepEmployeeFirstName": "Gerard",
    "salesRepEmployeeLastName": "Hernandez"
}
```



Made with Goodnotes

Deep Mappings (3)

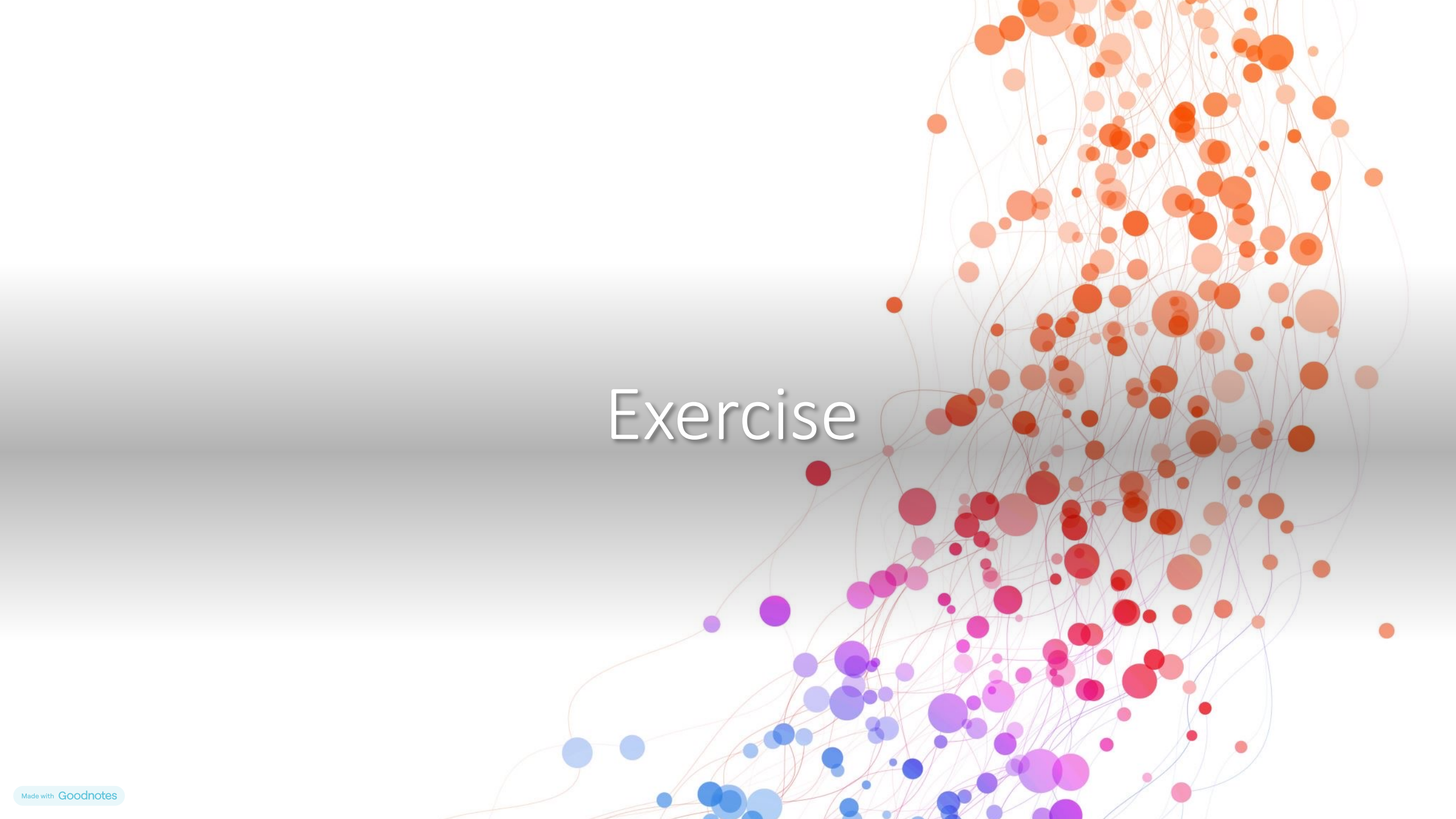
```java
@Setter @Getter
public class SimpleEmployeeDTO {
   private String lastName;
   private String firstName;
   public String getName() {
      return firstName + " "+ lastName;
   }
}
```

```java
public class SimpleCustomerDTO {
   private String customerName;
   private String phone;
   private String city;
   private String country;
  @JsonIgnore
   private SimpleCustomerDTO salesRepEmployee;
   public String getSalesPerson()  {
      return salesRepEmployee==null ? "-": salesRepEmployee.getName();
   }
}
```

```json
{
     "customerName": "Atelier graphique",
     "phone": "40.32.2555",
     "city": "Nantes",
     "country": "France",
     "salesPerson": "Gerard Hernandez"
}
```

Made with Goodnotes

Exercise



Made with Goodnotes

(1) Create Customer DTO

```
package sit.int204.classicmodelsservice.dtos

@Getter @Setter
public class SimpleCustomerDTO {
    private String customerName;
    private String phone;
    private String city;
    private String country;
    private String salesPerson;
}
```

Made with Goodnotes

Setup Model Mapper

- Add Dependency to Maven

```
<dependency>
   <groupId>org.modelmapper</groupId>
   <artifactId>modelmapper</artifactId>
   <version>3.1.1</version>
</dependency>
```

3.2.0

- Defined Bean for ModelMapper (in base package)

```
@Configuration
public class ApplicationConfig {
   @Bean
   public ModelMapper modelMapper() {
      return new ModelMapper();
   }
}
```

Made with Goodnotes

Using ModelMapper instead custom mapper

```
@Service
public class CustomerService {
   @Autowired
   private CustomerRepository repository;
   public Customer getCustomerById(Integer customerId) {
      return repository.findById(customerId).orElseThrow(()->new ResponseStatusException(
           HttpStatus.NOT_FOUND, "Customer id "+ customerId+ "Does Not Exist !!!"));
   }
}
```

```
@RestController
public class CustomerController {
  @Autowired  private CustomerService service;
  @Autowired  private ModelMapper modelMapper;
   @GetMapping("/{customerId}")
   public SimpleCustomerDTO getSimpleCustomerById(@PathVariable Integer customerId) {
      return modelMapper.map(service.getCustomerById(customerId), SimpleCustomer.class);
   }
}
```

```
{
    "customerName": "Blauer See Auto, Co.",
    "phone": "+49 69 66 90 2555",
    "city": "Frankfurt",
    "country": "Germany",
    "salesPerson": null
}
```

Made with Goodnotes

Deep Mapping (Testing for each DTO)

1

```
@Setter
@Getter
public class SimpleEmployeeDTO {
    private String lastName;
    private String firstName;
}
```

2

```
@Setter
@Getter
public class SimpleEmployeeDTO {
    private String lastName;
    private String firstName;
    public String getName() {
        return firstName + " "+ lastName;
    }
}
```

3

```
@Setter
public class SimpleEmployeeDTO {
    private String lastName;
    private String firstName;
    public String getName() {
        return firstName + " "+ lastName;
    }
}
```

```
@Getter @Setter
public class SimpleCustomerDTO {
    private String customerName;
    :
    private SimpleEmployeeDTO salesRepEmployee;
}
```

Made with Goodnotes

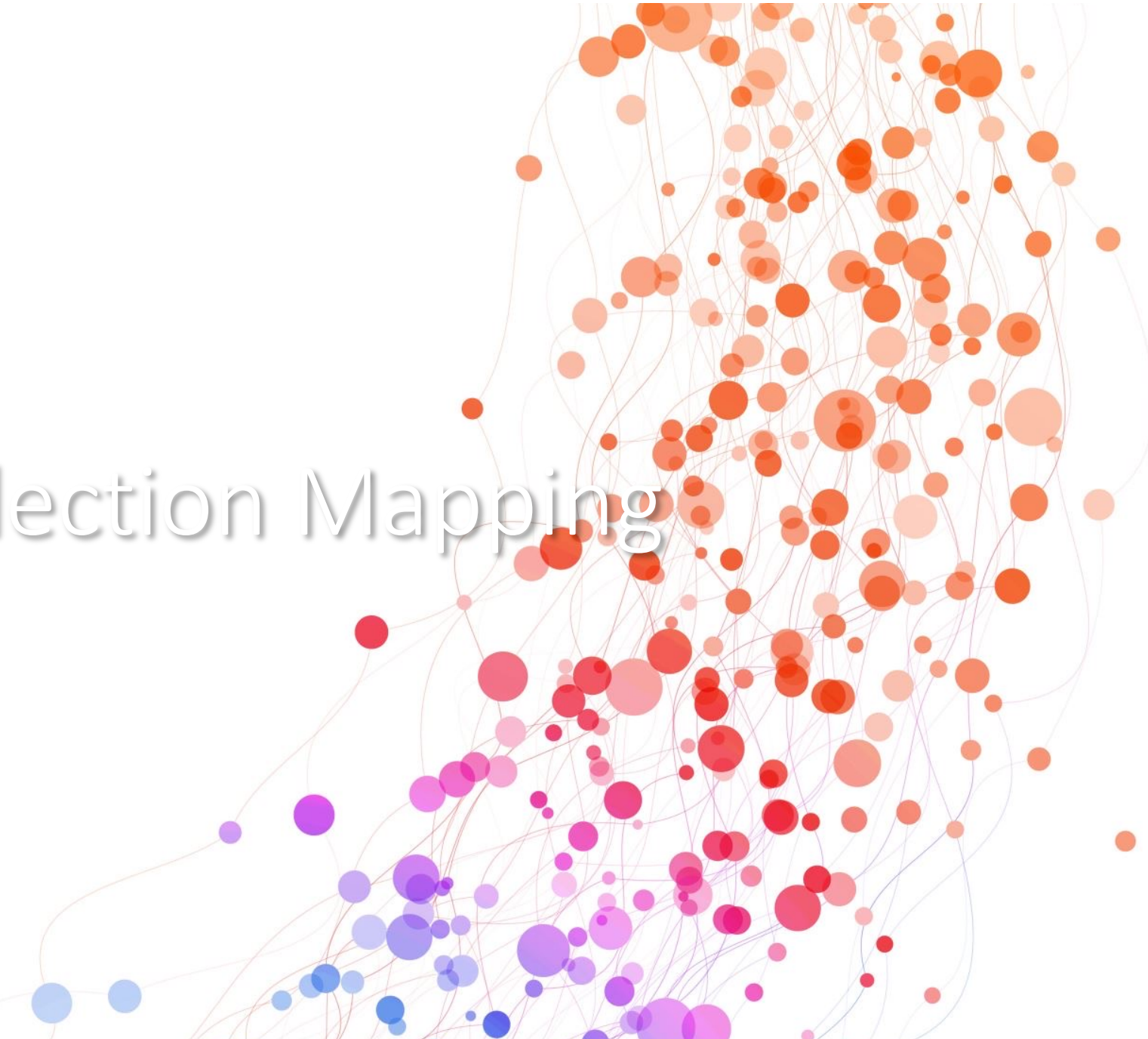
Modify SimpleCustomerDTO

```
@Getter @Setter
public class SimpleCustomerDTO {
    private String customerName;
    :
    private String salesRepEmployeeFirstName;
    private String salesRepEmployeeLastName;
}
```

```
@Getter @Setter
public class SimpleCustomerDTO {
    private String customerName;
    :
   @JsonIgnore
   private SimpleEmployeeDTO sales;
   public String getSalesPerson() {
       return sales == null ? "-" : sales.getName();
   }
}
```

Made with Goodnotes

Collection Mapping



Made with Goodnotes

Mapping Lists with ModelMapper

Map แบบ list

```
List<EmployeeDTO> dtos = employees .stream() .map(employee ->
modelMapper.map(employee, EmployeeDTO.class)) .collect(Collectors.toList());
```

convert กลับไปเป็น list

แปลงเป็น stream แล้ว map ทีละตัว

```
@GetMapping("")
public List<EmployeeDTO> getEmployees() {
    List<Employee> employeeList = repository.findAll();
     return employeeList.stream()
            .map(e -> modelMapper.map( e, EmployeeDTO.class))
            .collect(Collectors.toList());
}
```

Made with Goodnotes

General-purpose parameterized method

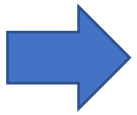
```
@GetMapping("")
public List<EmployeeDTO> getEmployees() {
    List<Employee> employeeList = repository.findAll();
     return return mapList(employeeList, EmployeeOfficeDTO.class);
}
```

```
public static <S, T> List<T> mapList(List<S> source, Class<T> targetClass) {
    return source.stream()
            .map(entity -> modelMapper.map(entity, targetClass))
            .collect(Collectors.toList());
}
```

Made with Goodnotes

Convert DTO to Entity

```java
@Getter   @Setter
@NoArgsConstructor
@AllArgsConstructor
public class EmployeeDTO {
    private Integer id;
    private String lastName;
    private String firstName;
    private String extension;
    private String email;
    private String jobTitle;
    private String officeId;
}
```

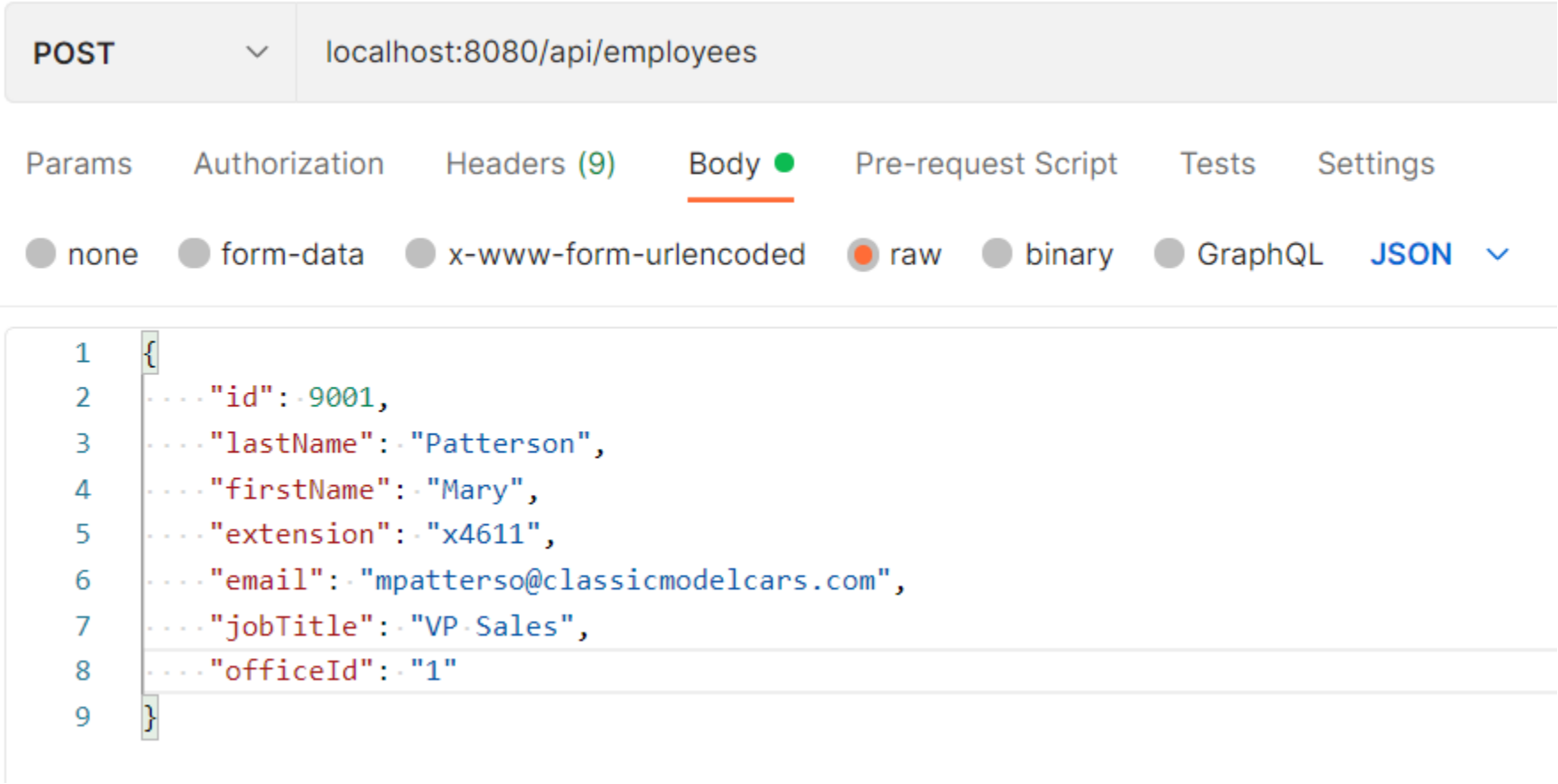


```java
@Entity
@Table(name = "employees")
public class Employee {
    @Id
    @Column(name = "employeeNumber", nullable = false)
    private Integer id;

    @Column(name = "lastName", nullable = false, length = 50)
    private String lastName;
```

```java
@PostMapping("")
public Employee create(@RequestBody EmployeeDTO newEmployee) {
    Employee employee = modelMapper.map(newEmployee, Employee.class);
    return repository.saveAndFlush(employee);
}
```

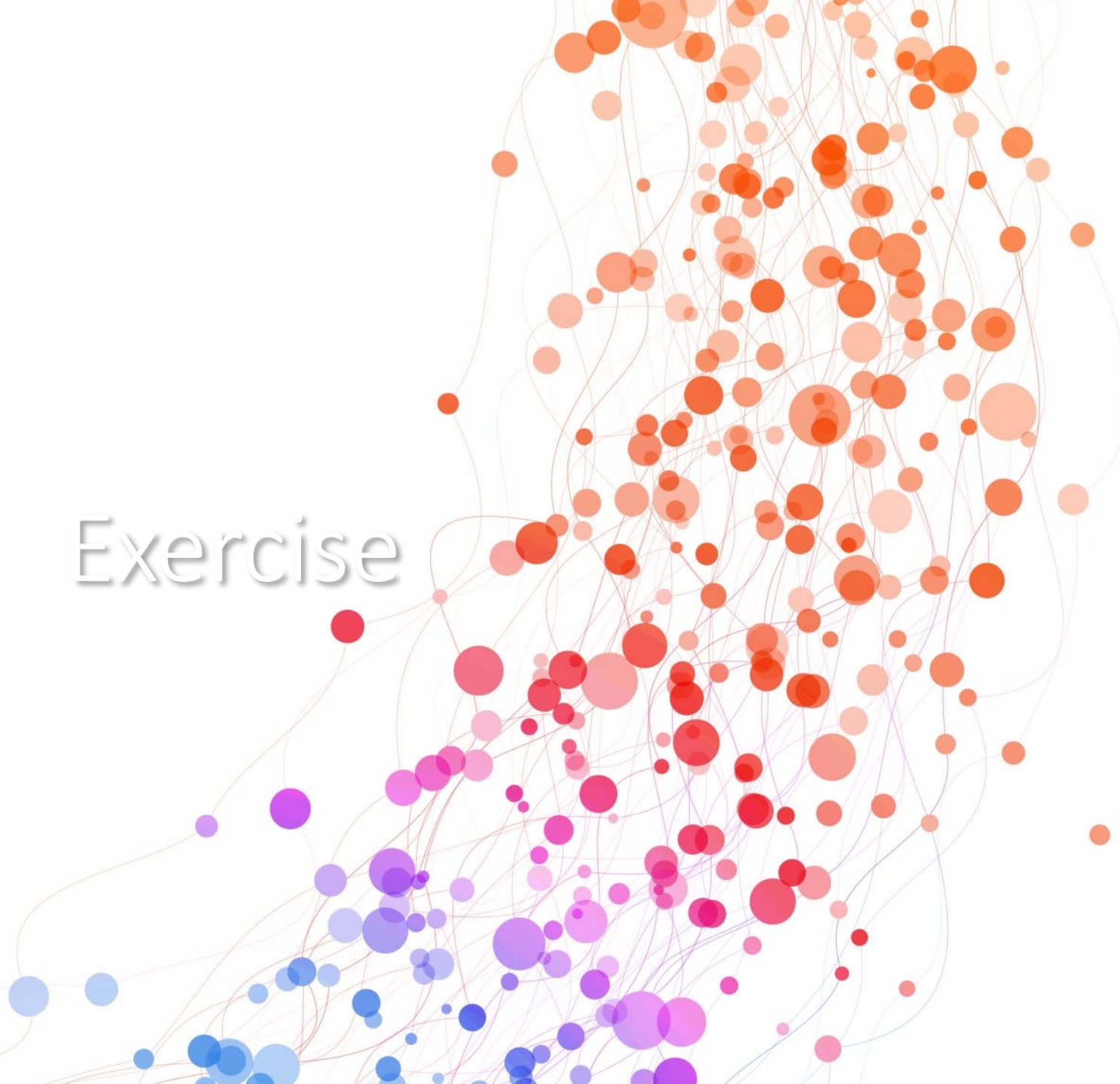
Made with Goodnotes

Request Example



Made with Goodnotes

Exercise



Made with Goodnotes

Mapping Lists with ModelMapper

```
List<EmployeeDTO> dtos = employees .stream() .map(employee ->
modelMapper.map(employee, EmployeeDTO.class))
.collect(Collectors.toList());
```

```
@GetMapping("")
public List<EmployeeDTO> getEmployees() {
    List<Employee> employeeList = repository.findAll();
     return employeeList.stream()
            .map(e -> modelMapper.map( e, EmployeeDTO.class))
            .collect(Collectors.toList());
}
```

Made with Goodnotes

General-purpose parameterized method

```
@GetMapping("")
public List<EmployeeDTO> getEmployees() {
    List<Employee> employeeList = repository.findAll();
     return return mapList(employeeList, EmployeeOfficeDTO.class, modelMapper);
}
```

```
public static <S, T> List<T> mapList(List<S> source, Class<T> targetClass, ModelMapper modelMapper) {
    return source.stream()
            .map(entity -> modelMapper.map(entity, targetClass))
            .collect(Collectors.toList());
}
```

Made with Goodnotes

Generic PageDTO Example

```
@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
public class PageDTO<T> {
    private List<T> content;
    private Boolean last;
    private Boolean first;
    private Integer totalPages;
    private Integer totalElements;
    private Integer size;
    @JsonIgnore
    private Integer number;
    public Integer getPage() {
        return number;
    }
}
```

Map page

Made with Goodnotes


```
{
  "content": [
    {
      "productCode": "S10_1678",
      "productName": "1969 Harley Davidson Ultimate Chopper",
      "productLine": "Motorcycles",
      "productScale": "1:10",
       "price": 95.7
    },
    :
    :
    {
      "productCode": "S10_1949",
      "productName": "1952 Alpine Renault 1300",
      "productLine": "Classic Cars",
      "productScale": "1:10",
      "price": 214.3
    }
  ],
  "first": true,
  "totalPages": 12,
  "totalElements": 111,
  "size": 10,
  "page": 0
}
```

Create Singleton ListMapper Service

ไม่ต้องสร้าง object

package sit.int204.classicmodelsservice.utils

```
public class ListMapper {
   private static final ListMapper listMapper = new ListMapper();
   private ListMapper() { }
   public <S, T> List<T> mapList(List<S> source,  Class<T> targetClass, ModelMapper modelMapper) {
      return source.stream().map(entity -> modelMapper.map(entity, targetClass))
            .collect(Collectors.toList());
   }
   public static ListMapper getInstance() {
      return listMapper;
   }
   public <S, T> PageDTO<T> toPageDTO(Page<S> source, Class<T> targetClass,
                              ModelMapper modelMapper) {
      PageDTO<T> page = modelMapper.map(source, PageDTO.class);
      page.setContent(mapList(source.getContent(), targetClass, modelMapper));
      return page;
   }
}
```

ต้องมี new ออกไป ไว้ return ก่อน

Made with Goodnotes

Create EmployeeDTO & Modify Application config

```
@Getter   @Setter
@NoArgsConstructor
@AllArgsConstructor
public class EmployeeDTO {
    private Integer id;
    private String lastName;
    private String firstName;
    private String extension;
    private String email;
    private String jobTitle;
    private String officeId;
}
```

```
@Configuration
public class ApplicationConfig {
   :
   @Bean
    public ListMapper listMapper() {
       return ListMapper.getInstance();
    }
}
```

```
@Entity
@Table(name = "offices")
public class Office {
    @Id
    @Column(name = "officeCode")
    private String id;
    @Column(name = "city", nullable = false, length = 50)
```

Made with Goodnotes

Create Employee Controller & Service

```
@Service
public class EmployeeService {
   @Autowired    private  EmployeeRepository repository;
   public Employee save(Employee employee) {
       return repository.saveAndFlush(employee);
   }
}
```

```
@Autowired     private ModelMapper modelMapper;
@Autowired     private ListMapper listMapper;
  :
@PostMapping("")
public Employee create(@RequestBody EmployeeDTO newEmployee) {
   Employee e = modelMapper.map(newEmployee, Employee.class);
   return employeeService.save (e);
}
```

Made with Goodnotes

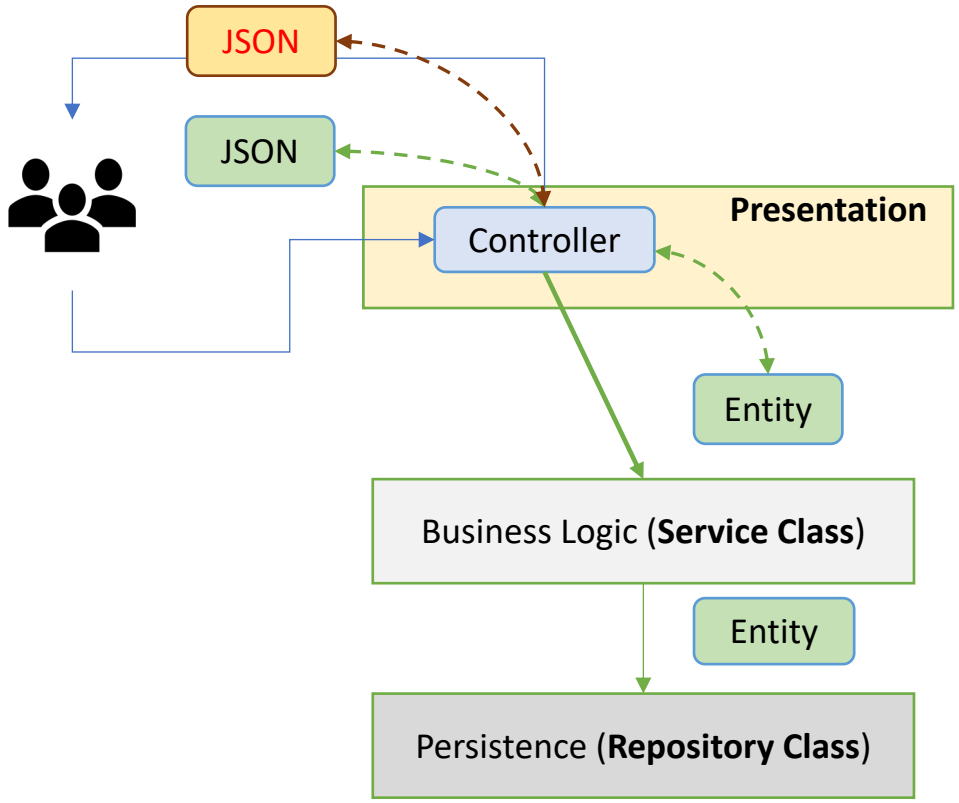
Spring RESTful API Exception Handling

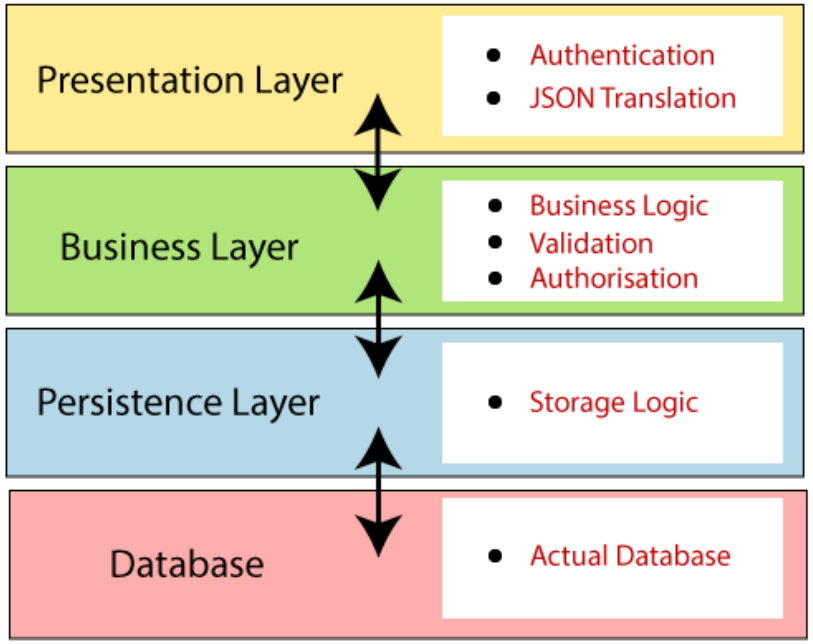
By

Pichet Limvajiranan

Made with Goodnotes

Spring Boot Layer Architectures





Made with Goodnotes

HTTP Status Codes

- When a client makes a request to an HTTP server - and the server successfully receives the request - the server must notify the client if the request was successfully handled or not.
- HTTP accomplishes this with five categories of status codes:
  - 100-level (Informational) - server acknowledges a request
  - 200-level (Success) - server completed the request as expected
  - 300-level (Redirection) - client needs to perform further actions to complete the request
  - 400-level (Client error) - client sent an invalid request
  - 500-level (Server error) - server failed to fulfill a valid request due to an error with server
- Based on the response code, a client can surmise the result of a particular request.

Made with Goodnotes

Handling Errors

- The first step in handling errors is to provide a client with a proper **status code**. Additionally, we may need to provide more information in the response body.
- Basic Responses
  - The simplest way we handle errors is to respond with an appropriate status code.
  - Here are some common response codes:
    - 400 Bad Request - client sent an invalid request, such as lacking required request body or parameter
    - 401 Unauthorized - client failed to authenticate with the server
    - 403 Forbidden - client authenticated but does not have permission to access the requested resource
    - 404 Not Found - the requested resource does not exist
    - 412 Precondition Failed - one or more conditions in the request header fields evaluated to false
    - 500 Internal Server Error - a generic error occurred on the server
    - 503 Service Unavailable - the requested service is not available

Made with Goodnotes

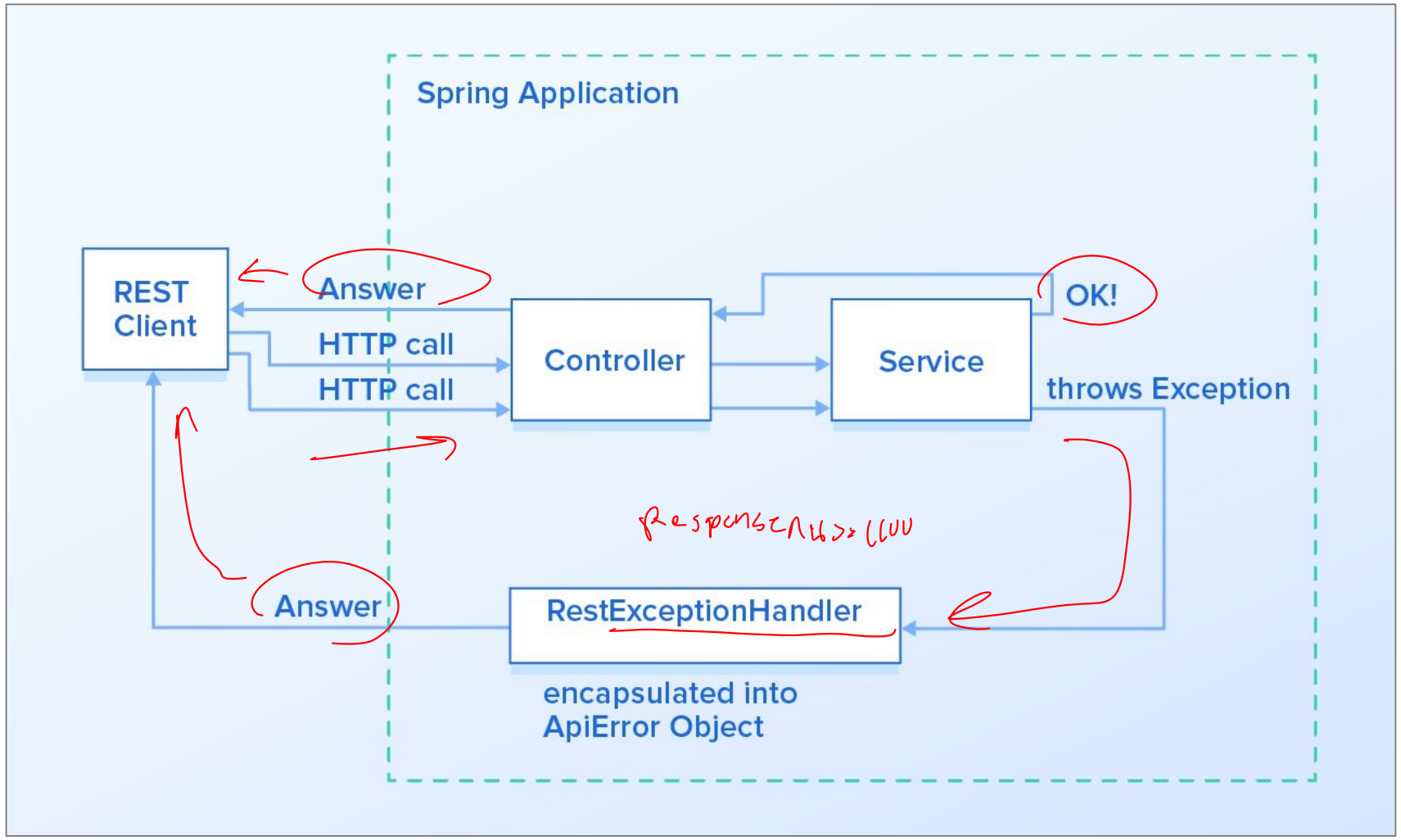
Standardized Response Bodies

- In an effort to standardize REST API error handling, the IETF devised RFC 7807, which creates a generalized error-handling schema.
- This schema is composed of five parts:
  - type - a URI identifier that categorizes the error
  - title - a brief, human-readable message about the error
  - status - the HTTP response code (optional)
  - detail - a human-readable explanation of the error
  - instance - a URI that identifies the specific occurrence of the error

```
{
    "type": "/errors/incorrect-user-pass",
    "title": "Incorrect username or password.",
    "status": 401,
    "detail": "Authentication failed due to incorrect username or password.",
    "instance": "/login/log/abc123"
}
```

Made with Goodnotes

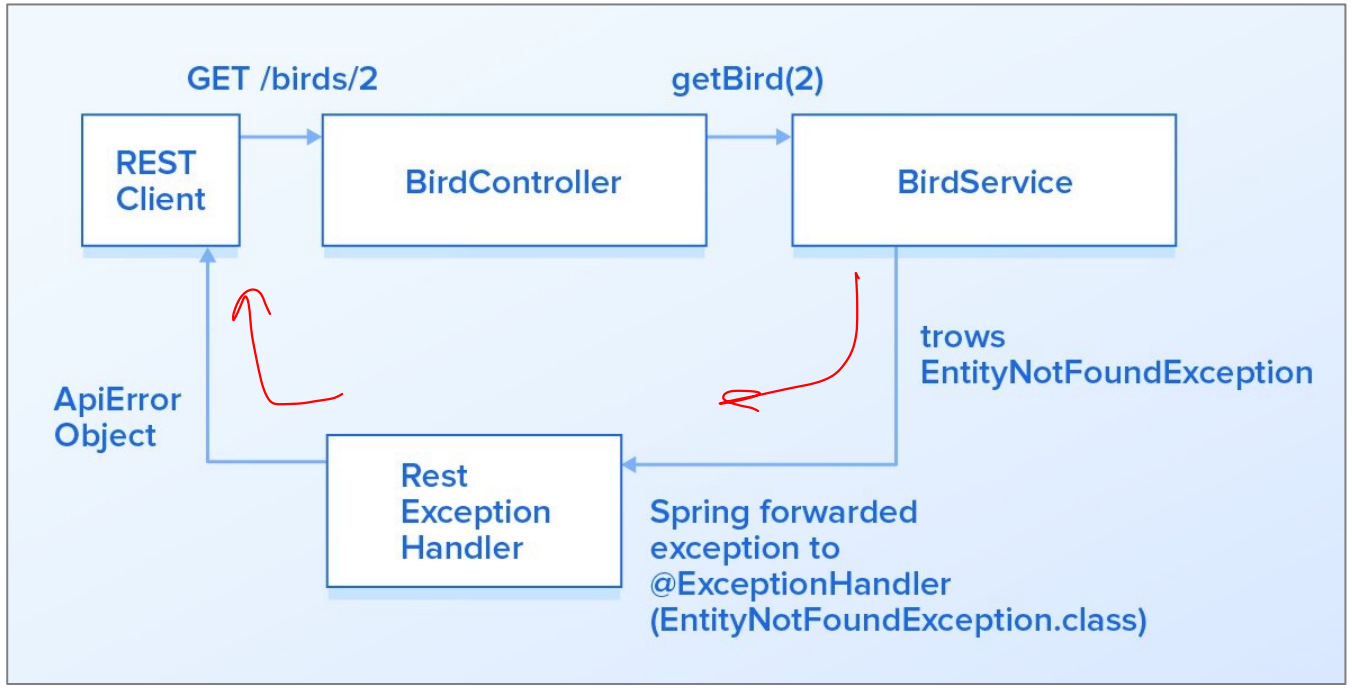
Rest Api - Exception Handling



Made with Goodnotes

Exception Handling

- Handling exceptions is an important part of building a robust application.
- Spring Boot provides tools to handle exceptions beyond simple ‘try-catch’ blocks.
  - @ResponseStatus
  - @ExceptionHandler
  - @ControllerAdvice



Made with Goodnotes

Spring Boot’s Default Exception Handling Mechanism

server.error.include-stacktrace=*always*

Body Cookies Headers (4) Test Results

Pretty Raw Preview Visualize JSON


```
{
    "timestamp": "2023-03-19T03:20:25.325+00:00",
    "status": 500,
    "error": "Internal Server Error",
    "trace": "org.springframework.web.client.HttpClientErrorExceptio
        lambda$getProductById$0(ProductService.java:25)\n\tat java.
        services.ProductService.getProductById(ProductService.java:
        (ProductController.java:56)\n\tat java.base/jdk.internal.re
        java.base/java.lang.reflect.Method.invoke(Method.java:577)\
        (InvocableHandlerMethod.java:207)\n\tat org.springframework
        java:152)\n\tat org.springframework.web.servlet.mvc.method.
        (ServletInvocableHandlerMethod.java:117)\n\tat org.springfra
        invokeHandlerMethod(RequestMappingHandlerAdapter.java:884)\
```


server.error.include-stacktrace=*on_param*
server.error.include-exception=true

Body Cookies Headers (4) Test Results

Pretty Raw Preview Visualize JSON


```
{
    "timestamp": "2023-03-19T03:42:00.775+00:00",
    "status": 500,
    "error": "Internal Server Error",
    "exception": "org.springframework.web.client.HttpClientErrorE
    "message": "404 1 does not exists !!!",
    "path": "/api/products/dtos/1"
}
```


Enum Constants

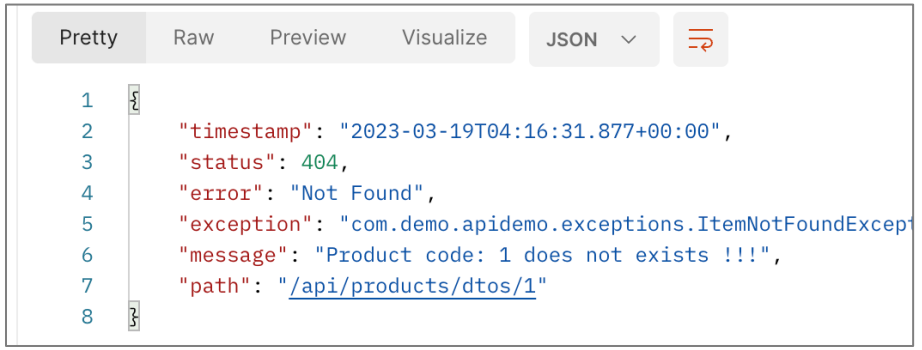
Enum Constant	Description
ALWAYS	Always add stacktrace information.
NEVER	Never add stacktrace information.
ON_PARAM	Add stacktrace attribute when the appropriate request parameter is not "false".

Made with Goodnotes

@ResponseStatus

- As the name suggests, @ResponseStatus allows us to modify the HTTP status of our response. It can be applied in the following places:
  - On the exception class itself
  - Along with the @ExceptionHandler annotation on methods
  - Along with the @ControllerAdvice annotation on classes
- In this section, we'll be looking at the first case only.

```
@ResponseStatus(value = HttpStatus.NOT_FOUND)
public class ItemNotFoundException extends RuntimeException {
   public ItemNotFoundException(String message) {
      super(message);
   }
```




```
public Product getProductById(String productCode) {
   return repo.findById(productCode).orElseThrow(
         ()->new ItemNotFoundException("Product code: "+ productCode + " does not exists !!!"));
}
```

Made with Goodnotes

Another way to achieve the same is by extending the ResponseStatusException class

@ResponseStatus, in combination with the server.error configuration properties, allows us to manipulate almost all the fields in our Spring-defined error response payload.

```
import org.springframework.http.HttpStatus;
import org.springframework.web.server.ResponseStatusException;

public class ItemNotFoundException extends ResponseStatusException {

    public ItemNotFoundException(String message){
        super(HttpStatus.NOT_FOUND, message);
    }
}
```

Made with Goodnotes

@ExceptionHandler

- The @ExceptionHandler annotation gives us a lot of flexibility in terms of handling exceptions.
- For starters, to use it, we simply need to create a method either in the controller itself or in a @RestControllerAdvice class and annotate it with @ExceptionHandler:

```
public class ItemNotFoundException extends RuntimeException {
   public ItemNotFoundException(String message) {
      super(message);
   }
  @Override
   public synchronized Throwable fillInStackTrace() {
      return this;
   }
```

```
@RestController
public class ProductController {
 :
    @ExceptionHandler(ItemNotFoundException.class)
    @ResponseStatus(HttpStatus.NOT_FOUND)
     public ItemNotFoundException handleItemNotFound (
         ItemNotFoundException exception) {
          return exception;
     }
```

Made with Goodnotes

Exception Error Code

- Now, let's finalize an error response payload for our APIs. In case of any error, clients usually expect two things:
  - An error code that tells the client what kind of error it is. Error codes can be used by clients in their code to drive some business logic based on it.
    - Usually, error codes are standard HTTP status codes, but we have also seen APIs returning custom errors code likes E001.
  - An additional human-readable message which gives more information on the error and even some hints on how to fix them or a link to API docs.
- We will also add an optional stackTrace field which will help us with debugging in the development environment.

Made with Goodnotes

Handling validation errors in the response.

```
@Getter
@Setter
@RequiredArgsConstructor
@JsonInclude(JsonInclude.Include.NON_NULL)
public class ErrorResponse {
   private final int status;
   private final String message;
   private final String instance;
   private String stackTrace;
   private List<ValidationError> errors;

   @Getter
   @Setter
   @RequiredArgsConstructor
   private static class ValidationError {
      private final String field;
      private final String message;
   }
   public void addValidationError(String field, String message){
      if(Objects.isNull(errors)){
         errors = new ArrayList<>();
      }
      errors.add(new ValidationError(field, message));
   }
}
```

```
@ExceptionHandler(ItemNotFoundException.class)
@ResponseStatus(code = HttpStatus.NOT_FOUND)
public ResponseEntity<ErrorResponse>
handleItemNotFound(ItemNotFoundException ex, WebRequest request) {
    ErrorResponse er = new
ErrorResponse(HttpStatus.NOT_FOUND.value(), ex.getMessage(),
            request.getDescription(false));
return ResponseEntity.status(HttpStatus.NOT_FOUND).body(er);
}
```

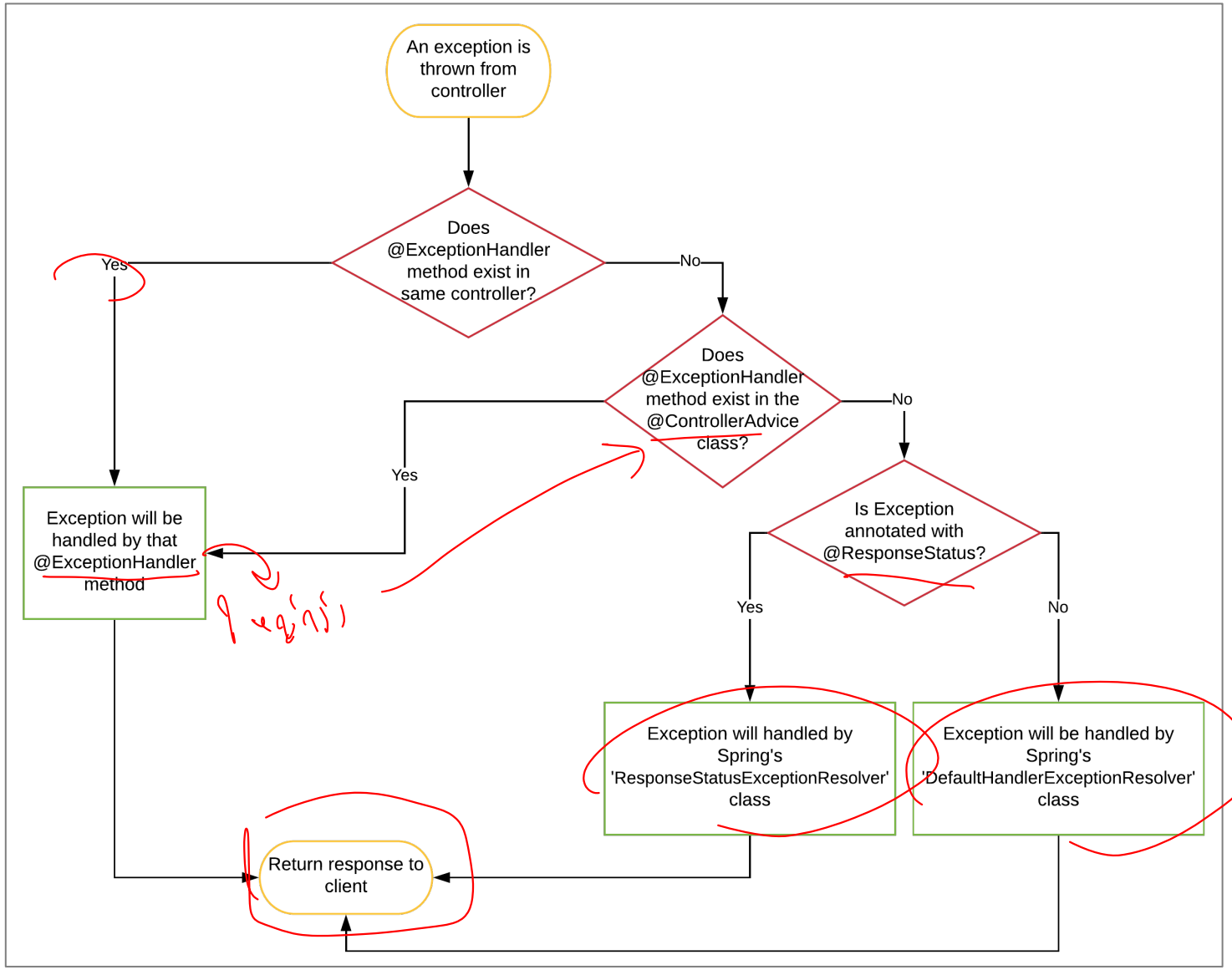
Made with Goodnotes

@RestControllerAdvice

- The term 'Advice' comes from Aspect-Oriented Programming (AOP) which allows us to inject cross-cutting code (called "advice") around existing methods. A controller advice allows us to intercept and modify the return values of controller methods, in our case to handle exceptions.
- Controller advice classes allow us to apply exception handlers to more than one or all controllers in our application:
- If we want to selectively apply or limit the scope of the controller advice to a particular controller, or a package, we can use the properties provided by the annotation:
  - @RestControllerAdvice("com.reflectoring.controller"): we can pass a package name or list of package names in the annotation's value or basePackages parameter. With this, the controller advice will only handle exceptions of this package's controllers.
  - @RestControllerAdvice(assignableTypes={Controller.class}): only controllers specify by assignableType will be handled by the controller advice.

Made with Goodnotes

How Does Spring Process The Exceptions?



Made with Goodnotes

@RestControllerAdvice (1)

```
@RestControllerAdvice
public class GlobalExceptionHandler extends ResponseEntityExceptionHandler {

    @ExceptionHandler(ItemNotFoundException.class)
    @ResponseStatus(HttpStatus.NOT_FOUND)
    public ResponseEntity<ErrorResponse> handleItemNotFoundException(
            ItemNotFoundException exception, WebRequest request) {
        return buildErrorResponse(exception, HttpStatus.NOT_FOUND, request);
    }
```

Made with Goodnotes

@ControllerAdvice (2)

```
@ExceptionHandler(MethodArgumentNotValidException.class)
@ResponseStatus(HttpStatus.UNPROCESSABLE_ENTITY)
public ResponseEntity<ErrorResponse> handleMethodArgumentNotValid(
        MethodArgumentNotValidException ex,
        WebRequest request
) {
    ErrorResponse errorResponse = new ErrorResponse(
            HttpStatus.UNPROCESSABLE_ENTITY.value(),
            "Validation error. Check 'errors' field for details.“, request.getDescription(false)
    );

    for (FieldError fieldError : ex.getBindingResult().getFieldErrors()) {
        errorResponse.addValidationError(fieldError.getField(),
                fieldError.getDefaultMessage());
    }
    return ResponseEntity.unprocessableEntity().body(errorResponse);
}
```

Made with Goodnotes

@ControllerAdvice (3)

```
@ExceptionHandler(Exception.class)
@ResponseStatus(HttpStatus.INTERNAL_SERVER_ERROR)
public ResponseEntity<ErrorResponse> handleAllUncaughtException(
        Exception exception, WebRequest request) {


    return buildErrorResponse(
            exception, "Unknown error occurred", HttpStatus.INTERNAL_SERVER_ERROR,  request
    );
}
```

Made with Goodnotes

@ControllerAdvice (4)

```
private ResponseEntity<ErrorResponse> buildErrorResponse(
      Exception exception, HttpStatus httpStatus, WebRequest request) {
   return buildErrorResponse( exception, exception.getMessage(), httpStatus, request);
}

private ResponseEntity<ErrorResponse> buildErrorResponse(
      Exception exception, String message, HttpStatus httpStatus, WebRequest request) {


     ErrorResponse errorResponse = new ErrorResponse(httpStatus.value(), message,
           request.getDescription(false)
   );

   return ResponseEntity.status(httpStatus).body(errorResponse);
}
```

Made with Goodnotes

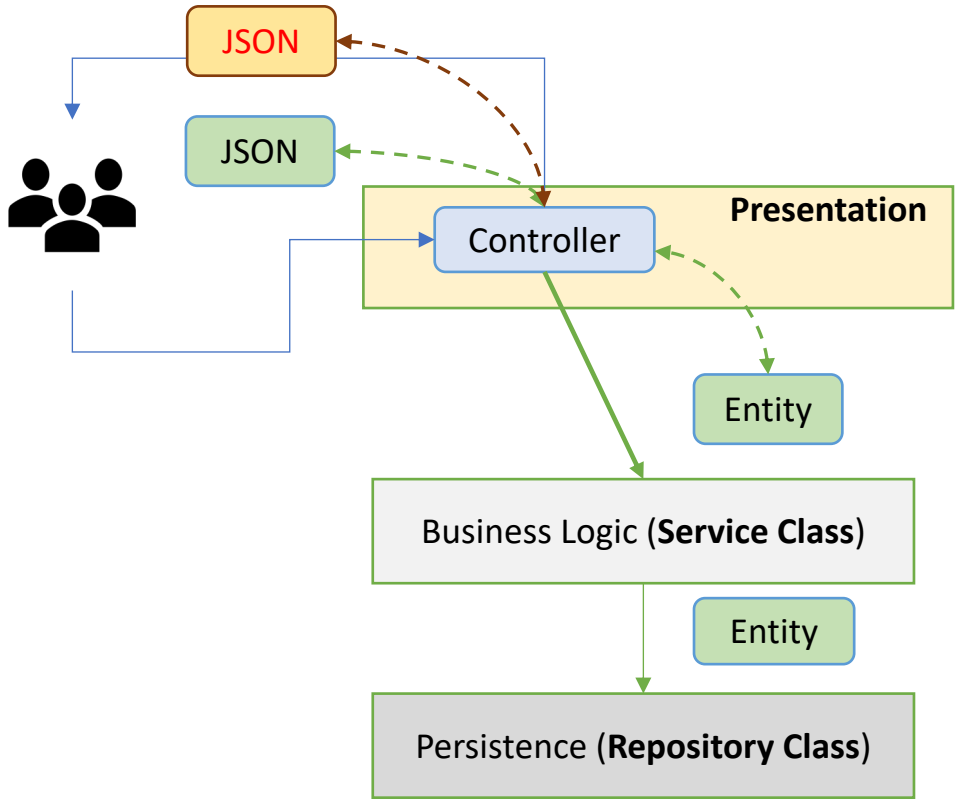
Spring RESTful API Bean Validation Basics

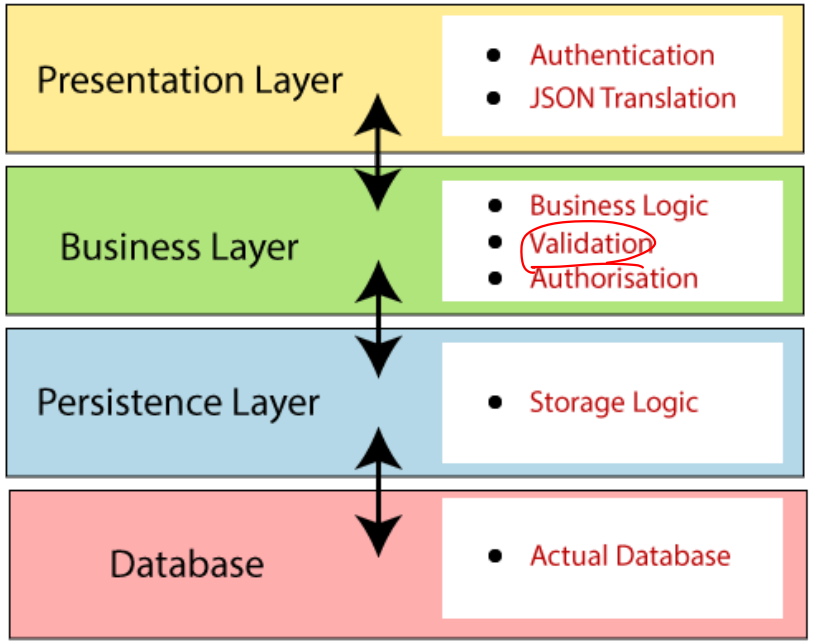
By

Pichet Limvajiranan

Made with Goodnotes

Spring Boot Layer Architectures





Made with Goodnotes

Handling validation errors in the response.

```java
@Data
@RequiredArgsConstructor
@JsonInclude(JsonInclude.Include.NON_NULL)
public class ErrorResponse {
  private final int status;
  private final String message;
  private final String instance;
  private String stackTrace;
  private List<ValidationError> errors;

  public void addValidationError(String field, String message) {
    if (Objects.isNull(errors)) {
      errors = new ArrayList<>();
    }
    errors.add(new ValidationError(field, message));
  }
}
```

```java
@Data
@RequiredArgsConstructor
private static class ValidationError {
   private final String field;
   private final String message;
}
```

Made with Goodnotes

@RestControllerAdvice (1)

```
@RestControllerAdvice
public class GlobalExceptionHandler extends ResponseEntityExceptionHandler {

    @ExceptionHandler(ItemNotFoundException.class)
    @ResponseStatus(HttpStatus.NOT_FOUND)
    public ResponseEntity<ErrorResponse> handleItemNotFoundException(
            ItemNotFoundException exception, WebRequest request) {
        return buildErrorResponse(exception, HttpStatus.NOT_FOUND, request);
    }
```

Made with Goodnotes

@ControllerAdvice (2)

```
@ExceptionHandler(MethodArgumentNotValidException.class)
@ResponseStatus(HttpStatus.UNPROCESSABLE_ENTITY)
public ResponseEntity<ErrorResponse> handleMethodArgumentNotValid(
        MethodArgumentNotValidException ex,
        WebRequest request
) {
    ErrorResponse errorResponse = new ErrorResponse(
            HttpStatus.UNPROCESSABLE_ENTITY.value(),
            "Validation error. Check 'errors' field for details.“, request.getDescription(false)
    );

    for (FieldError fieldError : ex.getBindingResult().getFieldErrors()) {
        errorResponse.addValidationError(fieldError.getField(),
                fieldError.getDefaultMessage());
    }
    return ResponseEntity.unprocessableEntity().body(errorResponse);
}
```

Made with Goodnotes

Java Bean Validation

- Bean Validation provides a common way of validation through constraint declaration and metadata for Java applications.
- Using by annotate domain model properties with declarative validation constraints which are then enforced by the runtime.
- There are built-in constraints, and you can also define your own custom constraints.
- Example:

```
@Table(name = "offices")
public class Office {
    @Id
    @Column(name = "officeCode", nullable = false, length = 10)
    @NotBlank
    private String officeCode;

    @Column(name = "city", nullable = false, length = 15)
    @Size(min=5,max = 15)
    @NotBlank
    private String city;
```

```
import jakarta.validation.constraints.NotBlank;
import jakarta.validation.constraints.NotNull;
import jakarta.validation.constraints.Size;
```

Made with Goodnotes

Common Validation Annotations

Some of the most common validation annotations are:

- **@NotNull** - validates that the annotated field is not null
- **@NotEmpty** - validates that the annotated field is not null or empty
- **@Min** - validates that the annotated field has a value no smaller than the value attribute
- **@Max** - validates that the annotated field has a value no greater than the value attribute
- **@Email** - validates that the annotated field is a valid email address
- **@Past** - validates that the date field is in the past
- **@PastOrPresent** - validates that the date field is in the past
- **@Pattern** - validates that the string field is matches a certain regular expression
- **@Email -** validates that the string field must be a valid email address.

Made with Goodnotes

Validator

- To validate if an object is valid, we pass it into a Validator which checks if the constraints are satisfied:

```
@Service
 public class OfficeService {
   @Autowired
   private Validator validator;

   public Office createNewOffice(Office office) {
      Errors errors = validator.validateObject(office);
      if (errors.hasErrors()) {
         throw new RuntimeException(errors.toString());
      }
      return repository.save(office);
   }
```

Made with Goodnotes

@Validation & @Valid

- We can use @Validation and @Valid annotations to let Spring know that we want to have a certain object validated.
- The @Validated annotation is a **class-level** annotation that we can use to tell Spring to validate parameters that are passed into a method of the annotated class.
- We can put the @Valid annotation **on method parameters and fields** to tell Spring that we want a method parameter or field to be validated.

Made with Goodnotes

Validating Input to a Spring REST Controller

- There are three things we can validate for any incoming HTTP request:
  - The request body
    - In POST and PUT requests, it's common to pass a JSON payload within the request body. Spring automatically maps the incoming JSON to a Java object. Now, we want to check if the incoming Java object meets our requirements.
  - Variables within the path (e.g. id in /foos/{id}) and Query parameters
    - We're not validating complex Java objects in this case, since path variables and request parameters are primitive types like int or their counterpart objects like Integer or String.

Made with Goodnotes

Validating a Request Body

- ```
  @Data
  public class NewCustomerDto {
      @NotNull
      @Min(900)
      private Integer id;
      @NotEmpty
      @Size(min=5, max = 50)
      private String customerName;
      @Size(min=3, max = 50)
      private String contactFirstName;
      @Size(min=3, max = 50)
      private String contactLastName;
      @Pattern(regexp = "^\\s*(?:\\+?(\\d{1,3}))?[-. (]*(\\d{3})[-. )]*(\\d{3})[-.
  ]*(\\d{4})(?: *x(\\d+))?\\s*$")
      private String phone;
  ```

Made with Goodnotes
  ```

^\s*(?:\+?(\d{1,3}))?[-. (]*(\d{3})[-.)]*(\d{3})[-.]*(\d{4})(?: *x(\d+))?\s*$

Expression	Description
^\s*	#Line start, match any whitespaces at the beginning if any.
(?:\+?(\d{1,3}))?	#GROUP 1: The country code. Optional. [-. (]* #Allow certain non numeric characters that may appear between the Country Code and the Area Code.
(\d{3})	#GROUP 2: The Area Code. Required. [-.)]* #Allow certain non numeric characters that may appear between the Area Code and the Exchange number.
(\d{3})	#GROUP 3: The Exchange number. Required. [-.]* #Allow certain non numeric characters that may appear between the Exchange number and the Subscriber number.
(\d{4})	#Group 4: The Subscriber Number. Required.
(?: *x(\d+))?	#Group 5: The Extension number. Optional. \s*$ #Match any ending whitespaces if any and the end of string.

Group1: Country Code (ex: 1 or 86)
Group2: Area Code (ex: 800)
Group3: Exchange (ex: 555)
Group4: Subscriber Number (ex: 1234)
Group5: Extension (ex: 5678)

18005551234 1
800 555 1234
+1 800 555-1234
+86 800 555 1234

1-800-555-1234 1
(800) 555-1234
(800)555-1234
(800) 555-1234

Made with Goodnotes

Validating a Request Body

- To validate the request body of an incoming HTTP request, we annotate the request body with the @Valid annotation in a REST controller:

```
@RestController
@RequestMapping("/customers")
public class CustomerControlller {
    :
    :
    @PostMapping("")
    public NewCustomerDto createCustomer(@Valid @RequestBody NewCustomerDto newCustomer) {
        return service.createCustomer(newCustomer);
    }
}
```

- If the validation fails, it will trigger a MethodArgumentNotValidException.
- By default, Spring will translate this exception to a HTTP status 400 (Bad Request).

Made with Goodnotes

Exercise: Validate Customer

1. Add Dependency to project.

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-validation</artifactId>
</dependency>
```

2. Create Customer Entity (Customer.java), Customer DTO (NewCustomerDto.java & Employee DTO (SimpleEmployeeDto.java)
3. Create Customer service
4. Create Customer controller

** You can gets 2-4 from MS-Teams **

Made with Goodnotes

Customer Service

```
@Service
public class CustomerService {
    :
    :
    public NewCustomerDto createCustomer(NewCustomerDto newCustomer) {
        if(repository.existsById(newCustomer.getId())){
            throw new ResponseStatusException(HttpStatus.CONFLICT, "Duplicate customer for id "+
                    newCustomer.getId());
        }
        Customer customer = mapper.map(newCustomer, Customer.class);
        return mapper.map(repository.saveAndFlush(customer), NewCustomerDto.class);
    }
    public List<NewCustomerDto> getAllCustomers() {
        return listMapper.mapList(repository.findAll(), NewCustomerDto.class, mapper);
    }
}
```

Made with Goodnotes

NewCustomer & SimpleEmployee DTO

```java
@Data
public class SimpleEmployeeDto {
   private Integer employeeNumber;
   @JsonIgnore
   private String firstName;
   @JsonIgnore
   private String lastName;
   public String getName() {
      return firstName + ' '+ lastName;
   }
}
```

```java
@Data
public class NewCustomerDto {
   @NotNull
   @Min(900)
   private Integer id;
   @NotEmpty
   @Size(min=5, max = 50)
   private String customerName;
   @Size(min=3, max = 50)
   private String contactFirstName;
   @Size(min=3, max = 50)
   :
   @NotNull
   private SimpleEmployeeDto sales;
   @Min(0)   @Max(10000)
   @NotNull(message = "Credit Limit Must be >=0 and <=10,000")
   private Double creditLimit;
}
```

Made with Goodnotes

Customer Controller

```
@RestController
@RequestMapping("/customers")
public class CustomerControlller {
    @Autowired
    CustomerService service;

    @GetMapping
    public List<NewCustomerDto> getCustomers() {
        return service.getAllCustomers();
    }
    @PostMapping("")
    public NewCustomerDto createCustomer(
      @Valid @RequestBody NewCustomerDto newCustomer) {

        return service.createCustomer(newCustomer);
    }
}
```

Made with Goodnotes

Test Data

```
{
"id": null,
"customerName": "SI",
"contactFirstName": "Khaitong",
"contactLastName": "Lim",
"phone": "0861681110",
"addressLine1": "54, rue Royale",
"addressLine2": null,
"city": "Nantes",
"state": null,
"postalCode": "44000",
"country": "France",
"sales" : {
    "employeeNumber": 1370
},
"creditLimit": 9000
}
```

```
{
"id": 201,
"customerName": "SI",
"contactFirstName": "Khaitong",
"contactLastName": "Lim",
"phone": "0861681110",
"addressLine1": "54, rue Royale",
"addressLine2": null,
"city": "Nantes",
"state": null,
"postalCode": "44000",
"country": "France",
"sales" : {
    "employeeNumber": 1370
},
"creditLimit": 9000
}
```

```
{
"id": 901,
"customerName": "SIT Vintage Shop",
"contactFirstName": "Khaitong",
"contactLastName": "Lim",
"phone": "0861681110",
"addressLine1": "54, rue Royale",
"addressLine2": null,
"city": "Nantes",
"state": null,
"postalCode": "44000",
"country": "France",
"sales" : {
    "employeeNumber": 1370
},
"creditLimit": 9000
}
```

Made with Goodnotes

Validating Input to a Spring Service Method

- Instead of (or additionally to) validating input on the controller level, we can also validate the input to any Spring components. In order to to this, we use a combination of the @Validated and @Valid annotations:

```
@Service
@Validated
class ValidatingService{

    void validateCustomer(@Valid Customer customer){
        // do something
    }
}
```

Made with Goodnotes

Validating Path Variables and Request Parameters

- Instead of annotating a class field like above, we're adding a constraint annotation (in this case @Min) directly to the method parameter in the Spring controller
- In contrast to request body validation a failed validation will trigger a HandlerMethodValidationException instead of a MethodArgumentNotValidException.

```
@GetMapping("")
  public ResponseEntity<Object> getAllProducts(
        @RequestParam(defaultValue = "") String partOfProductName,
        @RequestParam(defaultValue = "0") Double lower,
        @RequestParam(defaultValue = "0") Double upper,
        @RequestParam(defaultValue = "") String sortBy,
        @RequestParam(defaultValue = "ASC") String sortDirection,
        @RequestParam(defaultValue = "0") @Min(0) int pageNo,
        @RequestParam(defaultValue = "10") @Min(10) int pageSize
  ) {
            // your existing code
}
```

Made with Goodnotes

Add Handler method to Controller Advice

```
@ExceptionHandler(HandlerMethodValidationException.class)
  public ResponseEntity<ErrorResponse> handleHandlerMethodValidationException (
      HandlerMethodValidationException exception, WebRequest request) {

    ErrorResponse errorResponse = new ErrorResponse("Invalid parameter(s)",
          HttpStatus.UNPROCESSABLE_ENTITY.value(),
          "Validation error. Check 'errors' field for details.", request.getDescription(false));

     List<ParameterValidationResult> paramNames = exception.getAllValidationResults();
     for (ParameterValidationResult param : paramNames) {
        errorResponse.addValidationError(
              param.getMethodParameter().getParameterName(),
              param.getResolvableErrors().get(0).getDefaultMessage() + " ("+
              param.getArgument().toString()+ ")");
     }
     return ResponseEntity.unprocessableEntity().body(errorResponse);
  }
```

Made with Goodnotes

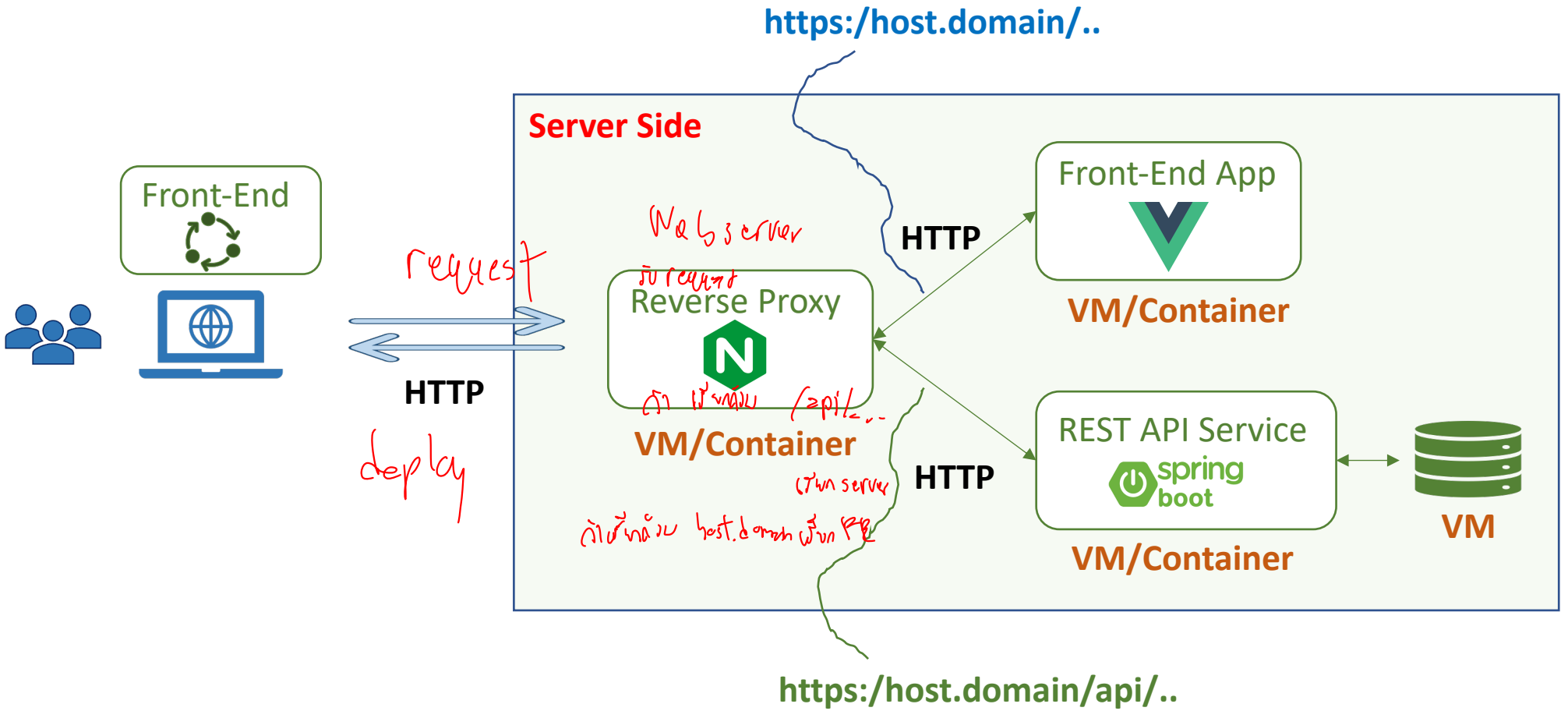
Spring RESTful API File Services

By

Pichet Limvajiranan

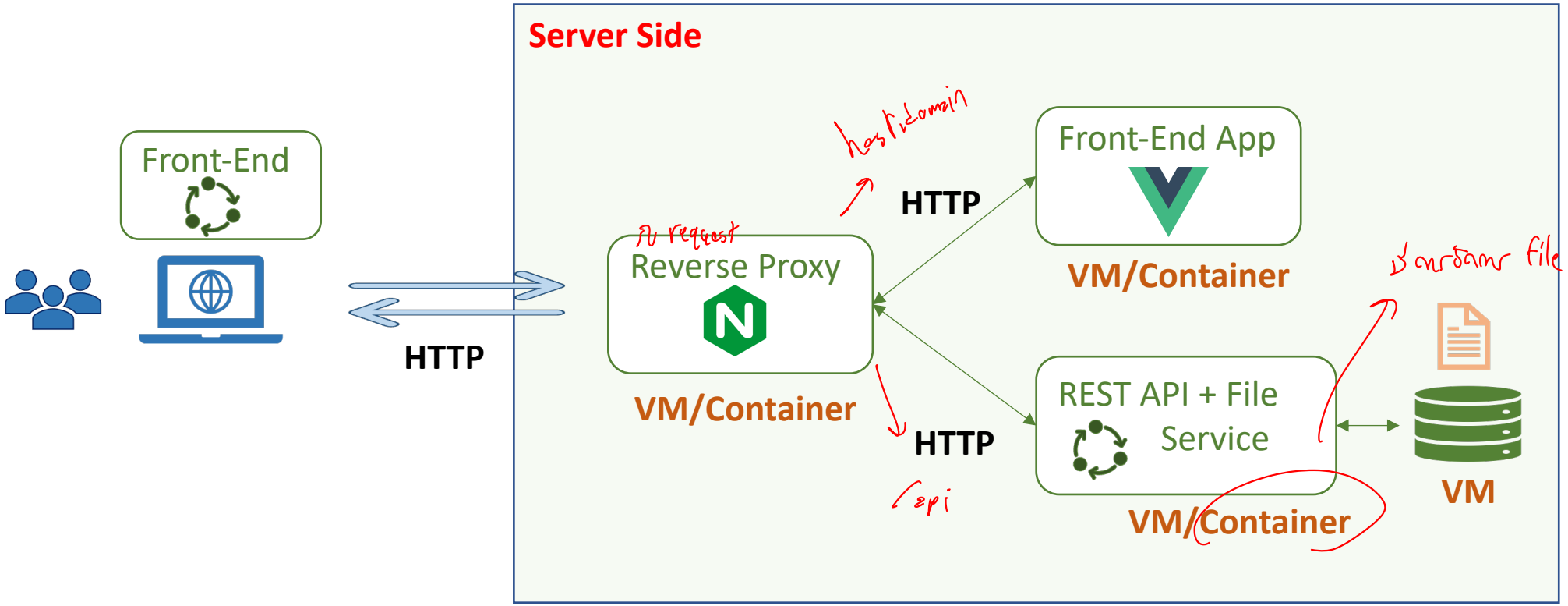
Made with Goodnotes

SPA Running Environment



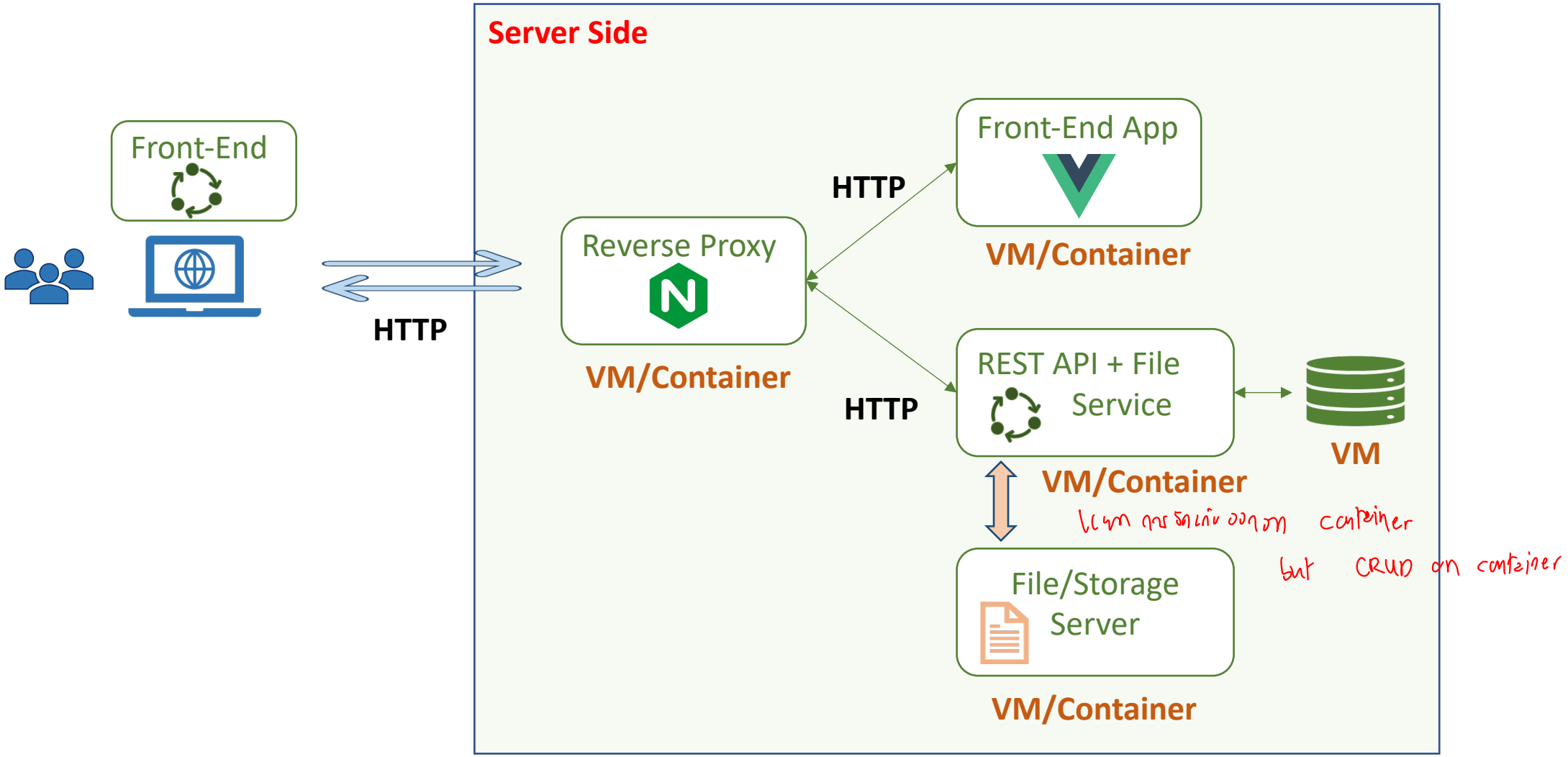
Made with Goodnotes

File Storage Architectures (1)



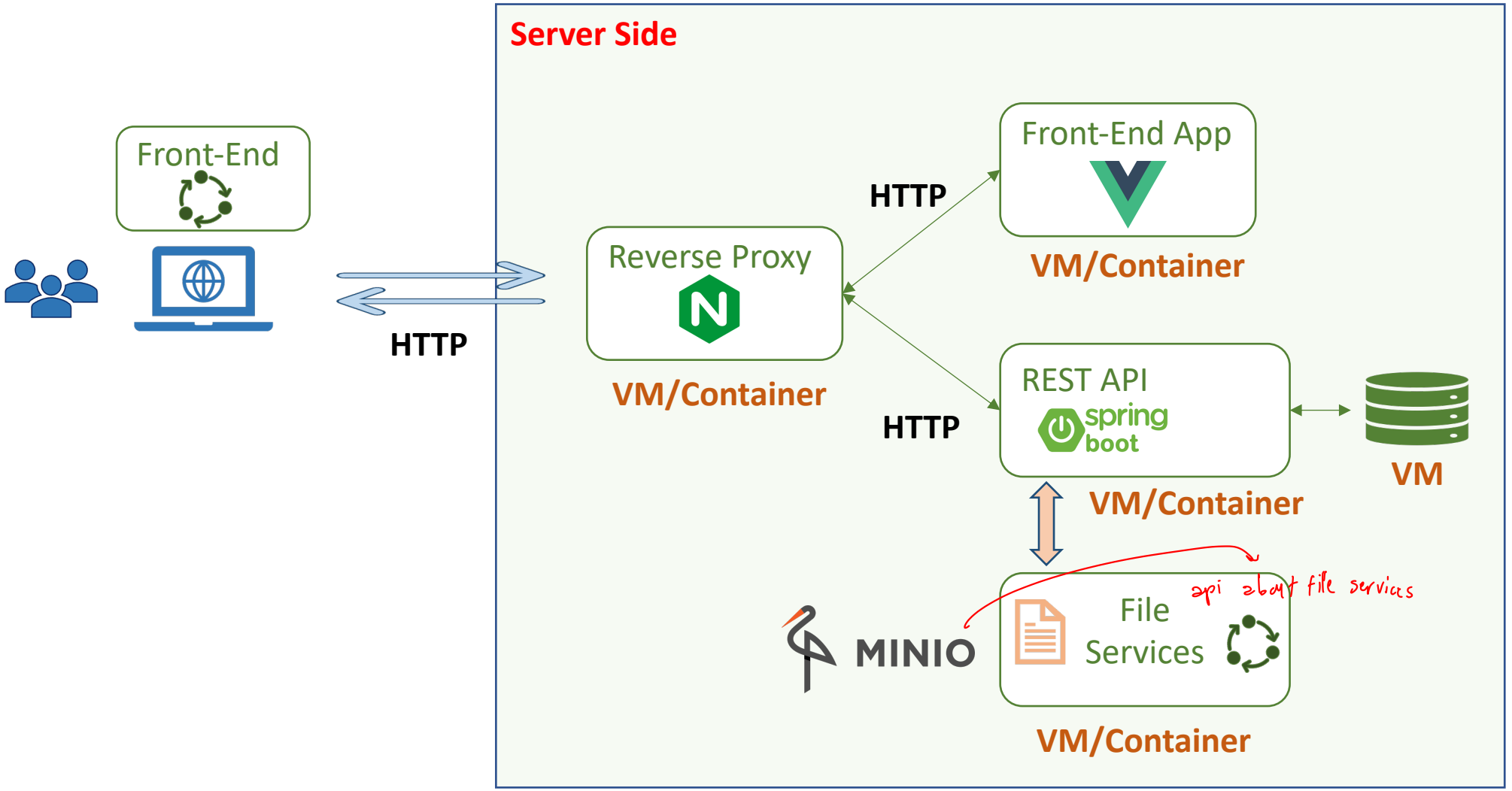
Made with Goodnotes

File Storage Architectures (2)



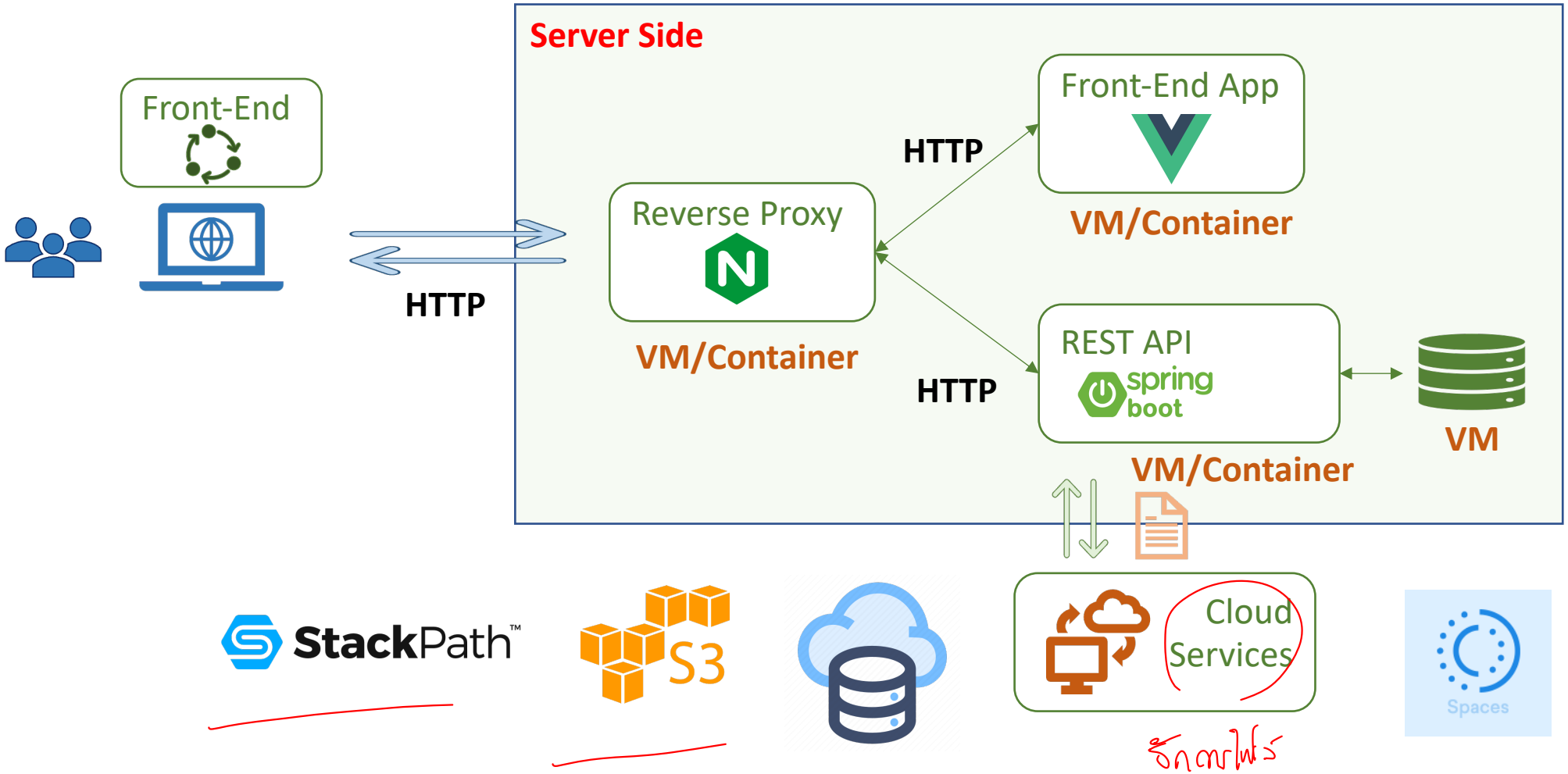
Made with Goodnotes

File Storage Architectures (3)



Made with Goodnotes

File Storage Architectures (4)



Made with Goodnotes

File Operation

- Upload
  - Send file – Form Data (Multipart) / File Only (FE)
  - **Read file data from Http Request (BE)**
  - **Write file to storage (BE)**
- Download
  - Return file URL / File content
- Delete
  - Read file name from Http Request
  - Remove file from storage

Made with Goodnotes

Configuring Server and File Storage Properties

```
## MULTIPART (MultipartProperties)
# Enable multipart uploads
spring.servlet.multipart.enabled=true

# Max file size.
spring.servlet.multipart.max-file-size=10MB

# Max Request Size
spring.servlet.multipart.max-request-size=80MB

## File Storage Properties # All files uploaded through the REST API will be stored in this directory
file.upload-dir=/public/classicmodels/uploads

# Threshold after which files are written to disk. spring.servlet.multipart.file-size-threshold=4KB

## File Storage Properties # All files uploaded through the REST API will be stored in this directory
file.upload-dir=./product-images
```

Made with Goodnotes

Automatically binding properties to a POJO class

- Spring Boot has an awesome feature called @ConfigurationProperties using which you can automatically bind the properties defined in the application.properties file to a POJO class.
- Let's define a POJO class called FileStorageProperties inside based-package.properties package to bind all the file storage properties.

```
file.upload-dir=./product-images
```

```
@ConfigurationProperties(prefix = "file")
@Getter
@Setter
public class FileStorageProperties {
  private String uploadDir;
}
```

```
@SpringBootApplication
@EnableConfigurationProperties({
      FileStorageProperties.class
})
public class DemoApplication {
   public static void main(String[] args) {
```

Made with Goodnotes

Binding properties to a POJO class

Create Configuration property class: FileStorageProperties.java (package based-package.properties

```
@ConfigurationProperties(prefix = "file")
@Getter
@Setter
public class FileStorageProperties {
  private String uploadDir;
}
```

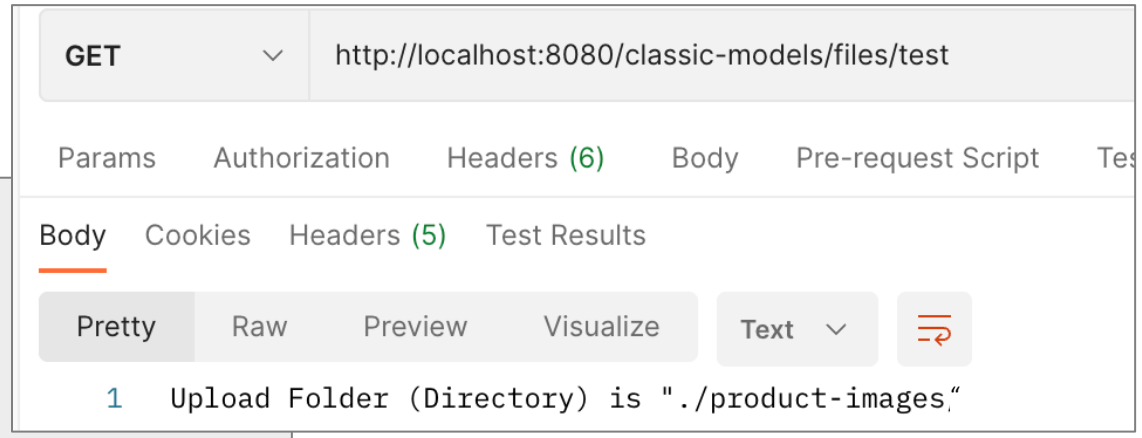
Register Configuration Property (into ApplicationConfig.java or ClassicmodelServiceApplication.java)

```
@EnableConfigurationProperties({
        FileStorageProperties.class
})
```

Made with Goodnotes

Test properties/POJO binding

```java
@RestController
@RequestMapping("/files")
public class FileController {
    @Autowired
    FileStorageProperties fileStorageProperties;

    @GetMapping("/test")
    public ResponseEntity<Object> testPropertiesMapping() {
        return ResponseEntity.ok("Upload Folder (Directory) is \""
                + fileStorageProperties.getUploadDir()+ "\"");
    }
}
```



Made with Goodnotes

FileService : Initialize

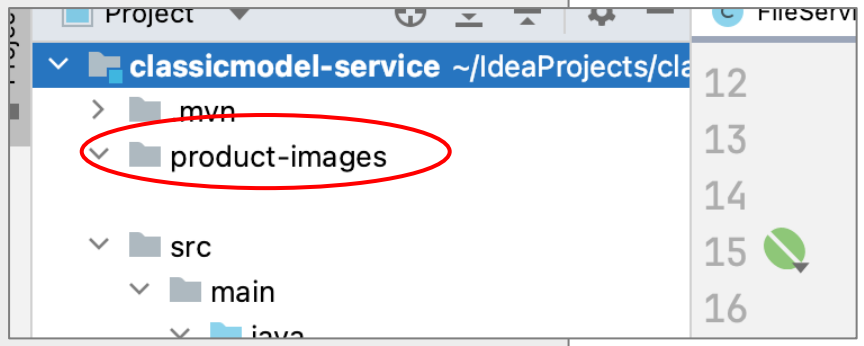
```
import java.io.IOException;
import java.net.MalformedURLException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.StandardCopyOption;
```

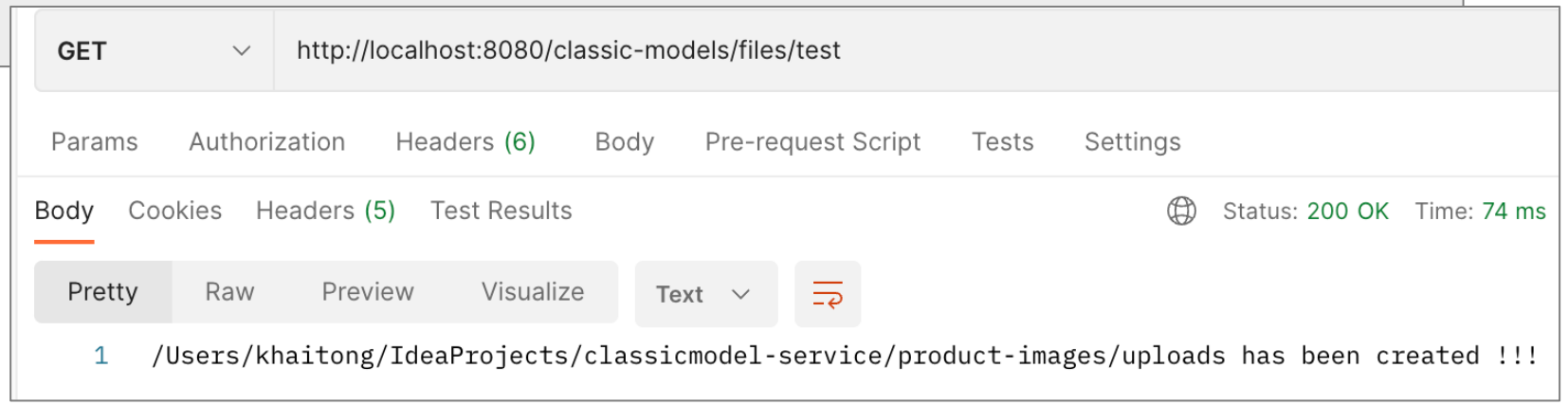
```
@Service
@Getter
public class FileService {
   private final Path fileStorageLocation;
   @Autowired
   public FileService(FileStorageProperties fileStorageProperties) {
      this.fileStorageLocation = Paths.get(fileStorageProperties
            .getUploadDir()).toAbsolutePath().normalize();
      try {
         if (! Files.exists(this.fileStorageLocation)) {
               Files.createDirectories(this.fileStorageLocation);
         }
      } catch (IOException ex) {
         throw new RuntimeException(
          "Could not create the directory where the uploaded files will be stored.", ex);
      }
   }
```

Made with Goodnotes

Modify FileController to Test FileService - Initialize

```java
@RestController
@RequestMapping("/files")
public class FileController {
    @Autowired
    FileService fileService;

    @GetMapping("/test")
    public ResponseEntity<Object> testPropertiesMapping() {
        return ResponseEntity.ok(fileService.getFileStorageLocation()+ " has been created !!!");
    }
}
```





Made with Goodnotes

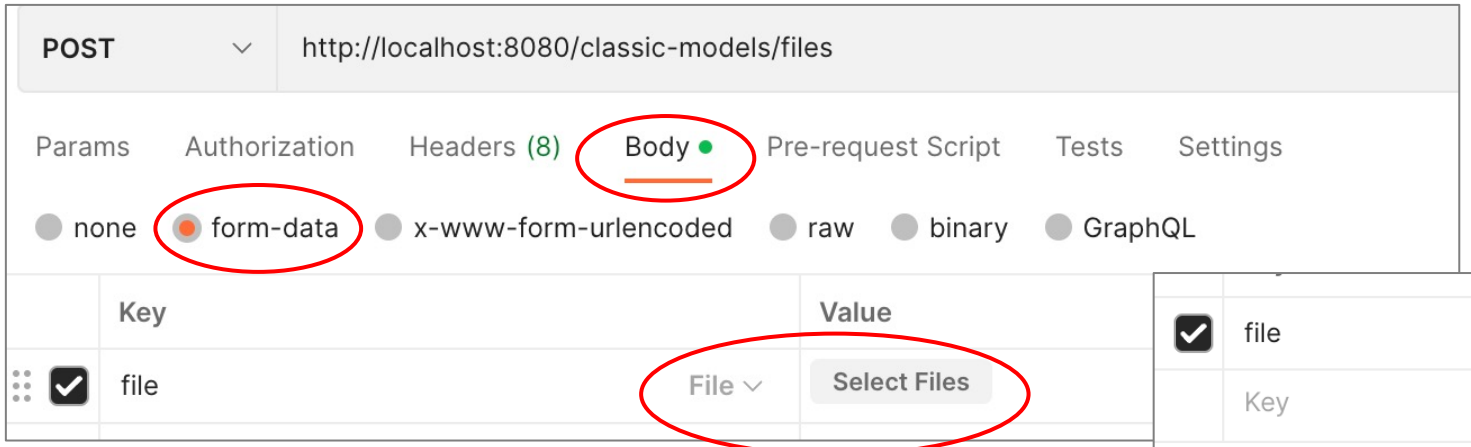
FileService: Storing File (add this method to FileService)

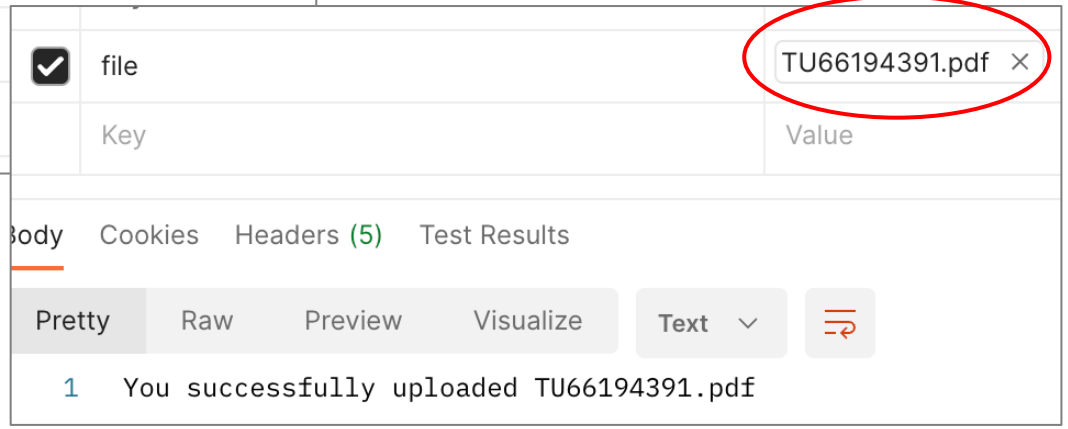
```
public String store(MultipartFile file) {
    // Normalize file name
    String fileName = StringUtils.cleanPath(file.getOriginalFilename());
    try {
        // Check if the file's name contains invalid characters
        if (fileName.contains("..")) {
            throw new RuntimeException("Sorry! Filename contains invalid path sequence " + fileName);
        }
        // Copy file to the target location (Replacing existing file with the same name)
        Path targetLocation = this.fileStorageLocation.resolve(fileName);
        Files.copy(file.getInputStream(), targetLocation, StandardCopyOption.REPLACE_EXISTING);
        return fileName;
    } catch (IOException ex) {
        throw new RuntimeException("Could not store file " + fileName + ". Please try again!", ex);
    }
}
```

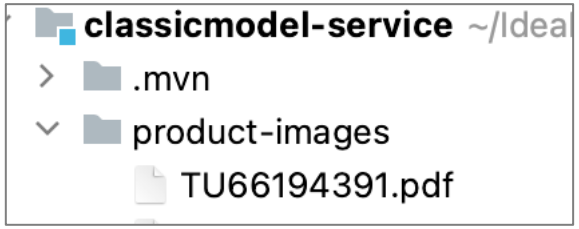
Made with Goodnotes

Add this method to FileController - Test FileService – Storing File

```
@PostMapping("")
public ResponseEntity<Object> fileUpload(@RequestParam("file") MultipartFile file) {
   fileService.store(file);
   return ResponseEntity.ok("You successfully uploaded " + file.getOriginalFilename());
}
```







Made with Goodnotes

FileService: LoadFile

```
public Resource loadFileAsResource(String fileName) {
    try {
        Path filePath = this.fileStorageLocation.resolve(fileName).normalize();
        Resource resource = new UrlResource(filePath.toUri());
        if (resource.exists()) {
            return resource;
        } else {
            throw new RuntimeException("File not found " + fileName);
        }
    } catch (MalformedURLException ex) {
        throw new RuntimeException("File operation error: " + fileName, ex);
    }
}
```

Made with Goodnotes

File Controller

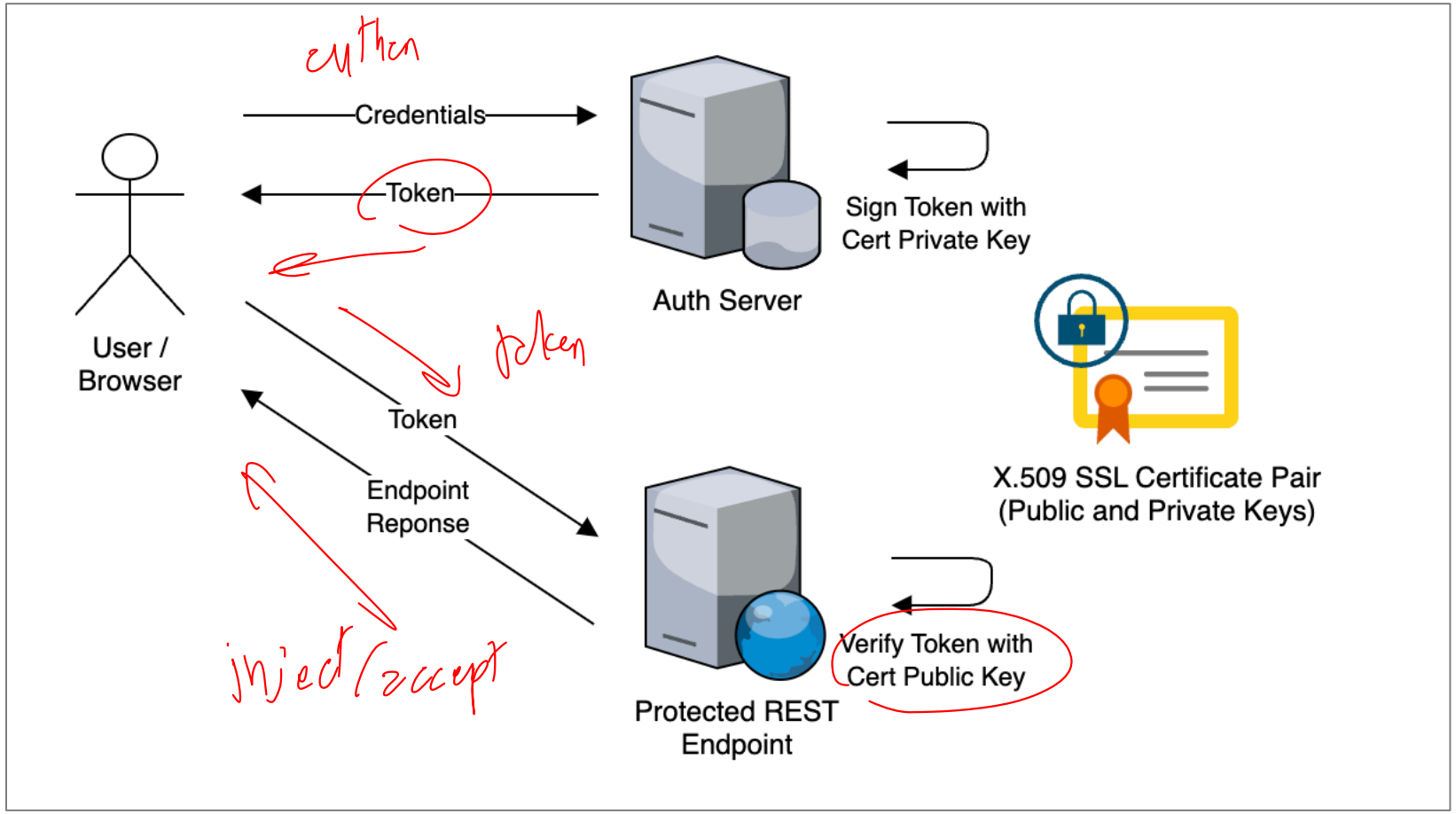
```
@GetMapping("/{filename:.+}")
@ResponseBody
public ResponseEntity<Resource> serveFile(@PathVariable String filename) {
    Resource file = fileService.loadFileAsResource(filename);
    return  ResponseEntity.ok().contentType(MediaType.IMAGE_JPEG).body(file);
}
```

Made with Goodnotes

Spring Boot
JWT-based Authentication

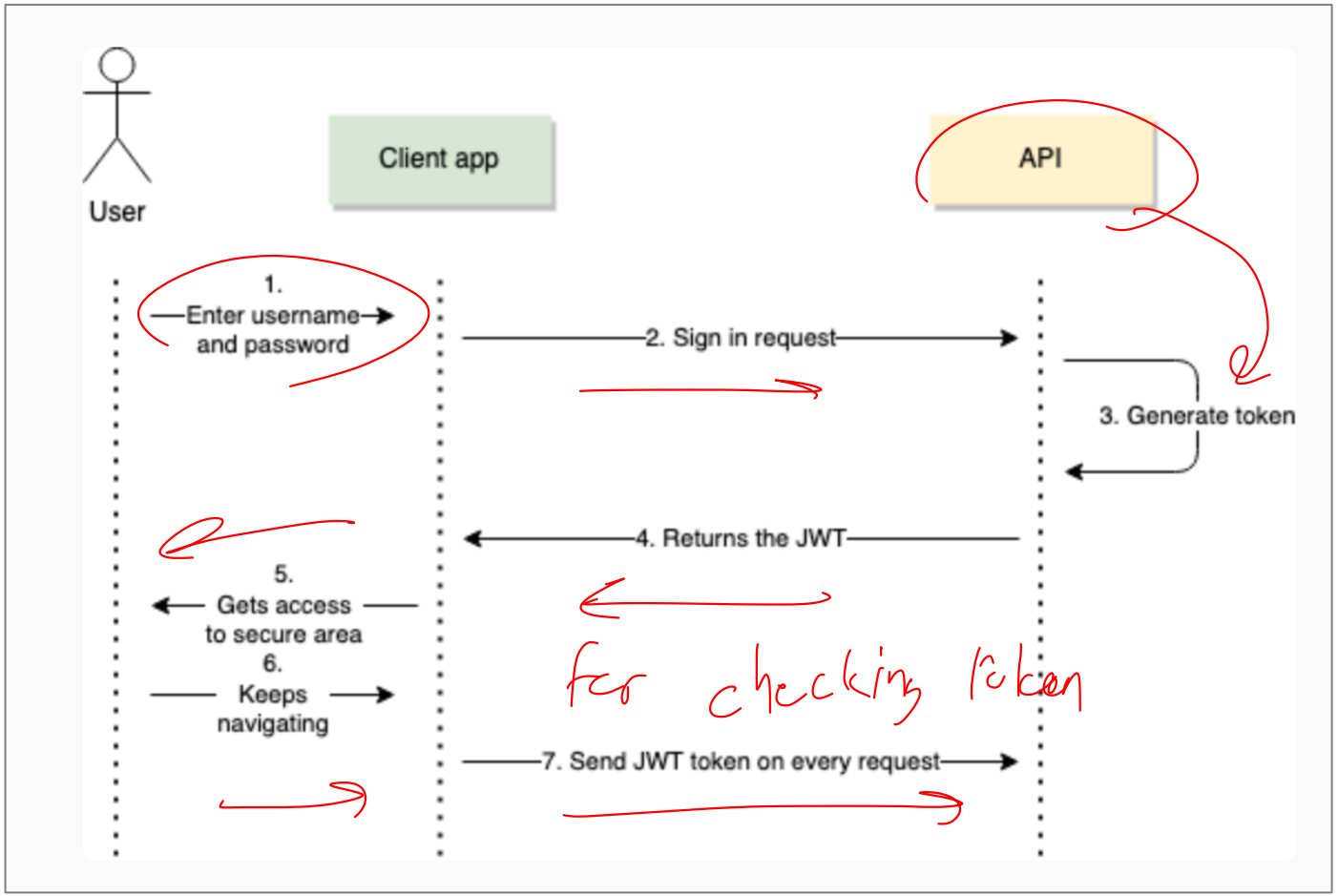
Made with Goodnotes

JWT-based Authentication



Made with Goodnotes

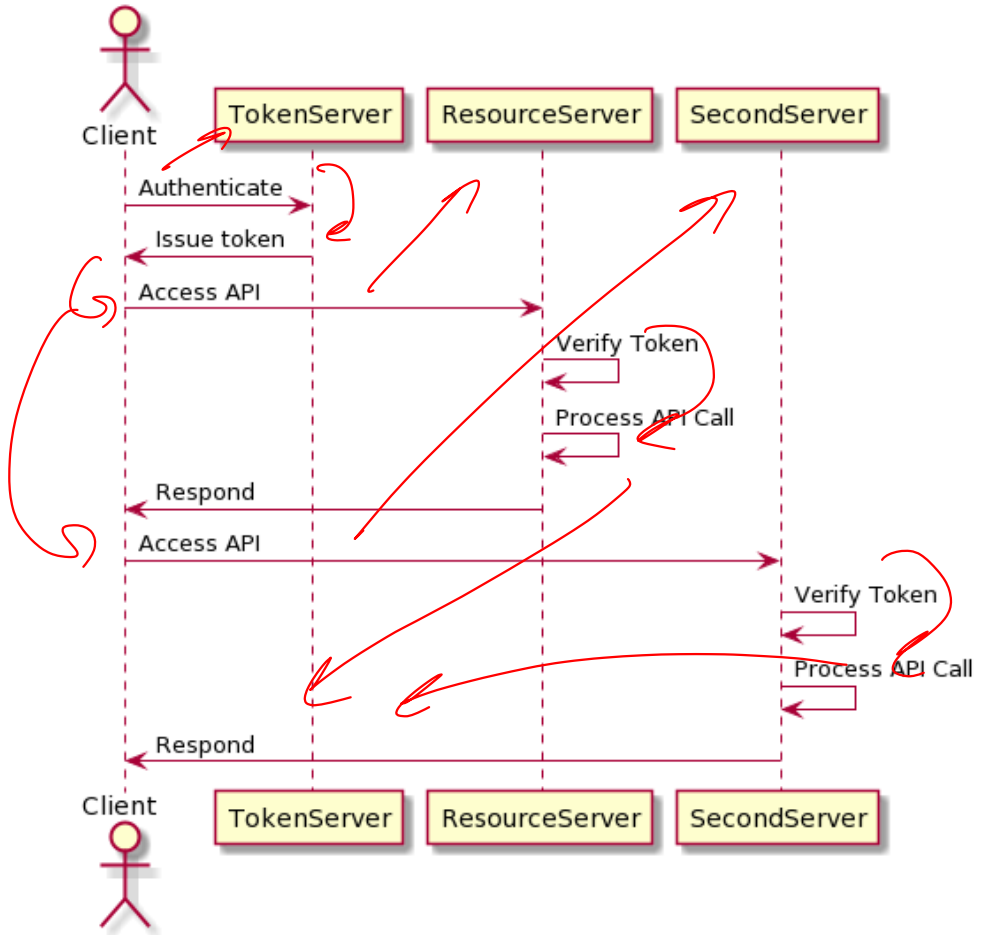
JWT-based Authentication



Made with Goodnotes

Session based vs Token based

- Scalability/Micro Service
  - A session often lives on a server or a cluster of servers.
- Some data that needs to be remembered across various parts of a website.



Made with Goodnotes

What is JSON Web Token?

- JSON Web Token (JWT) is an open standard (RFC 7519) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object.
- This information can be verified and trusted because it is digitally signed.
- JWTs can be signed using a secret (with the HMAC algorithm) or a public/private key pair using RSA or ECDSA.
- Although JWTs can be encrypted to also provide secrecy between parties, we will focus on signed tokens.
- Signed tokens can verify the integrity of the claims contained within it, while encrypted tokens hide those claims from other parties.
- When tokens are signed using public/private key pairs, the signature also certifies that only the party holding the private key is the one that signed it.

Made with Goodnotes

What is the JSON Web Token structure?

- In its compact form, JSON Web Tokens consist of three parts separated by dots (.), which are:
  - Header
    - The header typically consists of two parts: the type of the token, which is JWT, and the signing algorithm being used, such as HMAC SHA256 or RSA.
  - Payload
    - Which contains the claims. Claims are statements about an entity (typically, the user) and additional data.
  - Signature
    - The signature is used to verify the message wasn't changed along the way, and, in the case of tokens signed with a private key, it can also verify that the sender of the JWT is who it says it is.
  - Putting all together

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4
gRG9lIiwiaXNTb2NpYWwiOnRydWV9.
4pcPyMD09olPSyXnrXCjTwXyr4BsezdI1AVTmud2fU4
```

Made with Goodnotes

JSON Web Token structure

Encoded PASTE A TOKEN HERE

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.ey
JzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6Ikpva
G4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKx
wRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c
```

```
eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJh
QGdtYWkuY29tIiwiaWF0IjoxNzExM
zQzNzM1LCJleHAiOjE3MTEzNTA5M
zV9.nrKL5p1b2Q_lrs1d2dPvJoIzW5
4-YihoEdCoZhBDbcc
```

Online Decoder: https://jwt.io

Decoded

HEADER: ALGORITHM & TOKEN TYPE

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

PAYLOAD: DATA

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "iat": 1516239022
}
```

VERIFY SIGNATURE

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  your-256-bit-secret
) ☐ secret base64 encoded
```

Made with Goodnotes

How do JSON Web Tokens work?

- In authentication, when the user successfully logs in using their credentials, a JSON Web Token will be returned. Since tokens are credentials, great care must be taken to prevent security issues. In general, you should not keep tokens longer than required.
- Whenever the user wants to access a protected route or resource, the user agent should send the JWT, typically in the Authorization header using the Bearer schema. The content of the header should look like the following:

```
Authorization: Bearer <token>
```

Made with Goodnotes

Spring Boot JWT Example

Made with Goodnotes

Spring security

- Spring Security is a framework that enables a programmer to impose security restrictions to Spring-framework–based Web applications through JEE components.
- Its primary area of operation is to handle authentication and authorization at the Web request level as well as the method invocation level.
- The greatest advantage of this framework is that it is powerful yet highly customizable in its implementation.
- Although it follows Spring's convention over configuration, programmers can choose between default provisions or customizing them according to their needs.

Made with Goodnotes

Authentication

- The form of key, Grant the user who have the proper credentials
- Challenges the user to validate credentials (for example, through passwords, answers to security questions, or facial recognition)
- Usually done before authorization. First, we verify the user then check other things.
- For example, Employees in a company are required to authenticate through the network before accessing their company email

Made with Goodnotes

Authorization

- Determines what users can and cannot access
- In the form of permissions. In this term, only denotes validate the access permission.
- Verifies whether access is allowed through policies and rules
- Usually done after successful authentication

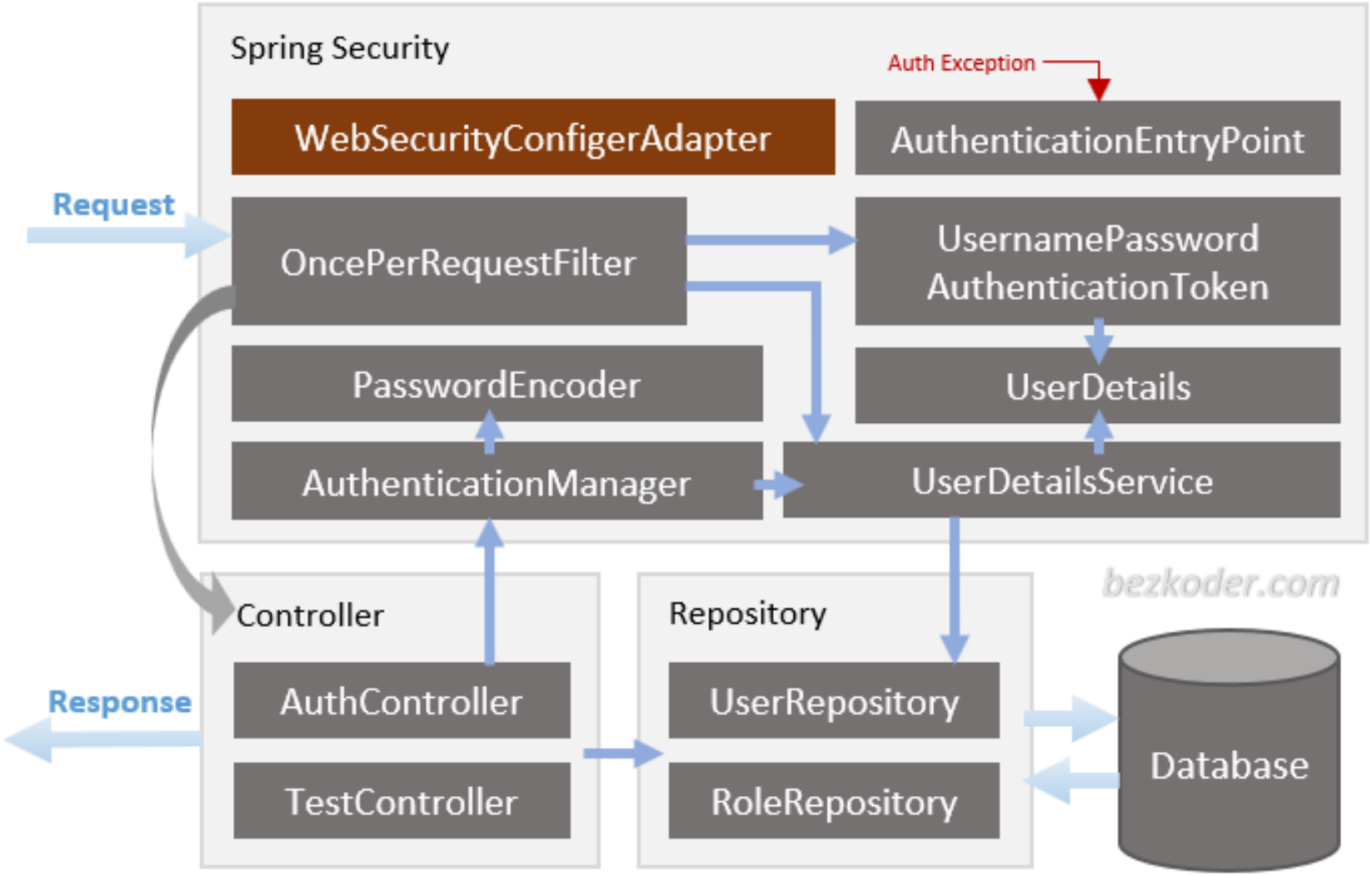
Made with Goodnotes

Best Practices for Securing REST APIs

- Use HTTPS all the time: With the use of SSL, all the authentication credentials can be cut down to an arbitrarily produced access-token, which uses the HTTP Basic Auth technique.
- Use Hashed Password: hashing of the password is vital to shield RESTful services because even when your password gets compromised by hackers in a hacking attempt, they will not be able to read them out. Various hashing algorithms make this approach a fruitful one. Some of them are MD5, PBKDF2, bcrypt, SHA algorithms, etc.
- Considering OAuth: If the basic auth is implemented to most of the APIs correctly, then it is a great choice, which is more secure also. With the introduction of the OAuth 2.0 authorization framework, all third-party applications get enabled to attain the partial right of entry to HTTP service(s).
- Validating Input Parameter: Security can be well executed of the request parameters get validated in the very beginning, before reaching in the application logic.

Made with Goodnotes

Spring Security



Made with Goodnotes

Dependencies

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-security</artifactId>
   <version>3.2.2</version>
</dependency>
```

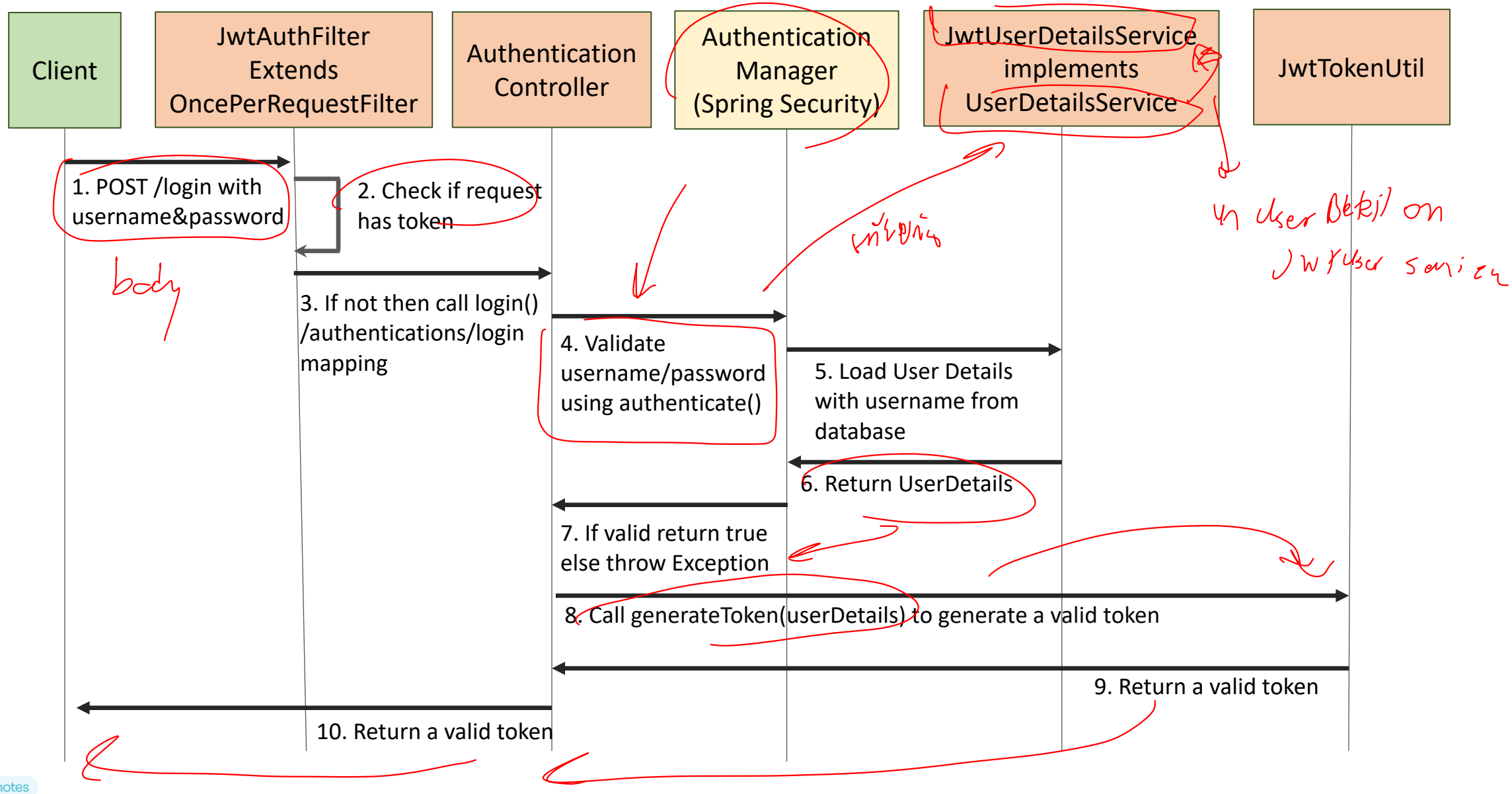
```
<dependency>
   <groupId>de.mkammerer</groupId>
   <artifactId>argon2-jvm</artifactId>
   <version>2.11</version>
</dependency>
```

```
<dependency>
   <groupId>org.bouncycastle</groupId>
   <artifactId>bcprov-jdk15on</artifactId>
   <version>1.64</version>
</dependency>
```

```
<dependency>
   <groupId>io.jsonwebtoken</groupId>
   <artifactId>jjwt-api</artifactId>
   <version>0.11.5</version>
</dependency>
<dependency>
   <groupId>io.jsonwebtoken</groupId>
   <artifactId>jjwt-impl</artifactId>
   <version>0.11.5</version>
   <scope>runtime</scope>
</dependency>
<dependency>
   <groupId>io.jsonwebtoken</groupId>
   <artifactId>jjwt-jackson</artifactId>
   <version>0.11.5</version>
   <scope>runtime</scope>
</dependency>
```

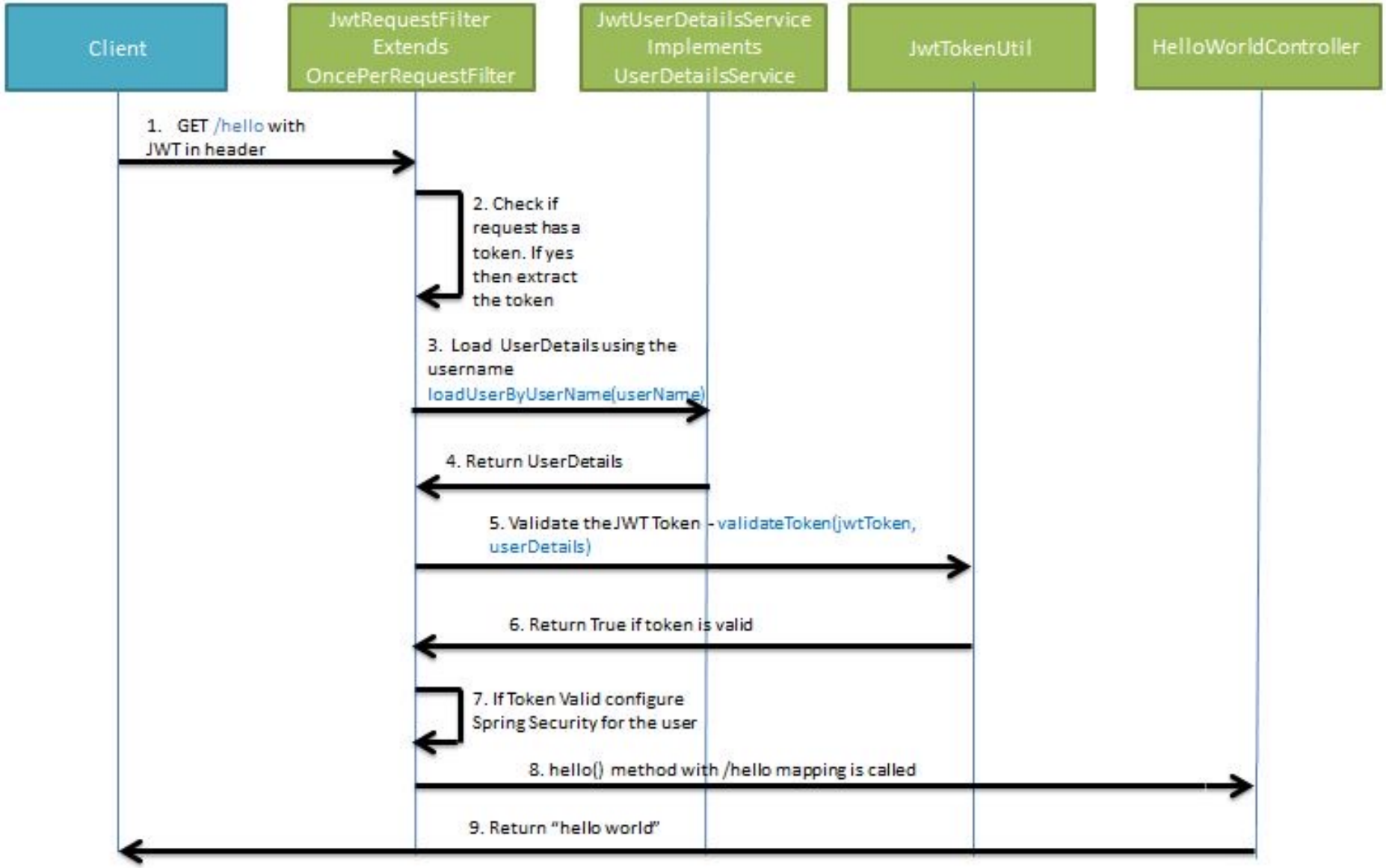
Made with Goodnotes

Generating JWT – Sequence diagram



Made with Goodnotes

Request Resource – Sequence Diagram



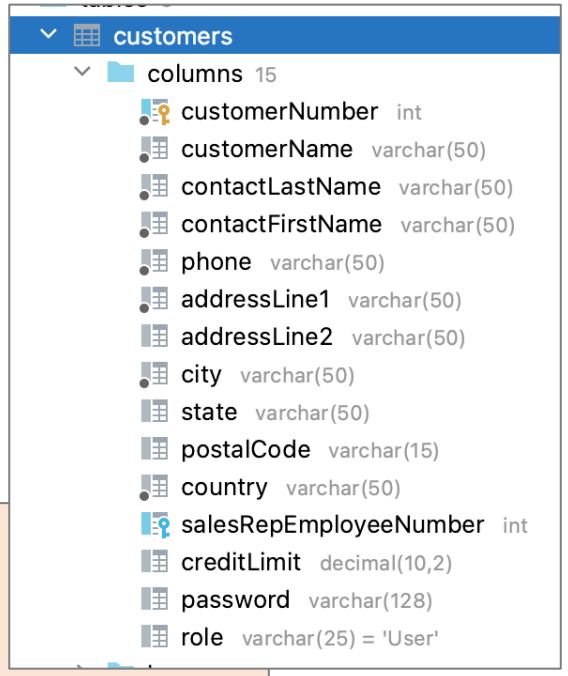
Made with Goodnotes

Implementations - Setting

1. Add dependencies
2. Preparing database
3. Edit Customer entity
4. Edit Customer repositoy
5. Create WebSecurityConfig.java

password	role
00 e2a1c23b6d431b840327ce8233…	USER
00 $argon2d$v=19$m=16,t=2,p=1…	ADMIN
00 e2a1c23b6d431b840327ce8233…	USER
00 $argon2d$v=19$m=16,t=2,p=1…	USER
00 $argon2d$v=19$m=16,t=2,p=1…	USER

https://argon2.online




```
package sit.int204.classicmodelservice.config;

@Configuration
@EnableWebSecurity
public class WebSecurityConfig {
    @Bean
    public SecurityFilterChain filterChain(HttpSecurity httpSecurity) throws Exception {
        httpSecurity.csrf(csrf -> csrf.disable())
                .authorizeRequests(authorize -> authorize.requestMatchers("/authentications/**").permitAll()
                        .anyRequest().authenticated()
                )
                .httpBasic(withDefaults());
        return httpSecurity.build();
    }
```

```
public interface CustomerRepository extends JpaRepository<Customer, Integer> {
    @Query("select c from Customer c where concat(c.contactFirstName,' ',c.contactLastName) = :name")
    Customer findByName(String name);
}
```

Made with Goodnotes

Implementations (2)

6. Create JwtRequestIUser.java
7. Create Spring Security User AuthUser.java

```
package sit.int204.classicmodelservice.dtos;

@Data
public class JwtRequestUser {
    @NotBlank
    private String userName;
    @Size(min = 8)
    @NotBlank
    private String password;
}
```

```
@Getter
@Setter
public class AuthUser extends User implements Serializable {
    public AuthUser() {
        super("anonymous", "", new ArrayList<GrantedAuthority>());
    }
    public AuthUser(String userName, String password) {
        super(userName, password, new ArrayList<GrantedAuthority>());
    }

    public AuthUser(String userName, String password, Collection<? extends
            GrantedAuthority> authorities) {
        super(userName, password, authorities);
    }
}
```

Made with Goodnotes

Implementations (3) - Create UserDetailsService: JwtUserDetailsService.java

```
@Service
public class JwtUserDetailsService implements UserDetailsService {
   @Autowired
   private CustomerRepository customerRepository;
   @Override
   public UserDetails loadUserByUsername(String userName) throws UsernameNotFoundException {
      Customer customer = customerRepository.findByName(userName);
      if(customer == null) {
         throw new ResponseStatusException(HttpStatus.NOT_FOUND, userName+ " does not exist !!");
      }
      List<GrantedAuthority> roles = new ArrayList<>();
      GrantedAuthority grantedAuthority = new GrantedAuthority() {
         @Override
         public String getAuthority() {
            return customer.getRole();
         }
      };
      roles.add(grantedAuthority);
      UserDetails userDetails = new AuthUser(userName, customer.getPassword(), roles);
      return userDetails;
   }
}
```

Made with Goodnotes

Implementations (4) – Create JwtHelper: JwtTokenUtil.java (1/2)

application.properties

```
jwt.secret=N7KgseMPtJ26AEved0ahUKEwj4563eioyFAxUyUGwGHbTODx0Q4dUDCBA
jwt.max-token-interval-hour=2
```

```
@Component
public class JwtTokenUtil implements Serializable {
   @Value("${jwt.secret}")
   private String SECRET_KEY;
   @Value("#{${jwt.max-token-interval-hour}*60*60*1000}")
   private long JWT_TOKEN_VALIDITY;
   SignatureAlgorithm signatureAlgorithm = SignatureAlgorithm.HS256;

   public String getUsernameFromToken(String token) {
      return getClaimFromToken(token, Claims::getSubject);
   }
   public Date getExpirationDateFromToken(String token) {
      return getClaimFromToken(token, Claims::getExpiration);
   }
   public <T> T getClaimFromToken(String token, Function<Claims, T> claimsResolver) {
     final Claims claims = getAllClaimsFromToken(token);
      return claimsResolver.apply(claims);
   }
   public Claims getAllClaimsFromToken(String token) {
      Claims claims = Jwts.parser().setSigningKey(SECRET_KEY)
            .parseClaimsJws(token).getBody();
      return claims;
   }
```

Made with Goodnotes

Implementations (4) – Create JwtHelper: JwtTokenUtil.java (2/2)

```
  private Boolean isTokenExpired(String token) {
      final Date expiration = getExpirationDateFromToken(token);
      return expiration.before(new Date());
   }
   public String generateToken(UserDetails userDetails) {
      Map<String, Object> claims = new HashMap<>();
      claims.put("info#1", "claim-objec 1");
      claims.put("info#2", "claim-objec 2");
      claims.put("info#3", "claim-objec 3");
      return doGenerateToken(claims, userDetails.getUsername());
   }
   private String doGenerateToken(Map<String, Object> claims, String subject) {
      return Jwts.builder().setHeaderParam("typ", "JWT").setClaims(claims).setSubject(subject)
            .setIssuedAt(new Date(System.currentTimeMillis()))
            .setExpiration(new Date(System.currentTimeMillis() + JWT_TOKEN_VALIDITY))
            .signWith(signatureAlgorithm, SECRET_KEY).compact();
   }
   public Boolean validateToken(String token, UserDetails userDetails) {
      final String username = getUsernameFromToken(token);
      return (username.equals(userDetails.getUsername()) && !isTokenExpired(token));
   }
}
```

Made with Goodnotes

Implementations (5) - Create AuthenticationController.java (Test-1)

```
@RestController
@RequestMapping("/authentications")
public class AuthenticationController {
    @Autowired
    JwtUserDetailsService jwtUserDetailsService;

    @Autowired
    JwtTokenUtil jwtTokenUtil;

    @PostMapping("/login")
    public ResponseEntity<Object> login(@RequestBody @Valid JwtRequestUser jwtRequestUser) {
        UserDetails userDetails = jwtUserDetailsService.loadUserByUserName(jwtRequestUser.getUserName());
        String token = jwtTokenUtil.generateToken(userDetails);
        return ResponseEntity.ok(token);
    }
}
```

Made with Goodnotes

Implementations (5) - Create AuthenticationController.java (Test-2)

```
@GetMapping("/validate-token")
public ResponseEntity<Object> validateToken(@RequestHeader("Authorization") String requestTokenHeader) {
   Claims claims = null;
   String jwtToken = null;
   if (requestTokenHeader != null && requestTokenHeader.startsWith("Bearer ")) {
      jwtToken = requestTokenHeader.substring(7);
      try {
           claims = jwtTokenUtil.getAllClaimsFromToken(jwtToken);
      } catch (IllegalArgumentException e) {
         System.out.println("Unable to get JWT Token");
      } catch (ExpiredJwtException e) {
         System.out.println("JWT Token has expired");
      }
   } else {
      throw new ResponseStatusException(HttpStatus.EXPECTATION_FAILED,
            "JWT Token does not begin with Bearer String");
   }
   return ResponseEntity.ok(claims);
}
```

Made with Goodnotes

JwtAuthFilter (1/2)

```
@Component
public class JwtAuthFilter extends OncePerRequestFilter {
   @Autowired    private JwtUserDetailsService jwtUserDetailsService;
   @Autowired.   private JwtTokenUtil jwtTokenUtil;
   @Override
   protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain chain)
         throws ServletException, IOException {
      final String requestTokenHeader = request.getHeader("Authorization");
      String username = null;
      String jwtToken = null;
      if (requestTokenHeader != null) {
         if (requestTokenHeader.startsWith("Bearer ")) {
            jwtToken = requestTokenHeader.substring(7);
            try {
               username = jwtTokenUtil.getUsernameFromToken(jwtToken);
            } catch (IllegalArgumentException e) {
               throw new ResponseStatusException(HttpStatus.BAD_REQUEST, e.getMessage());
            } catch (ExpiredJwtException e) {
               throw new ResponseStatusException(HttpStatus.BAD_REQUEST, e.getMessage());
            }
         } else {
            throw new ResponseStatusException(HttpStatus.BAD_REQUEST, "JWT Token does not begin with Bearer String");
         }
      }
```

Made with Goodnotes

JwtAuthFilter.java (2/2)

```
        if (username != null && SecurityContextHolder.getContext().getAuthentication() == null) {

            UserDetails userDetails = this.jwtUserDetailsService.loadUserByUsername(username);

            if (jwtTokenUtil.validateToken(jwtToken, userDetails)) {

                UsernamePasswordAuthenticationToken usernamePasswordAuthenticationToken = new
                 UsernamePasswordAuthenticationToken(userDetails, null, userDetails.getAuthorities());
                usernamePasswordAuthenticationToken
                        .setDetails(new WebAuthenticationDetailsSource().buildDetails(request));
                SecurityContextHolder.getContext().setAuthentication(usernamePasswordAuthenticationToken);
            }
        }
        chain.doFilter(request, response);
    }
}
```

Made with Goodnotes

Authentication with Spring AuthenticationManager (1/2)

```
@Configuration
@EnableWebSecurity
public class WebSecurityConfig {
   @Autowired private JwtUserDetailsService jwtUserDetailsService;
   @Autowired  private JwtAuthFilter jwtAuthFilter;
   @Bean
   public SecurityFilterChain filterChain(HttpSecurity httpSecurity) throws Exception {
      httpSecurity.csrf(csrf -> csrf.disable())
            .authorizeRequests(authorize -> authorize
                  .requestMatchers("/authentications/login").permitAll()
                  .requestMatchers("/authentications/validate-token").hasAuthority("ADMIN")
                  .anyRequest().authenticated()
            )
            .httpBasic(withDefaults());
      httpSecurity.addFilterBefore(jwtAuthFilter, UsernamePasswordAuthenticationFilter.class);
      return httpSecurity.build();
   }
```

Made with Goodnotes

Authentication with Spring AuthenticationManager (2/2)

```
@Bean
  public PasswordEncoder passwordEncoder() {
     return Argon2PasswordEncoder.defaultsForSpringSecurity_v5_8();
  }

  @Bean
  public AuthenticationProvider authenticationProvider() {
     DaoAuthenticationProvider authenticationProvider = new DaoAuthenticationProvider();
     authenticationProvider.setUserDetailsService(jwtUserDetailsService);
     authenticationProvider.setPasswordEncoder(passwordEncoder());
     return authenticationProvider;
  }

  @Bean
  public AuthenticationManager authenticationManager(AuthenticationConfiguration config) throws Exception {
     return config.getAuthenticationManager();
  }
}
```

Made with Goodnotes