1. Spring IOC

IOC (Inversion of Control):

- a programming principle we follow to achieve loose coupling, we transfer the control of an object or portions of a program to a container or framework. And there are three ways to achieve this.

    - dependency injection
        - We can move object binding from compile time to runtime by using the annotation @Autowired, and there are three ways to do this
            - Constructor based
                - We put @Autowired on the constructor
                - The most recommended way, it makes unit test easier, and guarantee that dependencies will be injected when the object is created
            - Setter based
                - We put @Autowired on the setter methods
                - Flexible, we can do partial injection when we want optional dependencies
            - Field based
                - We put @Autowired on the fields
                - Most convenient one, prevents cycle dependent issue, but could have security problem since we can access private field through reflection
    - factory design pattern
        - Factory pattern is a design pattern where we first create an interface, for example, a shape interface. Then we create different classes under this shape interface, ie. circle, triangle, square, etc. And then we create a factory class, which will output different objects based on the input. Ie, we'll provide a shape object of a circle if the input is a circle. In the factory, the handler will be annotated with @Autowired
    - strategy design pattern
        - Strategy design pattern is a design pattern where we create a strategy interface, and we create different operation class based on the strategy interface, and then we'll create a context class to execute strategy base on the input

Advantages using IOC:

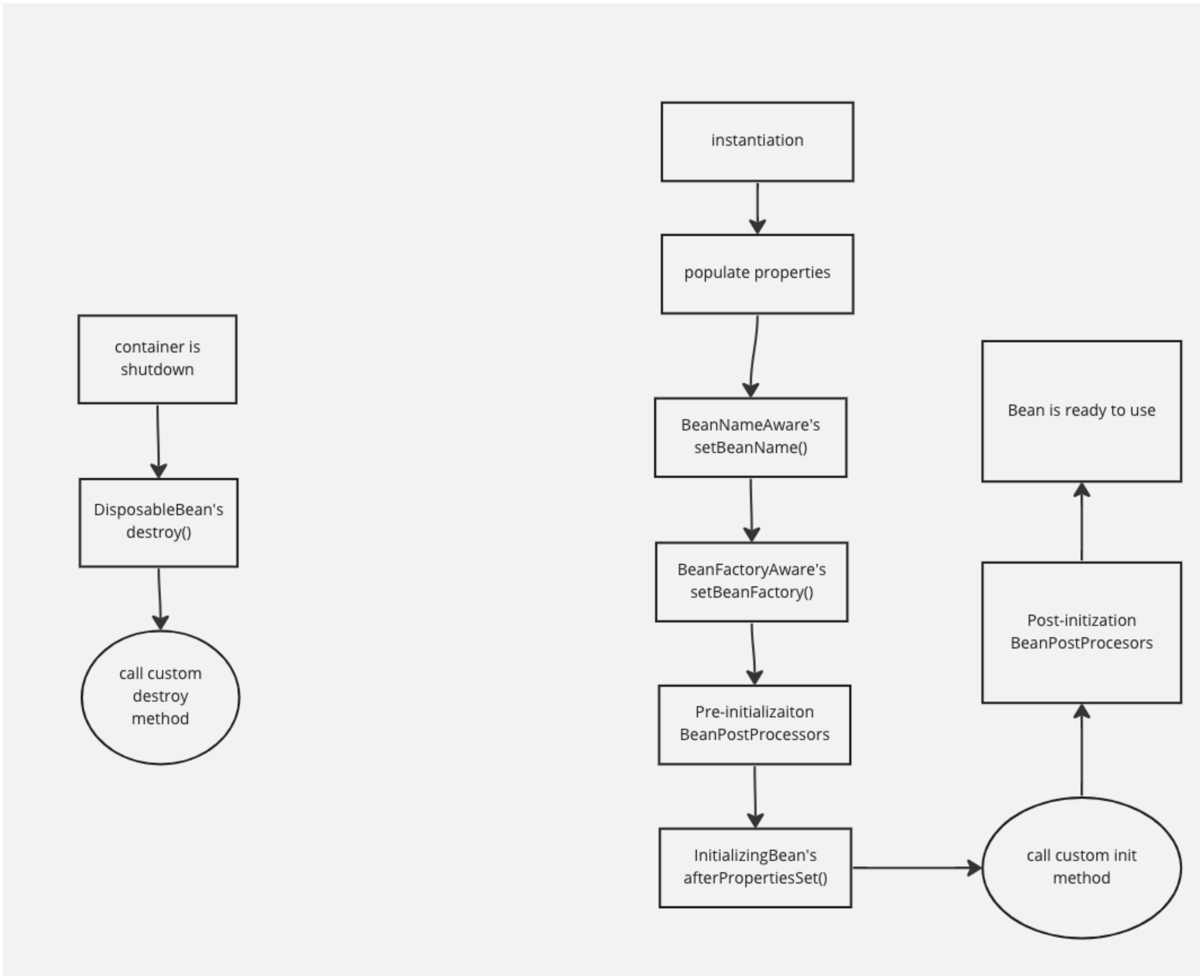
- decoupling the execution of task from its implementation
- making it easier to switch between different implementation
- greater ease in testing a programming by isolating a component or mocking its dependencies
- modularity of program

Bean Scope:

- For each bean, we can define the scope of the component using annotation @Scope("scope type")
  And there are seven different scope
  - singleton (default)
    - Spring IOC contain will automatically create a single component in the whole application when no specification is given
  - Prototype
    - If we want more than one component, we can specified the scope as prototype, it produces a new instance each and every time a bean is requested
  - Request
    - Create a new instance each time http request is received
  - Session
    - Create a new instance for each session
  - Application
    - Create a new instance for each servletContext
  - webSocket
    - Create a new instance for each Socket
  - Thread
    - Create a new instance for each Thread

Bean Lifecycle

- Bean life cycle is controlled by the spring container. When we start the program, the spring container gets started. Then the spring container will create the instance of a bean and dependencies are injected. Finally, the bean will be destroyed when the spring container is closed. If we want to execute some code on the bean instantiation and just after closing the spring container, we can provide our customer init() and destroy() methods.


instantiation
populate properties
BeanNameAware's setBeanName()
BeanFactoryAware's setBeanFactory()
Pre-initializaiton BeanPostProcessors
InitializingBean's afterPropertiesSet()
call custom init method
Post-initization BeanPostProcesors
Bean is ready to use
container is shutdown
DisposableBean's destroy()
call custom destroy method

We can use the following annotation to invoke methods after bean is created or before it's destroy

@PostConstruct

- annotated method will be invoked after the bean has been constructed

@PreDestroy

- annotated method is invoked just before the bean is about be destroyed inside the bean container

2. Spring AOP

AOP(Aspect-Oriented Programming)

- The goal of AOP is to increase modularity, in AOP, aspects enables the modularization of concerns such as transaction management, logging, security that cut across multiple types and object

Aspect:

- an aspect is a class that implements enterprise application concerns that cut across multiple classes. Such as transaction management, logging, security
- In order to make a class aspect, we need to mark the class using the annotation @Aspect

Advice

- Before
  - advice that executes before the join point
- After
  - advice that executes after the join point
- After Return
  - advice that executes after any return of the method
- After Throw
  - advice that executes after an exception is throw
- Around
  - advice that executes before and after the join point

JoinPoint

- a point during the execution of a program, such as the execution of a method or handling an exception

PointCut

- a predicate that matches the join point

- advice is associated with a pointcut expression and runs at any join point matched by the pointcut

Target

- object to be advised

@Transactional

- It’s a annotation we use it to wrap up a method in database transaction
- When spring detects @transaction on a bean, it will create a dynamic proxy for that bean
- It allows us to set propagation, isolation, timeout, read-only, and rollback conditions for our transaction. We can also specify the transaction manager
- It’s good for maintaining transactions, when exceptions happen during runtime, it’s automatically roll back
- The **propagation** attribute indicates if any component or service will participate in the transaction and how it will behave if the calling component or service already has or does not have an existing transaction. The default value is **REQUIRED** which means it will use the existing transaction created by the caller or it will create a new transaction if it doesn’t exist.

- Other available settings are:
  - **SUPPORTS**
    - uses the existing transaction of the caller, never creates a new transaction.
  - **NOT_SUPPORTED**
    - runs without a transaction and never creates a new transaction. Caller’s existing transaction will be suspended.
  - **REQUIRES_NEW**
    - always creates a new transaction. Caller’s existing transaction will be suspended.
  - **NEVER**
    - never creates a new transaction and **throws an exception (IllegalTransactionStateException)** if the caller has an existing transaction.
  - **MANDATORY**
    - always use the caller’s existing transaction. **Throws an exception** if the caller does not have an existing transaction.

- The default **rollback** behavior without explicit declaration will rollback on throwed **runtime unchecked** exceptions. (The checked exceptions does not trigger a rollback)

This can also be controlled by using the attributes **rollbackFor** and **rollbackForClassName**, and **noRollbackFor** or **noRollbackForClassName** to avoid rollback on listed exceptions.

When we define @Transactional(**timeout** = [int here]) it means that our transaction should complete in the given time frame(in milliseconds) otherwise we will get a **TransactionException**. By default it's -1 which means no timeout at all.

ReadOnly: A boolean flag that can be set to true if the transaction is effectively read-only. Defaults to **false**
Note:
When there're write attempts in the transaction, it won't cause any failure. It's just used to give a hint to the system to optimize the transaction.

Isolation indicates the transaction level of the transactional method.
5 options provided:

Default: isolation level in transaction method is as same as that in the database
The rest 4 isolation levels are:

Isolation Level	Dirty Read	Nonrepeatable Read	Phantom Read
READ UNCOMMITTED	Permitted	Permitted	Permitted
READ COMMITTED	--	Permitted	Permitted
REPEATABLE READ	--	--	Permitted
SERIALIZABLE	--	--	--

3. Spring MVC

MVC (model view controller)

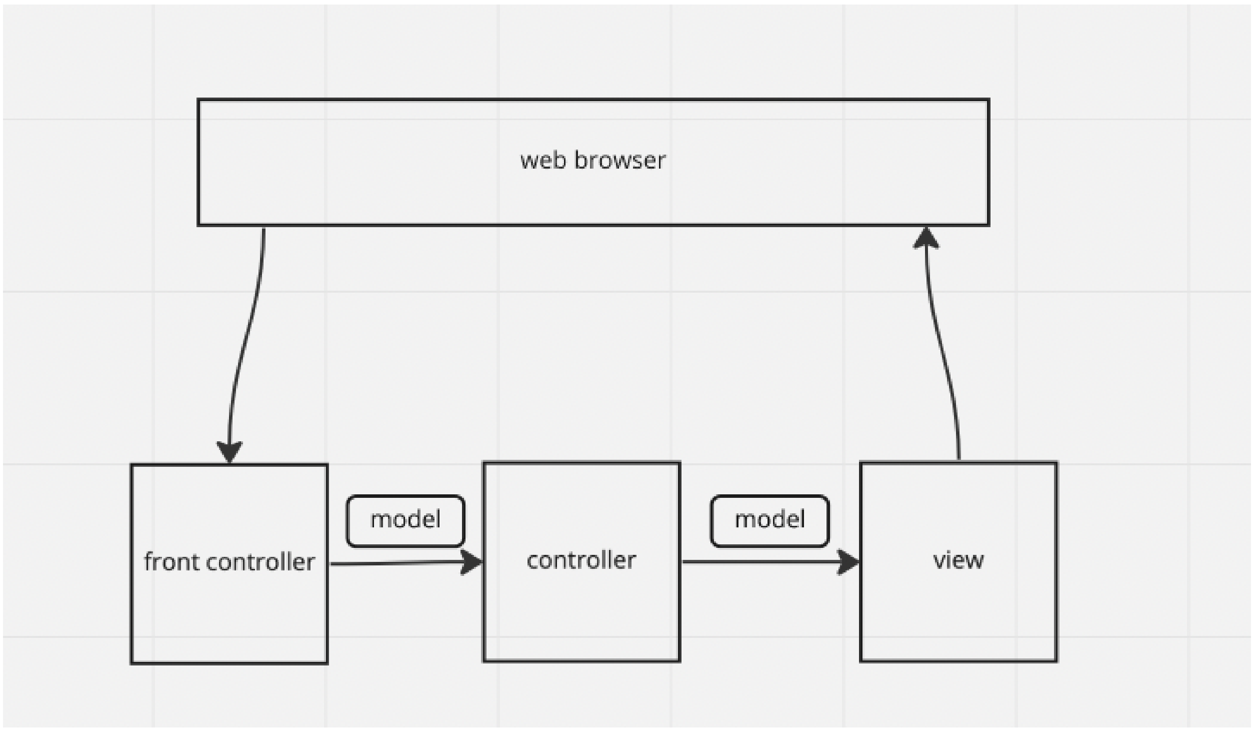
- Spring MVC is a framework that helps us with the separation of modules namely Model, View, Controller, and seamlessly handles the application integration.

- Model: contains the data of the application, the data can be a single object or a collection of objects

- Front controller: will intercept all the request, in spring mvc, the DispatchServerlet class works as the front controller

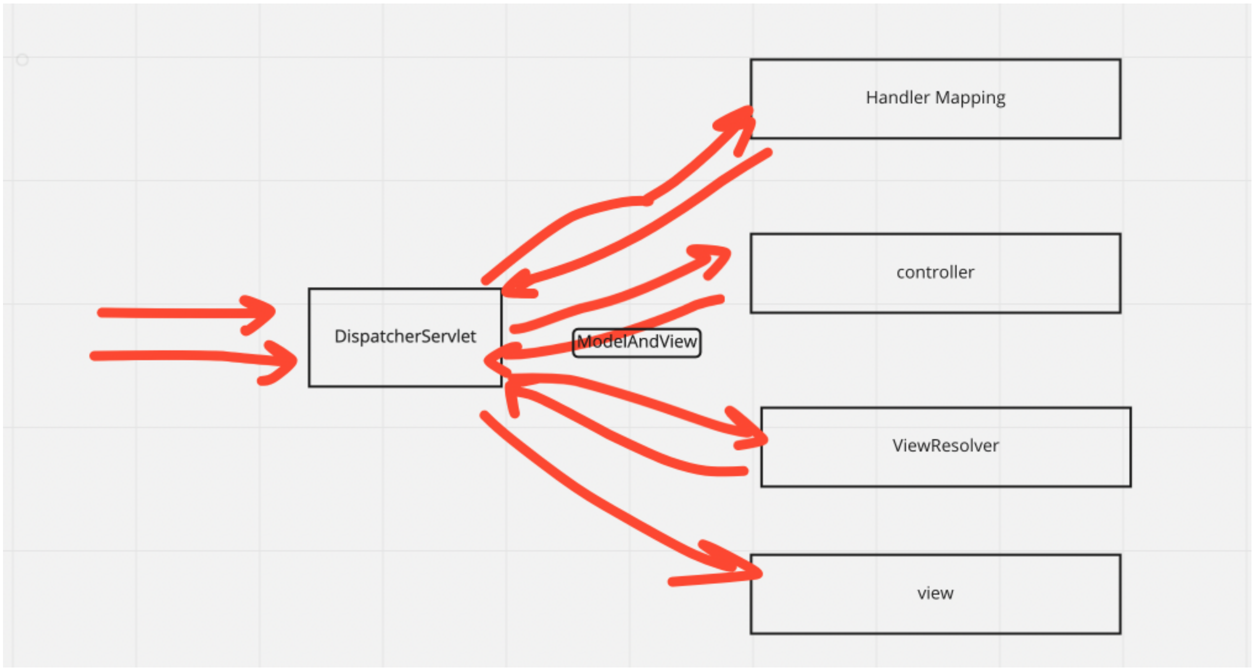
- Controller: business logic of an application

- View: display the information/data to user



Spring MVC workflow

1. All the request is intercepted by the DispatcherServlet that works as the front controller
2. The DispatcherServlet gets an entry of the handler mapping from the xml file and forwards the request to the controller
3. The controller returns an object of ModelAndView
4. The DispatcherServlet checks the entry of the view resolver in the xml file and invokes the specified view component



4. Spring Boot

Spring boot advantages

- Provides a flexible way to configure java beans, xml configuration, and database transaction
- Provides a powerful batch processing and manages rest endpoints
- Everything is auto configured, no manual configuration are need
- Offers annotation based spring application
- Ease dependency management
- Include embedded servlet container (Tomcat)

Spring boot Starter

- Spring boot provides Spring-boot-starters to help us handle the dependency management, which is a very difficult task when our project gets bigger

Auto Configuration

- Auto configuration automatically configures application based on the Jar dependencies you added in the project

Rest API Design

- Restful API are APIs that follow the REST principle, which is an architectural style based on http that is industry known, it provides guidelines for us to design requests that are sent easily and efficiently from client to server.

- Restful APIs are simple, and it's now an industry standard, it's scalable and stateless, it supports caching, therefore, the performance is good.

- HTTP methods & CRUD operations
  - create: post
  - read: get
  - update: post
  - delete: delete

- Idempotent:
  - Idempotent in here means that making multiple requests to the server is the same as making a single request. Some example will be get, put, delete
- Safe
  - A HTTP method is considered as safe if it doesn't modify the resource. An example would be get
- Cacheable
  - A HTTP method is considered as cacheable if its response can be cached, stored and used later. An example will be get

- HTTP status code:
  - 1xx   information
  - 2xx   success
  - 3xx   redirect
  - 4xx   client side error
  - 5xx   server side error

- Some common status code
  - 200   ok
  - 201   created
  - 202   accepted
  - 204   no content
  - 400   bad request

    - 401 unauthorized
    - 403 forbidden
    - 404 not found
    - 405 method not allowed
    - 500 internal server error

- HTTP url design
    - In order to handle crud (create, read, update, delete) actions, we use http methods

- If we want to create CRUD APIs for a list employees, we can do the following

get	/api/employees	retrieve a list of employees
get	/api/employees/10	retrieve a specific employee with id 10
put	/api/employees/10	update a specific employee with id 10
post	/api/employees	create a new employee
delete	/api/employees/10	delete a specific employee with id 10

- To access specified employee’s information, such as email, we can do the following

each employee has different email

get	/api/employees/10/emails	retrieve a list of emails for emp with id 10
get	/api/employees/10/emails/5	
post	/api/employees/10/emails	
put	/api/employees/10/emails/5	
delete	/api/employees/10/emails/5	

- We can also do some filtering, sorting or search

filter
get /api/employees?state=home

sort
get /api/employees?sort=salary,-created_at

search
get /api/employees?q=java

Spring Restful API

- In order to respond to HTTP requests, we also need to design APIs in the spring.
- The following is an example for the request to access user informations

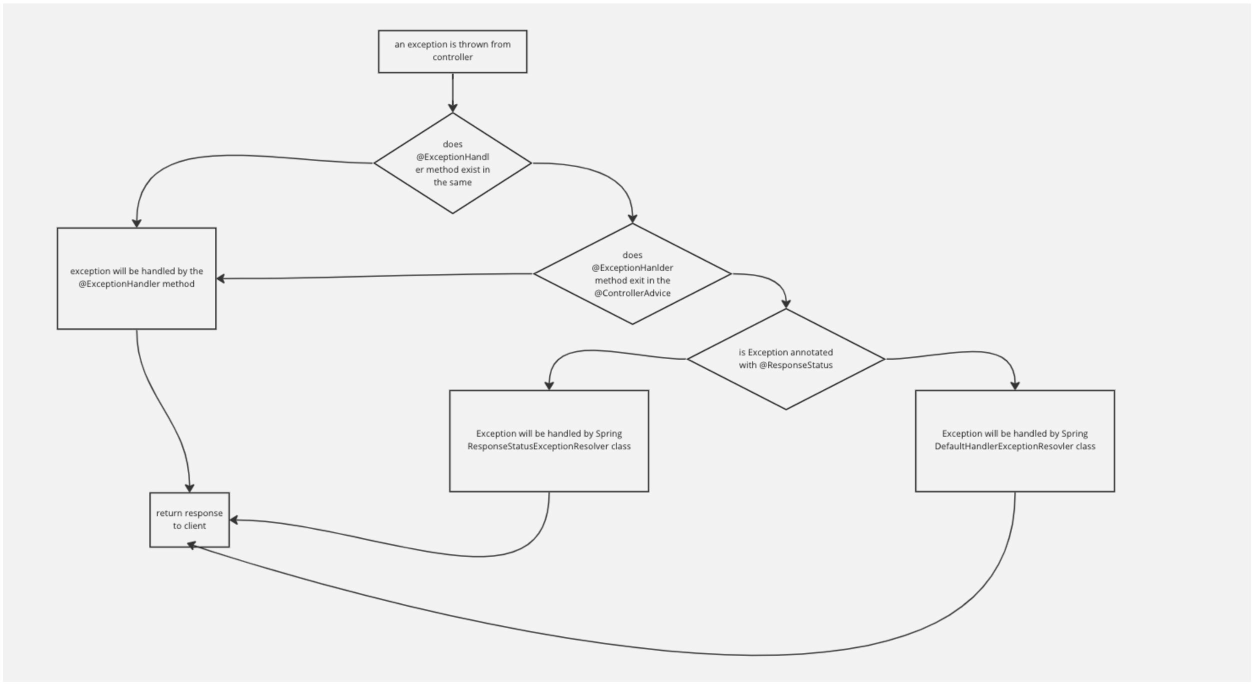
get   /api/user/{uid}
get   /api/user?pageNo=2&rows=10&orderBy=salary
post   /api/user
put   /api//user/{id}
delete /api//user/{id}

Annotations:

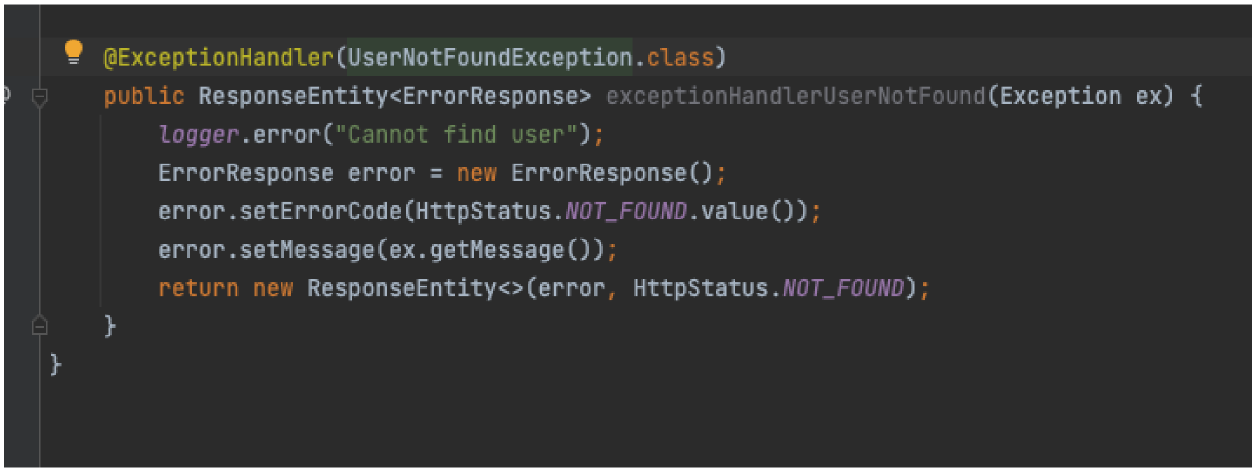
- RequestMapping, GetMapping, PutMapping, PostMapping, DeleteMapping
- RequestParam, PathVariable
- RequestBody (json -> java object), ResponseBody (java object -> json)
- Controller/RestController, Service, Repository
    - Controller + ResponseBody = RestController

5. Exception Handling Process

- During the runtime, exceptions and errors could happen, and Spring handle them by using exceptionHandler
- There are two types exceptionHandler
    - A local handler, by using the annotation @ExceptionHandler()
    - A global handler, by using the annotation @ControllerAdvice
    - Spring will first try to find a local handler, if no local handler is found, it will use the global handler

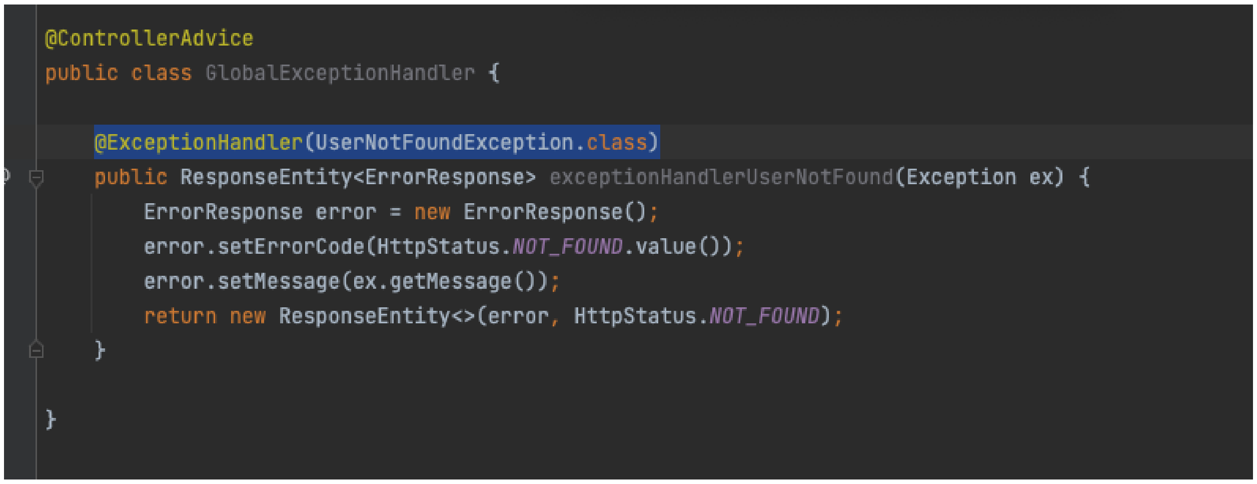

an exception is thrown from controller
does @ExceptionHandler method exist in the same
exception will be handled by the @ExceptionHandler method
does @ExceptionHanlder method exit in the @ControllerAdvice
is Exception annotated with @ResponseStatus
Exception will be handled by Spring ResponseStatusExceptionResolver class
Exception will be handled by Spring DefaultHandlerExceptionResovler class
return response to client

ExceptionHandler(local)



```
    @ExceptionHandler(UserNotFoundException.class)
    public ResponseEntity<ErrorResponse> exceptionHandlerUserNotFound(Exception ex) {
        logger.error("Cannot find user");
        ErrorResponse error = new ErrorResponse();
        error.setErrorCode(HttpStatus.NOT_FOUND.value());
        error.setMessage(ex.getMessage());
        return new ResponseEntity<>(error, HttpStatus.NOT_FOUND);
    }
}
```

ControllerAdvice (global)



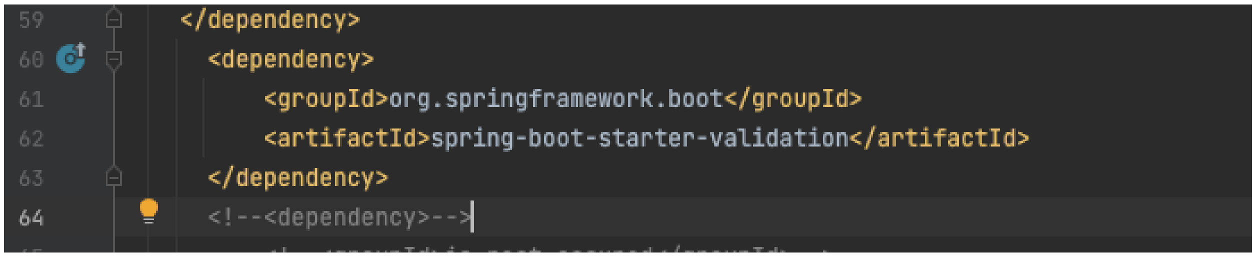
```
@ControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(UserNotFoundException.class)
    public ResponseEntity<ErrorResponse> exceptionHandlerUserNotFound(Exception ex) {
        ErrorResponse error = new ErrorResponse();
        error.setErrorCode(HttpStatus.NOT_FOUND.value());
        error.setMessage(ex.getMessage());
        return new ResponseEntity<>(error, HttpStatus.NOT_FOUND);
    }

}
```

@ResponseStatus(value = HttpStatus.NotFound)
public class NoSuchElementFoundException extends RuntimeException {

}

6. Validation

We can also do validation in spring boot, to do that, we need to add the following dependencies in our build configurations file.



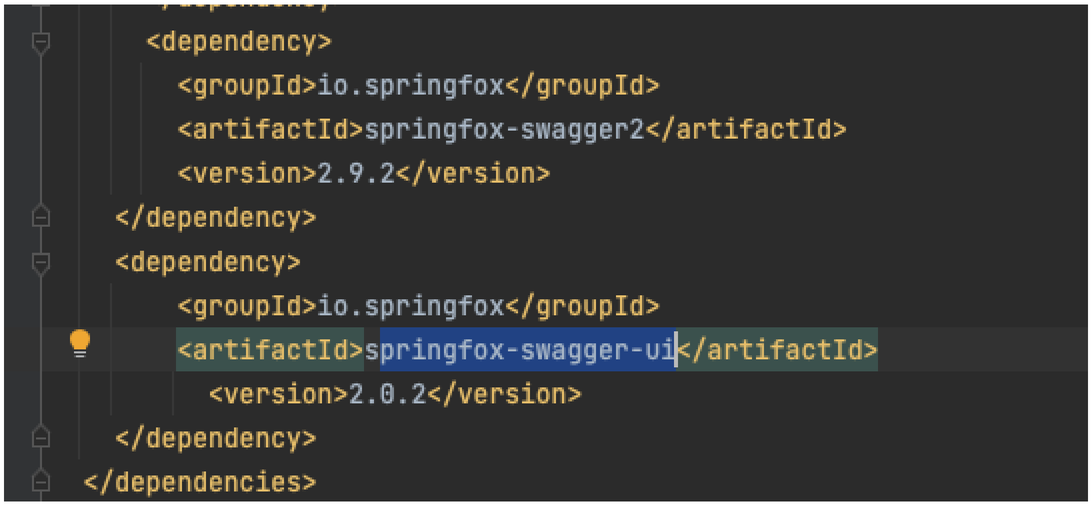
```
</dependency>
  <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-validation</artifactId>
  </dependency>
  <!--<dependency>-->
```

And in the field that needs to be validated, we can use annotations like @NotNull, @Max(), @Min(), Pattern(regex), Email, etc

7. Swagger

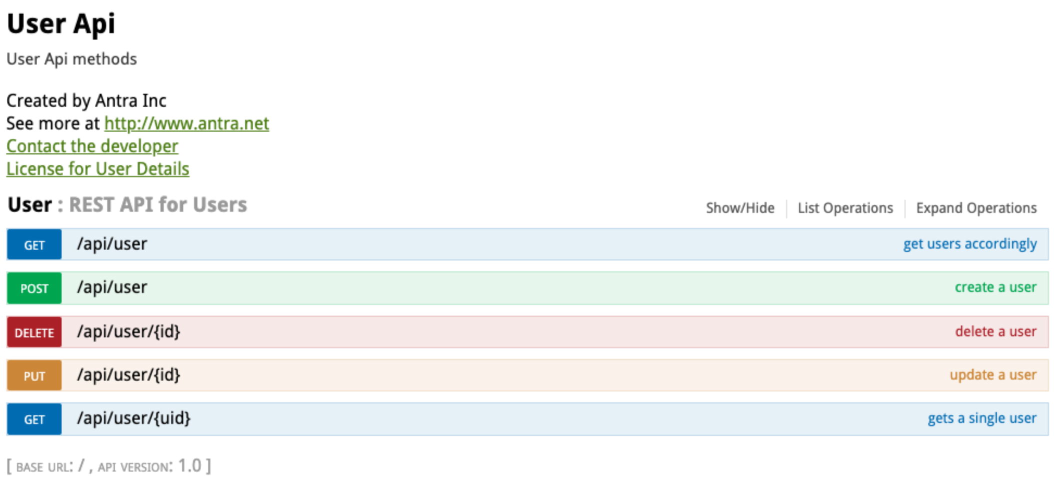
Swagger is a tool that provides a user interface to access our RESTful web services via the web browser.

To enable the Swagger in our application, we need to add the following dependencies in our build configurations file.



```
    <dependency>
      <groupId>io.springfox</groupId>
      <artifactId>springfox-swagger2</artifactId>
      <version>2.9.2</version>
  </dependency>
  <dependency>
      <groupId>io.springfox</groupId>
      <artifactId>springfox-swagger-ui</artifactId>
        <version>2.0.2</version>
  </dependency>
</dependencies>
```

And we'll get something like this



User Api

User Api methods

Created by Antra Inc
See more at http://www.antra.net
Contact the developer
License for User Details

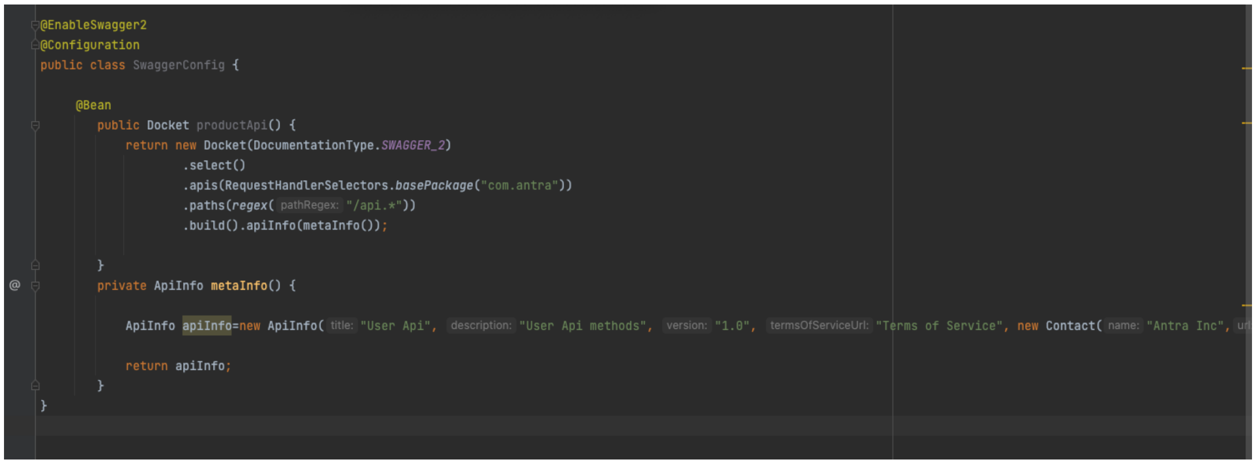
User : REST API for Users

Show/Hide | List Operations | Expand Operations

GET	/api/user	get users accordingly
POST	/api/user	create a user
DELETE	/api/user/{id}	delete a user
PUT	/api/user/{id}	update a user
GET	/api/user/{uid}	gets a single user

[BASE URL: / , API VERSION: 1.0]

In order to use swagger, we need to create a bean and configure it



```
@EnableSwagger2
@Configuration
public class SwaggerConfig {

    @Bean
    public Docket productApi() {
        return new Docket(DocumentationType.SWAGGER_2)
                .select()
                .apis(RequestHandlerSelectors.basePackage("com.antra"))
                .paths(regex( pathRegex: "/api.*"))
                .build().apiInfo(metaInfo());

    }
    private ApiInfo metaInfo() {

        ApiInfo apiInfo=new ApiInfo( title: "User Api",  description: "User Api methods",  version: "1.0",  termsOfServiceUrl: "Terms of Service", new Contact( name: "Antra Inc", ur

        return apiInfo;
    }
}
```