# 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
      - The most recommended way, it makes unit test easier
    - Setter based
      - Flexible, we can do partial injection
    - Field based
      - 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:

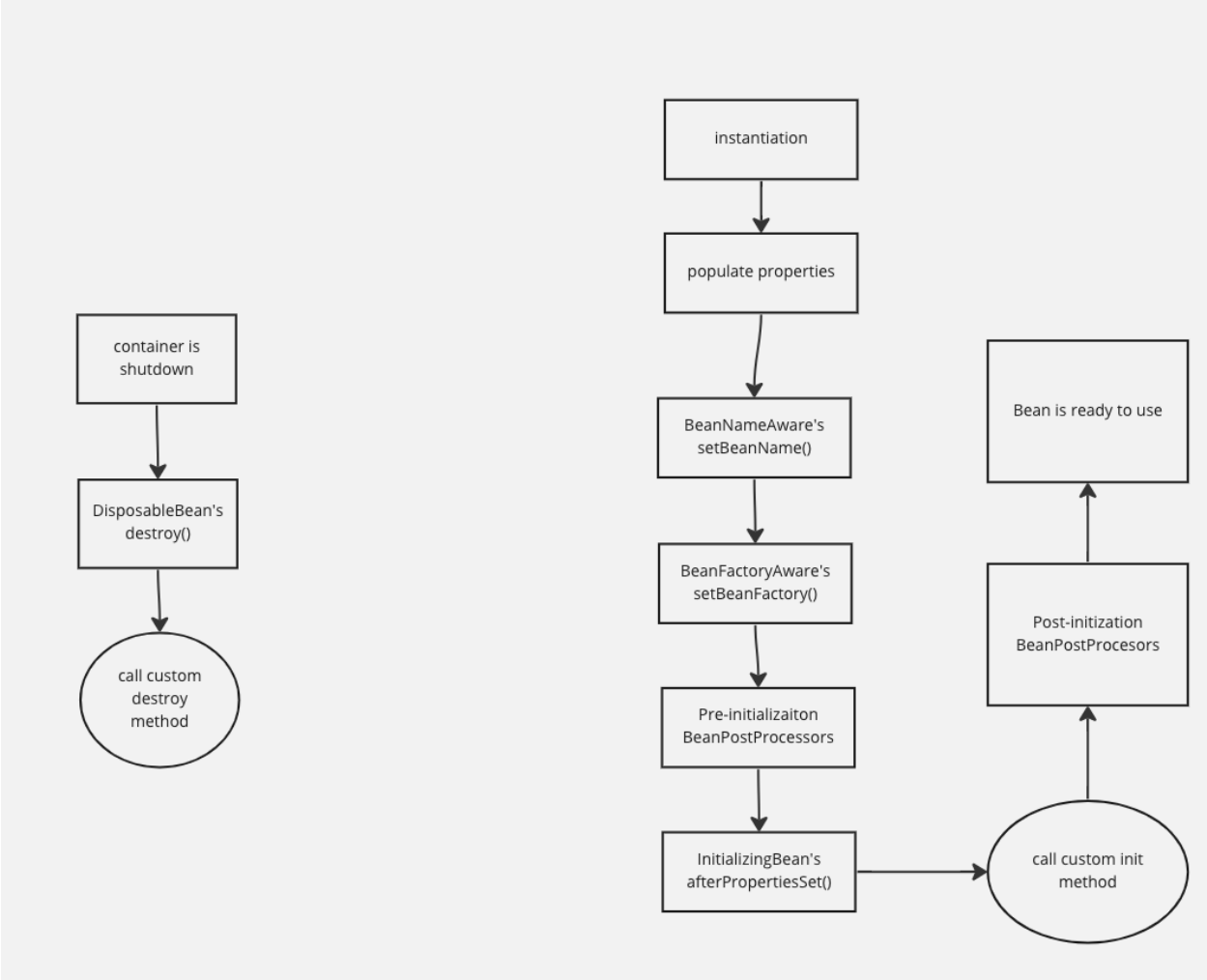
- decoupling the execution of task from its implementation
- making it easier to switch between different implementation

- 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




In order to put our logic into the process, we can use the following annotation

@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)

enables aspect oriented programming in spring application. 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

propagation/ rollback/ timeout

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 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.

isolation/readOnly

**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 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

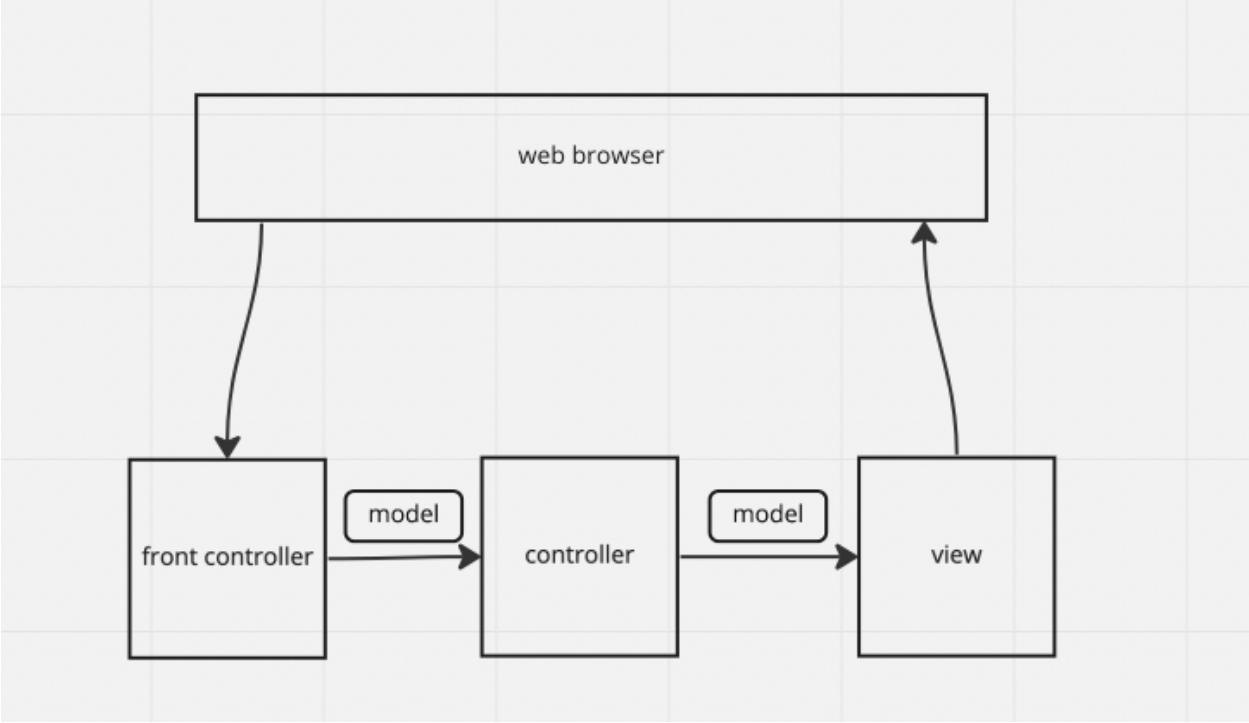
## MVC( model view controller)

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

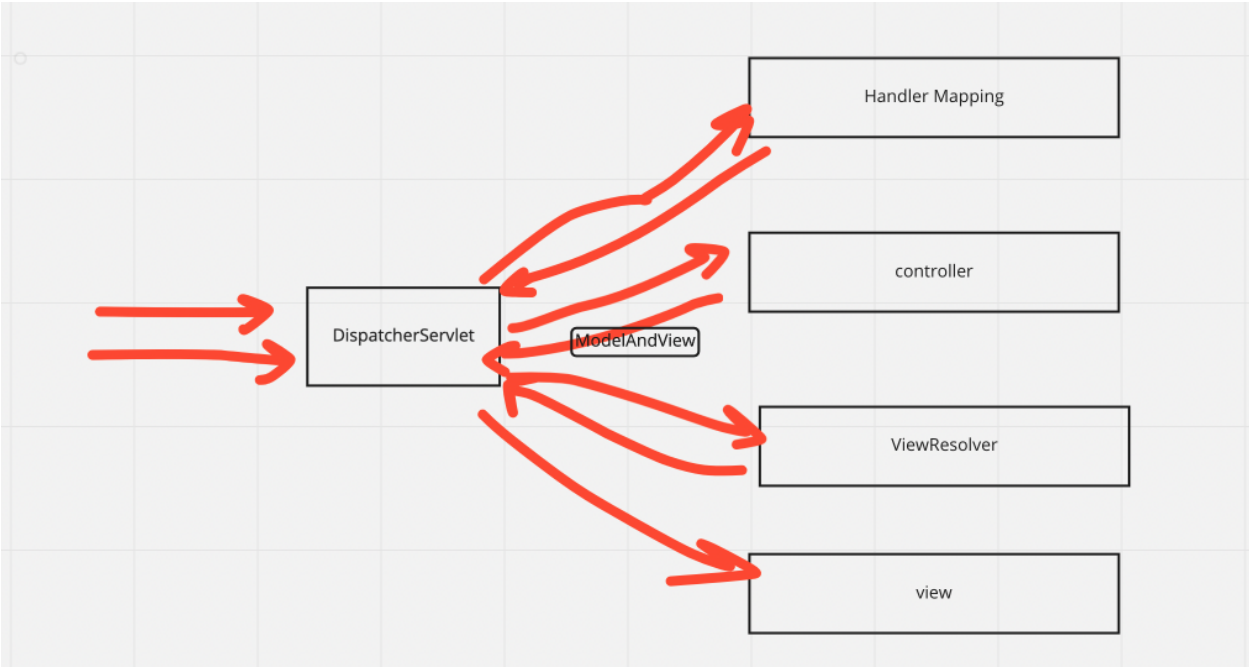
Front controller: will intercept all the request, in spring mvc, the DispatchServerlet class works as the front controller

Controller: business logic of an application

View: display the information/data to user




## Spring MVC workflow




All the request is intercepted by the DispatcherServlet that works as the front controller

The DispatcherServlet gets an entry of the handler mapping from the xml file and forwards the request to the controller
The controller returns an object of ModelAndView
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

- it provides a flexible way to configure java beans, xml configuration, and database transaction
- it provides a powerful batch processing and manages rest endpoints
- in spring boot, everything is auto configured, no manual configuration are need
- it offers annotation based spring application
- Ease dependency management
- Include embedded servlet container -> Tomcat

## Spring boot Starter

Handling dependency management is a difficult task for big projects. Spring boot introduces the Spring-boot-starter to help manage it

## Auto Configuration

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

## Rest API Design

http method: CRUD operations

- create: post
- read: get
- update: post
- delete: delete

idempotent: get, put, delete
safe: get
cacheable: get

cache control

http status code:

- 1xx information
- 2xx success
- 3xx redirect
- 4xx client side error
- 5xx server side error

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

handle crud (create, read, update, delete) actions using http method

| | | |
|---|---|---|
| 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 new employee |
| delete | /api/employees/10 | delete a specific employee with id 10 |

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

filter
get /api/employees?state=home

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

search
get /api/employees?q=java

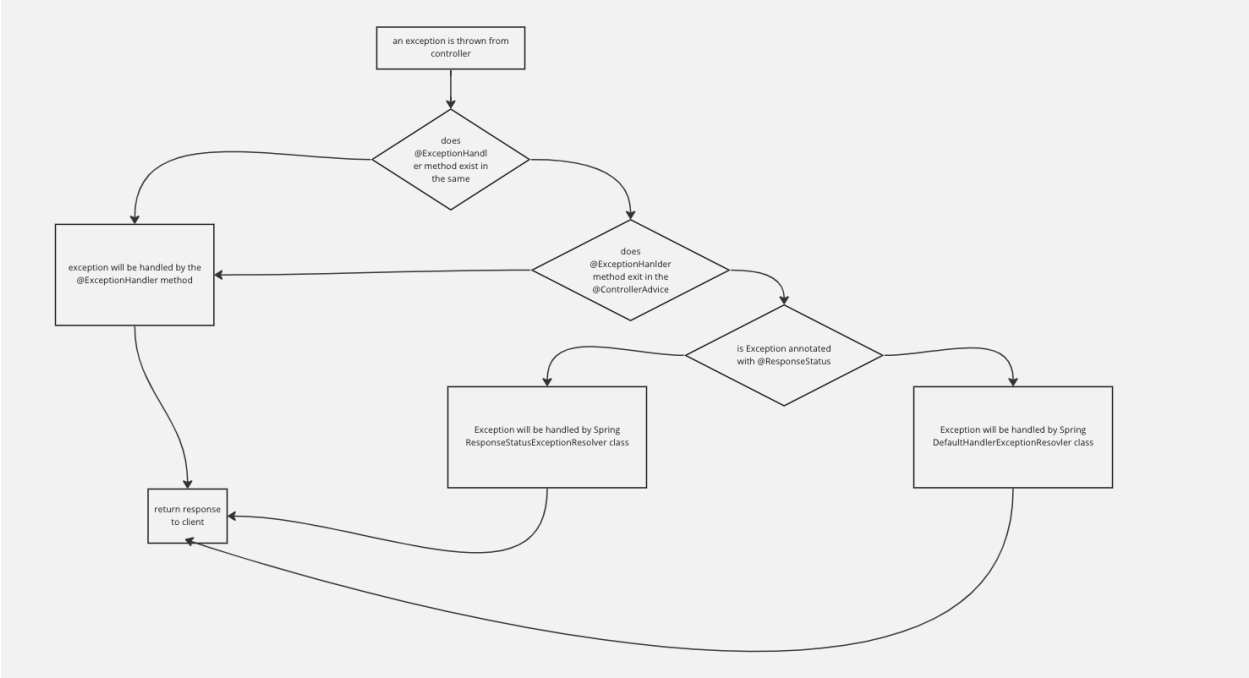
## Spring Restful API

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

annotation:

- 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




ExceptionHandler -> local

```java
@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

```java
@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.

```
59      </dependency>
60      <dependency>
61          <groupId>org.springframework.boot</groupId>
62          <artifactId>spring-boot-starter-validation</artifactId>
63      </dependency>
64      <!--<dependency>-->
```

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

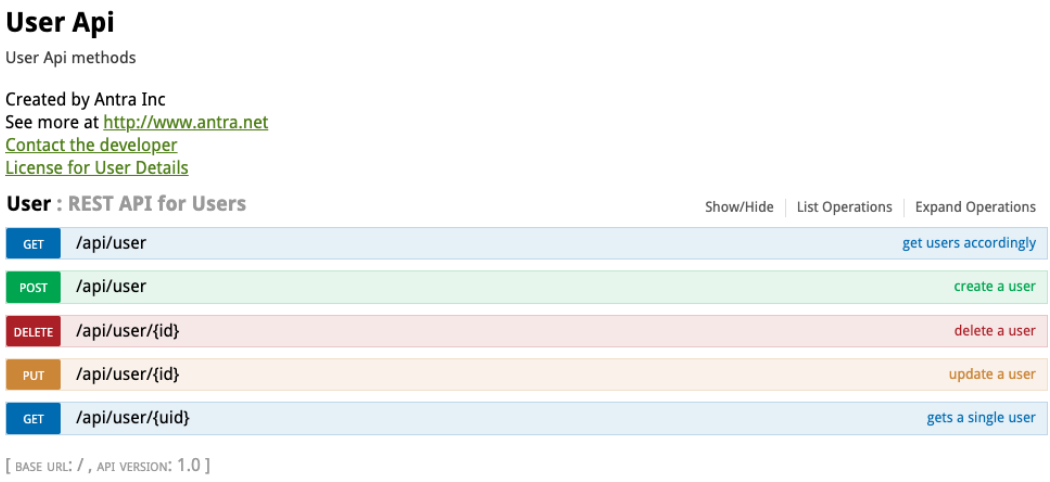
# 7. Swagger

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




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

```java
@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;
       }
}
```