**Domain class annotations**

@Entity denotes the class is an entity.

@Table denotes the database table to which this entity is mapping.

@Id denotes the primary key of the entity.

@GeneratedValue denotes the strategy of generating the primary key. The default strategy is the AUTO strategy.

@Column denotes the column mapping of entity attribute.

@JoinColum denotes the foreign key column.

@OnDelete denotes the cascade delete action. In this example, When the customer entity gets deleted, all its accounts will be removed at the same time.

**@Contoller** is a stereotype for the presentation layer to define controller.

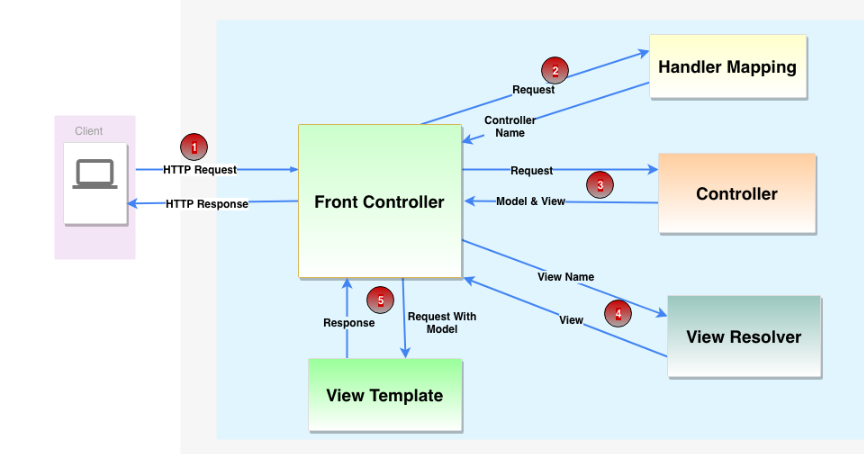
* Controller class is responsible to handle different kinds of client requests based on the request mapping.
* We can create controller class by using @Controller annotation. It is usually used with @RequestMapping annotation to define handle method for specific URI mapping.
* Controllers control the flow of the application execution.
* @Controller annotation is simply a specialization of the @Component class and allows implementation classes to be autodetected through the classpath scanning.
* In *Spring MVC architecture the*DispatcherServlet*works as*Front Controller. DispatcherServlet process the request and pass the request to the controller class annotated with *@Controller*.

**On a high level, Spring MVC workflow move around following points.**

1. Dispatcher Servlet intercepts all incoming requests.
2. Converts payload of the request to Spring MVC data structure.
3. Send a request to the specific controller for processing.
4. Invoke business rule/ processing from the controller.
5. Send a response to the view for rendering (using view resolver.)
6. For RESTful controllers, send the response directly to client skipping view resolver.

### *DispatcherServlet*

* DispatcherServlet works as front controller in the Spring MVC application and it loads the spring bean configuration file and initialize all the beans that are configured.
* If annotations are enabled, it also scan the package and configure any bean annotated with @Component, @Repository, @Service, @Controller annotations.
* DispatcherServlet that takes care of all incoming requests and delegate it to different controller handler methods.



**ContextLoaderListener**

* ContextLoaderListener is the listener to start up and shutdown spring root web application context.
* Its important functions are to tie up the life cycle of ApplicationContext to the life cycle of ServletContext and to automate creation of application context.
* We can use it to define shared beans that can be used across different spring contexts.

**Spring MVC is built on top of MVC architecture.**

DispatcherServlet works as front controller in Spring MVC application that takes care of all the incoming requests and delegate it to different controller handler method.

Model can be any Java Bean in spring framework. Spring provides automatic binding of form data to java beans. We can set model beans as attributes to be used in the view pages.

View resolvers are responsible for resolving for finding the correct view page. Once the view page is identified, control is given back to the DispatcherServlet controller.

Dispatcher controller is responsible for rendering the view and returning the final response to the client.

**Spring MVC Interceptor**

Interceptors are useful to intercept the client request and process it. There are three options to intercept client request using Spring interceptors.

* preHandle
* postHandle
* afterComplete (after the view is rendered to the client)

Interceptors are useful for cross-cutting concerns and help us avoid duplicate code (e.g. logging, profiling etc.).

We can create spring interceptor by implementing a HandlerInterceptor interface or by extending the abstract class HandlerInterceptorAdapter.

### @EnableWebMvc annotation in Spring MVC

The @EnableWebMvc annotation allows you to configure Spring application without using the XML based configuration/declaration. This annotation is equivalent to using the <mvc:annotation-driven /> in XML. Here is the list of some important features of the @EnableWebMvc annotation:

* Enable support for request processing.
* Support for validation using JSR-303 bean validation.
* Number and date formatting support.
* HttpMessageConverter support

**@Service** is a stereotype for the service layer. This class file are used to write business logic in a difference layer, separated from @RestController.

Annotated class which executes business logic, perform calculations and call external API in the service layer.

**@Repository** is a stereotype for the persistence layer. It is used in Data Access Layer.

**org.springframework.web.bind.annotation.RestController**

* **@RestController** is class level annotation that is used to handle the requests.
* @*RestController* is a specialized version of the controller. The controller is defined with @RestController annotation, therefore ResponseBody is not required.
* It is a combination of the @Controller and @ResponseBody annotation. It returns the domain object.
* @RestController annotation is used to define restful webservices.

***@RequestMapping***

* The *@RequestMapping* is used to map web request on to specific handler class and handler method application.
* We can apply the *@RequestMapping annotation* to class-level and/or method-level in a controller.
* When RequestMapping is used on class level it creates base URI for which the controller will be used.
* When this annotation is used on method level, it will give you the URI on which handler methods will be executed.
* Sometimes you may want to perform different operation based on the Http Method used, even though the request URI may remain same.
* **@**RequestMapping annotation is used on request handling methods inside controller class.
* We can define the request method to consume and produce objects.
* The default request method is GET.

@RequestMapping(value="/vehicles", method=RequestMethod.GET)

* It is used to define the request URI to access the REST endpoints.

**@RequestMapping annotation provides several options to customize its behavior.**

Consumes – **The consumable media types of the mapped request**, narrowing the primary mapping. (e.g. @RequestMapping(consumes = {"application/json", "application/xml"})).

method – The HTTP request methods to map (e.g. method = {RequestMethod.GET,RequestMethod.POST}).

header – The **headers of the mapped request**.

name – the **name of the mapping**.

value – **The primary mapping expressed by this annotation**

produces – **The producible media types of the mapped request**.

Here is an example for the @RequestMapping

@Controller

public class SpringMVCController {

@RequestMapping(value = { "/greetings", "/hello-world"},

method = {RequestMethod.GET,RequestMethod.POST},

consumes = {"application/json","application/xml"},

produces = { "application/json"},

headers = {"application/json"

})

public String hellpWorld() {

return "Hello";

}

}

**@GetMapping**

* @GetMapping is used for mapping Http GET requests onto specific handler method.
* @GetMapping is composed annotation that act as a shortcut for @RequestMapping(method=RequestMethod.GET)

**@PostMapping**

* @PostMapping is used for mapping Http POST requests onto specific handler method.
* @PostMapping is composed annotation that act as a shortcut for @RequestMapping(method=RequestMethod.POST)

**@PutMapping**

* @PutMapping is used for mapping Http PUT requests onto specific handler method.
* @PutMapping is composed annotation that act as a shortcut for @RequestMapping(method=RequestMethod.PUT)

**@PatchMapping**

* @PatchMapping is used for mapping Http PATCH requests onto specific handler method.
* @PatchMapping is composed annotation that act as a shortcut for @RequestMapping(method=RequestMethod.PATCH)

**@DeleteMapping**

* @DeleteMapping is used for mapping Http DELETE requests onto specific handler method.
* @DeleteMapping is composed annotation that act as a shortcut for @RequestMapping(method=RequestMethod.DELETE)

|  |  |
| --- | --- |
| org.springframework.web.bind.annotation.GetMapping | @GetMapping(value="/vehicles") |
| org.springframework.web.bind.annotation.PostMapping | @PostMapping(value="/vehicles") |
| org.springframework.web.bind.annotation.PutMapping | @PutMapping(value = "/customers/{customerId}") |
| org.springframework.web.bind.annotation.PatchMapping | @PatchMapping(value = "/customers/{customerId}") |
| org.springframework.web.bind.annotation.DeleteMapping | @DeleteMapping(value = "/customers/{customerId}") |

***@RequestBody* - org.springframework.web.bind.annotation.RequestBody**

* The @RequestBody annotation is used in the handler method parameter.
* @RequestBody annotation is used to binds the HttpRequest body to a domain object.
* The @RequestBody annotation indicates that a method parameter should be bound to the value of Http Request body.
* @RequestBody, all the data will be passed to the server to the body of the web request.
* HttpMessageConverter is responsible for converting from HttpRequest message to object.
* Spring automatically deserializes incoming HttpRequest to the Java Object using HttpMessageConverter.

@PostMapping("/product/save")

public String saveProduct(@RequestBody Product product){}

**@ResponseBody**

* *The @ResponseBody annotation is used to annotate request handler method*
* *The @ResponseBody annotation allows you to return the response directly in the HTTP response body.*
* *Spring converts the returned object into a response body by using HttpMessageConverter.*
* *The @ResponseBody annotation is useful while working on the REST services in Spring MVC where we want to write the response directly into the HTTP response.*
* *Spring treats the result of the method as the response itself.*
* *We can use this annotation when we do not want to send/place the response into the Model or view name is not needed.*

@GetMapping("/products/{id}")

public @ResponseBody Product saveProduct(@PathVariable("id") String id) {

//

}

**@ResponseStatus** - **org.springframework.web.bind.annotation.ResponseStatus**

* @ResponseStatus is used on methods and exception classes.
* @ResponseStatus marks a method or exception class with status code and a reason that must be returned.
* we can specify the desired Http status of the response in the handler method with this annotation.

@ResponseStatus(code=HttpStatus.CREATED)

* If we want to specify the response status of a controller method, we can mark that method with @ResponseStatus.

**@PathVariable**

* @PathVariable is used for mapping dynamic values from the URI to handle method arguments.
* We specify path variable as part of the @RequestMapping and bind a method argument with @PathVariable.
* The Path variable is defined as curly braces {} in request URI.
* It is used to get the parameter value passed along with the URL.

@RequestMapping("/products/{code}")

public String getProduct(@PathVariable(value = "code") String code,

@RequestParam(value = "category", required = false) String category) {

.......

}

**@RequestParam**

* @RequestParam is used to annotate request handler method arguments.
* The @RequestParam annotation binds the request parameter to method parameter in your controller.
* In simple words, we can use the @RequestParam annotation to get values from query parameters and from parameters.
* Request parameters passed by the browser/client as part of the HTTP request.
* It is used to read or accessing the http request parameter from the requested URL.

@GetMapping("/greeting")

public String sayHello(@RequestParam String name) {

return "Hello "+name+"!!!";

}

<http://localhost:8080/greeting?name=javadevjournal>

The **RequestParam annotation**can be **map multiple values in a list**.

@GetMapping("/products")

public String getProducts(@RequestParam List < String > id) {

return "Products: " + id;

}

Here is the output for our requests:

curl -i -H "Accept: application/json" -H "Content-Type: application/json" http://localhost:8080/products?id=12,13,14

HTTP/1.1 200

Content-Type: application/json;charset=UTF-8

Content-Length: 22

Date: Sun, 16 Dec 2018 02:01:46 GMT

Products: [12, 13, 14]

http://localhost:8080/products?id=12&id=13

Output: Products: [12, 13]

**@ModelAttribute**

* The *@ModelAttribute annotation* can be used at method and method parameter level.
* The @ModelAttribute annotation that binds a method parameter or method return value to a named model attribute, exposed to a web view.
* [Spring MVC](https://www.javadevjournal.com/spring-mvc/spring-mvc-tutorial/) will invoke all methods with @ModelAttribute annotation before any handler method executed by the framework.
* An @ModelAttribute on a method argument indicates the argument should be retrieved from the model. If not present in the model, the argument should be instantiated first and then added to the model.

**@ModelAttribute**

* The *@ModelAttribute annotation* can be used at method and method parameter level.
* The @ModelAttribute annotation that binds a method parameter or method return value to a named model attribute, exposed to a web view.
* [Spring MVC](https://www.javadevjournal.com/spring-mvc/spring-mvc-tutorial/) will invoke all methods with @ModelAttribute annotation before any handler method executed by the framework.
* An @ModelAttribute on a method argument indicates the argument should be retrieved from the model. If not present in the model, the argument should be instantiated first and then added to the model.

@PostMapping("/customer-registration")

public String processForm(@ModelAttribute("customer") Customer customer) {}

The annotation is used to define objects which should be part of a Model. So if you want to have a Customer object referenced in the Model you can use the following method:

@ModelAttribute("customer")

public Person getCustomer() {

//

}

we don’t have to specify the model key, Spring uses the method’s name by default

@ModelAttribute

public Person getCustomer() {

//

}

**@ExceptionHandler ( org.springframework.web.bind.annotation.ExceptionHandler)**

* @ExceptionHandler is used at method levels to handle exception at controller level.
* @ExceptionHandler is used to define the class exception it will catch.

@ExceptionHandler(CustomException1.class)

* @ExceptionHandler values can be set to an array of exception types.
* If an exception is thrown that matches one of the types in the list, then the method annotated with matching @ExceptionHandler will be invoked.
* **@ExceptionHandler is used to handle the specific exception thrown by request and sending the custom responses to the client.**
* the @ExceptionHandler annotated method is only active for that particular Controller, not globally for the entire application.

//...

@ExceptionHandler({ CustomException1.class, CustomException2.class })

public void handleException() {

//

}

**@ControllerAdvice ( org.springframework.web.bind.annotation.ControllerAdvice)**

* @ControllerAdvice annotation is used to handle the exception globally.
* It applies them across the application, not just to an individual controller.

Let’s discuss some important points for the @ControllerAdvice annotation:

1. The @ControllerAdvice annotation is a specialized @Component annotation. We have the flexibility to use this annotation for multiple controller classes (This works based on the Exception and not bind to the Controller).
2. Spring Boot automatically detects all classes annotated with @ControllerAdvice during start-up as part of the classpath scanning.

**@RestControllerAdvice**

* This annotation is applied in Java classes.
* @RestControllerAdvice is a convenience annotation which combines @ControllerAdvice and @ResponseBody.
* This annotation is used along with the @ExceptionHandler to handle exceptions that occurs within the controller.

**Exception handling in a RESTful API** with **a meaningful error message** and **the status code** is a desired and must have feature. A good error message helps API client to take corrective actions.

**ResponseStatusException** (**Spring 5 and Above**)

Spring 5 introduced the ResponseStatusException class. We **can create an instance** of it **providing an HttpStatus and optionally a reason and a cause:**

@GetMapping(value = "/{id}")

public Foo findById(@PathVariable("id") Long id, HttpServletResponse response) {

try {

Foo resourceById = RestPreconditions.checkFound(service.findOne(id));

eventPublisher.publishEvent(new SingleResourceRetrievedEvent(this, response));

return resourceById;

}

catch (MyResourceNotFoundException exc) {

throw **new ResponseStatusException**(

HttpStatus.NOT\_FOUND, "Foo Not Found", exc);

}

}

**ResponseEntity - org.springframework.http.ResponseEntity**

ResponseEntity is used in some of the RESTful controller methods to represent the whole response includes status code, headers and response body.

Unlike @ResponseBody annotation which only populates the response body in Http Response, it gives us more freedom to manipulate the whole Http Response.

ResponseEntity represents the whole Http Response, status code, header and body.

We can use ResponseEntity to fully configure the http response.

- return new ResponseEntity<>("helloworld",HttpStatus.OK);

- return ResponseEntity.OK("helloworld");

- return new ResponseEntity<>("helloworld",headers,HttpStatus.OK);

**@PutMapping(value = "/customers/{customerId}")**

public ResponseEntity<Customer> updateCustomer(@PathVariable Integer customerId,@RequestBody Customer newCustomer){

return customerRepository.findById(customerId).map(customer -> {

customer.setCustomerName(newCustomer.getCustomerName());

customer.setDateofBirth(newCustomer.getDateofBirth());

customer.setPhoneNumber(newCustomer.getPhoneNumber());

customerRepository.save(customer);

return ResponseEntity.ok(customer);

}).orElseThrow(() -> new ResourceNotFoundException("Customer [customerId="+customerId+"] can't be found"));

}

**RestTemplate(org.springframework.web.client.RestTemplate)**

- RestTemplate is used to create application that consumes RESTful web services.

- you can use exchange() method to consume web services for all Http methods.

- use HttpHeaders to set the Request Headers.

- use HttpEntity to wrap the request object.

- provide the url, httpmethod and return type for exchange() method.

**Spring Boot** makes it easier to create production ready application in no time.

Spring Boot provides Rapid Application development feature to create spring application quickly.

Spring Boot provides the default configuration which helps to avoid the lot of boiler code which is painful to maintain.

Spring Boot provides a lot of starter projects to help you create a different type of spring application.

Spring Boot is basically an extension of spring framework which eliminated the boiler plate configurations required for setting up a spring application.

**Advantages**

1. It comes with embedded Server Tomcat or Jetty, so you don’t deploy war.
2. You can easily create the spring projects using spring Boot initializer.
3. It increases the productivity as you can create spring application very quickly.
4. It reduces development code by avoiding a lot of boiler code.
5. It provides the default configuration which helps to create spring application faster.
6. It provides the lot of starter projects for easy maven configuration. You don’t have to worry about version mismatch

**Disadvantages**

1. If you want to convert the old spring application to Spring Boot application, It may not be straightforward and can be time-consuming.

**Features**

1. Starter dependencies to simplify build and application configuration.
2. Embedded server to avoid complexity in application deployment.
3. Metrics, Health check and externalized configuration.
4. By default, Spring Boot uses an embedded container to run the application.

Spring Boot uses the public static void main entry point to launch an embedded web server.

Also, it takes care of the binding of the servlet, filter, and servlet context initializer beans from application context to the embedded servlet container.

1. Spring Boot is automatically scan all the classes in the same package or sub packages of main class for components.
2. Spring boot provides the option to deploying it as web archive in an external container as well. In this case, we have to extend the SpringBootServletInitializer.

**Auto configuration features**

* Auto configuration will attempt to automatically try to setup our application with default behaviour based on the jars in the classpath.
* **Spring boot automatically configures** 
  + A DispatcherServlet and ContextLoadListener
  + Spring MVC using @EnableWebMvc
* For example, if you add the spring-boot-starter-web starter in our application in POM.xml, Spring Boot auto configuration understands that we are building a MVC application and it added all required dependencies in our classpath for a Spring MVC application.
* Spring boot will add similar configuration based on spring boot starters added in the pom.xml

**@SpringBootApplication**

- The Entry point of spring boot application is class contains @SpringBootApplication annotation.

- This class should have main method to run the spring boot application.

- **@SpringBootApplication annotation is used to enable those three features**.

- **@EnableAutoConfiguration** - It enables the spring boot auto configuration. Spring boot automatically configures your application based on the dependencies you have added to the project by using @EnableAutoConfiguration.

- **@ComponentScan** - It enables component scan in the package where the component is located. Spring boot application scans all the beans and package declarations when the application initializes.

- **@Configuration** - It allows to register extra beans in context or import additional configuration.

- The class that is annotated with @SpringBootApplication must be kept in the base package. It will scan only its sub packages.

**spring-boot-starter-parent**

spring-boot-starter-parent dependency which is used internally by all spring boot dependencies.

We should specify the version number for spring boot starter parent dependency. then other starter dependencies, we do not need to specify the version of dependency.

<parent>

<group-id>org.springframework.boot</group-id>

<artifact-id>spring-boot-starter-parent</artifact-id>

<version>2.1.4.RELEASE</version>

</parent>

Spring Boot team provides a list of dependencies to support the spring boot version for its every release.

you do not need to provide a version for dependencies in the build configuration file.

Spring Boot automatically configures the dependency version based on the release.

* Provides UTF-8 encoding
* Repackage jar and makes it executable
* When you run mvn clean install package in spring boot, you will get 2 jar.

e.g. springExample-0.0.1.jar springExample-0.0.1.original.jar

**spring boot starters**

Starters are templates provided by Spring Boot, that provide dependencies to our application.

They are generally bundle of many dependencies.

- Spring boot provides a number of starter dependencies for different spring modules.

- Handling dependency management is difficult task for big projects. Spring Boot resolves this problem by providing a set of dependency for developer convenience.

- For example, if you want to use Spring and JPA for database access, it is sufficient if you include spring-boot-starter-data-jpa dependency in your project.

<dependency>

<group-id>org.springframework.boot</group-id>

<artifact-id>spring-boot-starter-data-jpa</artifact-id>

</dependency>

* For example, Let’s say you want to work with Spring MVC application, you can simply include spring-boot-starter-web as dependency in POM.xml
* Starter contains a considerable amount of dependencies that you have to get task up and running rapidly

**application.properties file**

Spring boot provides a lot of properties can be overridden by application.properties.

For example, if you want to specify the prefix and suffix in spring MVC application.

Spring.mvc.view.prefix=/WEB-INF/jsp

Spring.mvc.view.suffix=.jsp

You can run the application in the custom port number using Server.port

Server.port = 9090

We have to keep application.properties file under resources folder of the project.

server.port=8080

spring.application.name=demoService

debug=true

logging.level.root=WARN

**Spring Initializer**

- One of the ways to bootstrapping spring boot application by using Spring Initializer.

- Spring Initializer can produce spring boot project structure for you. It doesn’t create the application code.

However, it will give you an essential project structure and either maven or Gradle build specification to fabricate your code with.

Spring Initializer can be utilized a few different ways , including :

* Online interface
* Via Spring Tool Suite
* Using Spring Boot CLI

**Spring Boot servlet Initializer**

- The traditional way of deployment is making the spring boot application class extends the SpringBootServletInitializer class.

- Spring Boot servlet Initializer class file allows you to configure the application when it is launched by using servlet container.

@SpringBootApplication

public class DemoApplication extends SpringBootServletInitializer {

protected SpringApplicationBuilder configure(SpringApplicationBuilder application){

return application.sources(DemoApplication.class);

}

public void static main(String []args){

SpringApplication.run(DemoApplication.class,args);

}

}

**spring-boot-starter-actuator** dependency is used to monitor and manage your application.

**spring-boot-starter-security** dependency is used for Spring Security.

**spring-boot-starter-web** dependency is used to write a Rest end point.

**spring-boot-starter-test** dependency is used for writing test cases.

**spring-boot-dev-tools**

Dev tools are introduced to increase the productivity of the developer. You don’t need to redeploy your application every time you makes the changes

Application that uses spring-boot-dev-tools will automatically restart whenever files on the classpath changes. This can be useful feature when working in an IDE as it give fast feedback loop for code changes.

spring-boot-dev-tools are **used for auto reload of application with spring boot**.

**Its scope is for runtime**.

<dependency>

<group-id>org.springframework.boot</group-id>

<artifact-id>spring-boot-dev-tools</artifact-id>

<scope>runtime</scope>

</dependency>

**Spring Boot actuator**

* Spring Boot actuator is one of the important features of spring boot. It is used to access the current state of running application in production environment.

There are various metrics which you can use to check the current state of the application.

* Spring Boot actuator provides restful webservices end point which you can simply use and check various metrics.

For example, /metrics: this is restful end point will show you metrics such as free memory, processors, uptime, and many more properties.

* Spring Boot Actuator will help you to monitor your application in production environment.

Restful end points are sensitive, it means that it will have restricted access and will be shown only to restricted users. You can change this property overriding it in application.properties.

**Implement spring security in Spring Boot Application**

* Implementation of spring security in spring boot application requires very little configuration.
* You need to add Spring-Boot-starter-Security starter in Pom.xml.
* You need to create spring config class which will extend WebSecurityConfigurerAdapter and override required method to achieve security in Spring Boot Application.

**Configure the data-source using spring Boot application**

Use either spring-boot-starter-jdbc or spring-boot-starter-jpa and include a jdbc driver on the classpath.

# Database

spring.datasource.driver = com.mysql.cj.jdbc.Driver

spring.datasource.url = jdbc:mysql://localhost:3306/sample

spring.datasource.username = root

spring.datasource.password = password

**Configuring logging in Spring Boot Application**

* Spring boot comes with support of java util logging, log4j2, logback and it will be preconfigured as application output.
* Hence, you can simply specify logging.level in application.properties
* You can control the logging level in application.properties

Logging.level.org.framework=DEBUG

Logging.level.com.example=INFO

* By adding logback.xml file to application (src/main/resources), we can override the default logging configuration providing by spring boot.

**YAML (Yet Another Mark-up Language)**

* It is an alternative .properties file.
* It allows the hierarchical configuration.
* Java parser for YAML is snake parser.
* Provided by spring boot starters.

HttpStatus.NOT\_FOUND

HttpStatus.OK

HttpStatus.CREATED

**Cross Origin Resource Sharing (CORS)**

Cross Origin Resource sharing is security concept that allows restricting the resources implemented in web browsers.

It prevents the JavaScript code producing or consuming the request against different origin.

For Example, your web application is running on port 8080 and by using JavaScript, you are consuming RESTful web services from 9002 port. Under such situations, you will face the cross-origin resource sharing security issue on your web browser.

**Enable CORS in controller method**

* @CrossOrigin annotation is used both at class and method level to enable cross origin requests.
* CrossOrigionResourceSharing enables cross-domain communication. To enable this communication, you just need to add the @CrossOrigin annotation.
* By default, the @CrossOrigin annotation allows all origin, all headers, the Http methods specified in the request mapping annotation and max age of 30 min. you can customize the behaviour by specifying the corresponding attribute values.
* CrossOrigin can be used in controller and handler method level.
* This annotation allows the cross-domain communication for the annotated handler methods.
* This @CrossOrigin annotation enables cross-origin resource sharing only for this specific method. Let’s take an example where we want to allow only [http://localhost:9002](http://localhost:9000/) to send cross-origin requests.
* This annotation supports specific REST API and not for the entire application.

@RequestMapping(value="/products")

@CrossOrigin(origins="http://localhost:9002")

public ResponseEntity<Object> getProducts(){}

**import** javax.persistence.Column;

**import** javax.persistence.Entity;

**import** javax.persistence.GeneratedValue;

**import** javax.persistence.Id;

**import** javax.persistence.Table;

**import** lombok.Data;

**import** lombok.NoArgsConstructor;

@Entity

@Data

@NoArgsConstructor

@Table(name="userinfo")

**public** **class** User {

@Id

@GeneratedValue

**private** **long** id;

@Column(name="username")

**private** String name;

**private** String city;

}

**import** org.springframework.data.jpa.repository.JpaRepository;

**import** com.example.demo.model.User;

**public** **interface** UserRepository **extends** JpaRepository<User,Long> {

}

**import** java.util.List;

**import** com.example.demo.model.User;

**public** **interface** UserService {

**public** String createUser(User user);

**public** List<User> getUserAll();

**public** User findById(**long** userId);

**public** User updateById(**long** userId, User user);

**public** User updateByRequestBodyId(User user);

**public** User updatePartially(**long** userId, User user);

**public** **void** deleteById(**long** userId);

**public** String deleteAll();

}

**package** com.example.demo.service;

**import** java.util.List;

**import** java.util.Optional;

**import** org.springframework.beans.factory.annotation.Autowired;

**import** org.springframework.stereotype.Service;

**import** org.springframework.transaction.annotation.Transactional;

**import** com.example.demo.config.aop.advice.TrackExecutionTime;

**import** com.example.demo.model.User;

**import** com.example.demo.repository.UserRepository;

@Service

@Transactional

**public** **class** UserServiceImpl **implements** UserService {

@Autowired

UserRepository repository;

@Override

**public** String createUser(User user) {

repository.save(user);

**return** "User - " + user.getName() + "added successfuly ";

}

@Override

@TrackExecutionTime

**public** List<User> getUserAll() {

List<User> userList = repository.findAll();

**return** userList;

}

@Override

@TrackExecutionTime

**public** User findById(**long** userId) {

User user = **null**;

// User user = repository.getOne(userId);

Optional<User> Ouser = repository.findById(userId);

user = Ouser.get();

**return** user;

}

@Override

**public** User updateById(**long** userId, User user) {

user.setId(userId);

User updateUser = repository.save(user);

**return** updateUser;

}

@Override

**public** User updateByRequestBodyId(User user) {

User updateUser = repository.save(user);

**return** updateUser;

}

@Override

**public** User updatePartially(**long** userId, User user) {

Optional<User> Ouser = repository.findById(userId);

User upuser = Ouser.get();

**if** (user.getName() != **null**)

upuser.setName(user.getName());

**if** (user.getCity() != **null**)

upuser.setCity(user.getCity());

User updateUser = repository.save(upuser);

**return** updateUser;

}

@Override

**public** **void** deleteById(**long** userId) {

repository.deleteById(userId);

}

@Override

**public** String deleteAll() {

repository.deleteAll();

**return** "All User records deleted successfuly";

}

}

**package** com.example.demo.controller;

**import** java.util.List;

**import** org.springframework.beans.factory.annotation.Autowired;

**import** org.springframework.http.HttpStatus;

**import** org.springframework.http.ResponseEntity;

**import** org.springframework.web.bind.annotation.DeleteMapping;

**import** org.springframework.web.bind.annotation.GetMapping;

**import** org.springframework.web.bind.annotation.PatchMapping;

**import** org.springframework.web.bind.annotation.PathVariable;

**import** org.springframework.web.bind.annotation.PostMapping;

**import** org.springframework.web.bind.annotation.PutMapping;

**import** org.springframework.web.bind.annotation.RequestBody;

**import** org.springframework.web.bind.annotation.RequestMapping;

**import** org.springframework.web.bind.annotation.RestController;

**import** com.example.demo.model.User;

**import** com.example.demo.service.UserService;

@RestController

@RequestMapping("/jpa/")

**public** **class** UserController {

@Autowired

UserService service;

@PostMapping("/users")

**public** ResponseEntity<String> createUser(@RequestBody User user) {

String strOutput = service.createUser(user);

**return** **new** ResponseEntity<String>(strOutput, HttpStatus.***CREATED***);

}

@GetMapping("/users")

**public** List<User> getUserAll() {

**return** service.getUserAll();

}

@GetMapping("/users/{userId}")

**public** ResponseEntity<User> getUserById(@PathVariable **long** userId) {

User user = service.findById(userId);

**if** (user == **null**) {

**return** **new** ResponseEntity<User>(HttpStatus.***NOT\_FOUND***);

}

**return** **new** ResponseEntity<User>(user, HttpStatus.***OK***);

// It return the status, but it will not print user details.

// return new ResponseEntity<User>(HttpStatus.OK);

}

@PutMapping("/users/{userId}")

**public** ResponseEntity<User> updateById(@PathVariable **long** userId, @RequestBody User user) {

User userTemp = service.findById(userId);

**if** (userTemp == **null**) {

**return** **new** ResponseEntity<User>(HttpStatus.***NOT\_FOUND***);

}

User updateUser = service.updateById(userId, user);

**return** **new** ResponseEntity<User>(updateUser, HttpStatus.***OK***);

}

@PutMapping("/users/")

**public** ResponseEntity<User> updateByRequestBody(@RequestBody User user) {

User userTemp = service.findById(user.getId());

**if** (userTemp == **null**) {

**return** **new** ResponseEntity<User>(HttpStatus.***NOT\_FOUND***);

}

User updateUser = service.updateByRequestBodyId(user);

**return** **new** ResponseEntity<User>(updateUser, HttpStatus.***OK***);

}

@PatchMapping("/users/{userId}")

**public** ResponseEntity<User> updatePartially(@PathVariable **long** userId, @RequestBody User user) {

User userTemp = service.findById(user.getId());

**if** (userTemp == **null**) {

**return** **new** ResponseEntity<User>(HttpStatus.***NOT\_FOUND***);

}

User updateUser = service.updatePartially(userId, user);

**return** **new** ResponseEntity<User>(updateUser, HttpStatus.***OK***);

}

@DeleteMapping("/users/{userId}")

**public** **void** deleteById(@PathVariable **long** userId) {

service.deleteById(userId);

}

@DeleteMapping("/users")

**public** ResponseEntity<String> deleteAllUsers() {

String strmessage = service.deleteAll();

**return** **new** ResponseEntity<String>(strmessage, HttpStatus.***OK***);

}

}

**import** org.springframework.boot.SpringApplication;

**import** org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

**public** **class** SpringBootJpaHibernateApplication {

**public** **static** **void** main(String[] args) {

SpringApplication.*run*(SpringBootJpaHibernateApplication.**class**, args);

}

}

**import** org.springframework.beans.factory.annotation.Autowired;

**import** org.springframework.context.annotation.Bean;

**import** org.springframework.context.annotation.Configuration;

**import** org.springframework.security.config.annotation.authentication.builders.AuthenticationManagerBuilder;

**import** org.springframework.security.config.annotation.method.configuration.EnableGlobalMethodSecurity;

**import** org.springframework.security.config.annotation.web.builders.HttpSecurity;

**import** org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;

**import** org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;

**import** org.springframework.security.core.userdetails.UserDetailsService;

**import** org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;

@Configuration

@EnableWebSecurity

@EnableGlobalMethodSecurity(prePostEnabled=**true**)

**public** **class** SecurityConfig **extends** WebSecurityConfigurerAdapter {

@Autowired

UserDetailsService userDetailsService;

@Override

**protected** **void** configure(AuthenticationManagerBuilder auth) **throws** Exception {

auth.userDetailsService(userDetailsService).passwordEncoder(encodePWD());

auth.inMemoryAuthentication().withUser("basicin").password("password").roles("ADMIN");

}

@Override

**protected** **void** configure(HttpSecurity http) **throws** Exception {

http.csrf().disable();

http.authorizeRequests().antMatchers("/rest/\*\*").authenticated().anyRequest()

.permitAll().and()

.authorizeRequests().antMatchers("/secure/\*\*").authenticated().anyRequest()

.hasAnyRole("ADMIN").and()

.formLogin().permitAll();

}

@Bean

**public** BCryptPasswordEncoder encodePWD() {

**return** **new** BCryptPasswordEncoder();

}

}

**package** com.example.demo.service;

**import** java.util.Collection;

**import** java.util.stream.Collectors;

**import** org.springframework.security.core.GrantedAuthority;

**import** org.springframework.security.core.authority.SimpleGrantedAuthority;

**import** org.springframework.security.core.userdetails.UserDetails;

**import** com.example.demo.model.User;

**import** lombok.Getter;

**import** lombok.Setter;

@Getter

@Setter

**public** **class** CustomUserDetails **implements** UserDetails {

**private** **static** **final** **long** ***serialVersionUID*** = 1256711395932122675L;

**private** User user;

@Override

**public** Collection<? **extends** GrantedAuthority> getAuthorities() {

**return** user.getRoles().stream().map(role->**new** SimpleGrantedAuthority("ROLE\_"+role)).collect(Collectors.*toList*());

}

@Override

**public** String getPassword() {

**return** user.getPassword();

}

@Override

**public** String getUsername() {

**return** user.getUsername();

}

@Override

**public** **boolean** isAccountNonExpired() {

**return** **true**;

}

@Override

**public** **boolean** isAccountNonLocked() {

**return** **true**;

}

@Override

**public** **boolean** isCredentialsNonExpired() {

**return** **true**;

}

@Override

**public** **boolean** isEnabled() {

**return** **true**;

}

}

**import** org.springframework.beans.factory.annotation.Autowired;

**import** org.springframework.security.core.userdetails.UserDetails;

**import** org.springframework.security.core.userdetails.UserDetailsService;

**import** org.springframework.security.core.userdetails.UsernameNotFoundException;

**import** org.springframework.stereotype.Service;

**import** com.example.demo.model.User;

**import** com.example.demo.repository.UserRepository;

@Service

**public** **class** CustomUserDetailsService **implements** UserDetailsService {

@Autowired

UserRepository userRepository;

@Override

**public** UserDetails loadUserByUsername(String username) **throws** UsernameNotFoundException {

User user = userRepository.findByUsername(username);

CustomUserDetails userDetails = **null**;

**if**(user!=**null**) {

userDetails = **new** CustomUserDetails();

userDetails.setUser(user);

}**else** {

**throw** **new** UsernameNotFoundException("user not exists with name"+username);

}

**return** userDetails;

}

}