

Spring Boot

Spring Boot is an opinionated addition to the Spring platform, focused on convention over configuration — highly useful for getting started with minimum effort and creating standalone, production-grade applications.

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

Spring Boot Primary Goals

- Provide a radically faster and widely accessible getting-started experience for all Spring development.
- Be opinionated out of the box but get out of the way quickly as requirements start to diverge from the defaults.
- Provide a range of non-functional features that are common to large classes of projects (such as embedded servers, security, metrics, health checks, and externalized configuration).
- Absolutely no code generation and no requirement for XML configuration.

Key Spring Boot features

- Spring Boot starters
- Spring Boot autoconfiguration
- Elegant configuration management
- Easy-to-use embedded servlet container support

Spring Boot Annotations

@SpringBootApplication

We use this annotation to mark the main class of a Spring Boot application.

`@SpringBootApplication` encapsulates `@Configuration`, `@EnableAutoConfiguration`, and `@ComponentScan` annotations with their default attributes.

@SpringBootApplication

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

@SpringBootApplication annotation that adds:

- @Configuration: Tags the class as a source of bean definitions for the application context.
- @EnableAutoConfiguration: Tells Spring Boot to start adding beans based on classpath settings, other beans, and various property settings. For example, if spring-webmvc is on the classpath, this annotation flags the application as a web application and activates key behaviors, such as setting up a DispatcherServlet.
- @ComponentScan: Tells Spring to look for other components, configurations, and services in the com/example package, letting it find the controllers.

Starting with Spring Initializer

- Go to <https://start.spring.io>. This service pulls in all the dependencies you need for an application and does most of the setup for you.
- Choose either Maven and the language you want to use. This guide assumes that you chose Java.
- Click Dependencies and select Spring Web.
- Click Generate.
- Download the resulting ZIP file, which is an archive of a web application that is configured with your choices.

REST API:

1. What is it?

REST stands for Representational State Transfer, and it's an architectural style for designing networked applications.

An API (Application Programming Interface) is like a set of rules that allows different software applications to communicate with each other.

2. Key Principles:

Resources: In REST, everything is treated as a resource. Resources can be objects, data, or services. For example, a REST API for a bookstore might have resources like "books," "authors," and "reviews."

HTTP Methods: REST APIs use standard HTTP methods to perform actions on these resources:

- GET: Retrieve data from the server (e.g., getting a list of books).
- POST: Create new data on the server (e.g., adding a new book).
- PUT: Update existing data on the server (e.g., editing book details).
- DELETE: Remove data from the server (e.g., deleting a book).

Statelessness: Each request from a client to a server must contain all the information needed to understand and fulfil that request. The server doesn't store any information about the client's state between requests.

Uniform Interface: REST APIs have a consistent and well-defined structure. They often use URLs to identify resources, and the response format is typically in JSON or XML.

3. How It Works:

A client (like a web browser or a mobile app) sends an HTTP request to a server, specifying the HTTP method and the URL of the resource it wants to interact with.

The server processes the request, performs the necessary action (e.g., retrieving data from a database or updating a record), and sends back an HTTP response. This response usually includes the requested data or a confirmation of the action taken.

Clients and servers communicate using standard HTTP status codes, such as 200 (OK), 201 (Created), 404 (Not Found), and 500 (Internal Server Error), to indicate the outcome of the request.

4. Use Cases:

REST APIs are commonly used for building web services, which can power a wide range of applications:

Mobile apps can use REST APIs to fetch data from a server.

Websites can interact with databases through REST APIs.

IoT devices can send and receive data using RESTful communication.

Cloud services often expose REST APIs for developers to integrate with their platforms.

5. Benefits:

Simplicity:

REST is easy to understand and implement.

Scalability:

It can handle a large number of clients and requests.

Flexibility: Clients and servers can be written in different programming languages and run on different platforms.

In essence, a REST API is a way for software applications to communicate over the internet, using a set of well-defined rules and principles. It's widely used for building web services because of its simplicity, scalability, and flexibility. Clients make requests to servers to perform actions on resources, and the server responds with data or status information. This enables a wide range of applications to work together seamlessly on the web.

In Spring Framework, both `@Controller` and `@RestController` are annotations used for creating components that handle HTTP requests. However, they serve different purposes and have some key differences:

Purpose:

@Controller :

This annotation is typically used to define a class as a Spring MVC controller. It is intended for building web applications that return HTML views. Controllers annotated with `@Controller` are used to handle HTTP requests and return views, which are typically

HTML templates or JSP pages. These views are rendered by the server and sent to the client's browser.

@RestController: This annotation, on the other hand, is used to define a class as a RESTful web service controller. It is specifically designed for building RESTful APIs that return data in various formats such as JSON or XML. Controllers annotated with @RestController are used to handle HTTP requests and return data directly to the client in a format suitable for consumption by other applications, typically without rendering HTML views.

Response Handling:

@Controller: Methods in classes annotated with @Controller typically return String values representing view names or ModelAndView objects. These values are used to render HTML views.

@RestController: Methods in classes annotated with @RestController return Java objects directly. These objects are automatically converted into the requested response format (e.g., JSON) using message converters provided by Spring. The response is usually sent as the body of the HTTP response.

Response Content Negotiation:

@Controller: When using @Controller, you often need to configure content negotiation separately if you want to support multiple response formats (e.g., JSON and XML). This can involve additional configuration.

@RestController: @RestController is configured by default to handle content negotiation and convert responses to JSON, making it easier to build APIs that support multiple formats out of the box.

```
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.bind.annotation.RequestMapping;

@RestController
public class HelloController {
    @RequestMapping("/")
    public String index() {
        return "Greetings from Spring Boot!";
    }
}
```

The class is marked as a `@RestController`, meaning it is ready for use by Spring MVC to handle web requests. `@RequestMapping` maps `/` to the `index()` method. When invoked from a browser or by using curl on the command line, the method returns pure text. That is because `@RestController` combines `@Controller` and `@ResponseBody`, two annotations that results in web requests returning data rather than a view.

Create Spring Boot Project

Step 1: Open Spring Initializr <http://start.spring.io>.

Step 2: Select the Spring Boot version

Step 2: Provide the Group name.

Step 3: Provide the Artifact Id.

Step 5: Add the dependencies Spring Web, Spring Data JPA, and H2/MySQL Database.

Step 6: Click on the Generate button. When we click on the Generate button, it wraps the specifications in a Jar file and downloads it to the local system.

Configure and Setup MySQL Database

```
server.port=8888

spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost:3306/users_database?useSSL=false
spring.datasource.username=root
spring.datasource.password=root

spring.jpa.show-sql=true
spring.jpa.database-platform=org.hibernate.dialect.MySQL8Dialect
spring.jpa.hibernate.ddl-auto=update
```

`@Entity` annotation indicates that the class is a persistent Java class.

– `@Table` annotation provides the table that maps this entity.

- @Id annotation is for the primary key.
- **@GeneratedValue annotation is used to define generation strategy for the primary key. GenerationType.AUTO means Auto Increment field.
- @Column annotation is used to define the column in database that maps annotated field.

:-(@RequestParam)

@RequestParam :- you can use the @RequestParam annotation to extract query parameters from an HTTP request.

```
@RestController
public class TestController2 {
    @GetMapping("/hi")
    public String Query(@RequestParam String id) {
        return id;
    }
    @GetMapping("/details")
    public String details(@RequestParam(name = "emp_id") int id,
        @RequestParam String name) {
        return " employee id " + id + "/n" + "employee name :" + name;
    }
}
```

(@PathVariable)

@PathVariable:-

@PathVariable annotation is used to extract values from the URI path of a URL. It allows you to capture dynamic parts of a URL and use them as method parameters in your controller methods. Here's how you can use @PathVariable in a Spring Boot application:

```

package com.example.simplebootapp;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RestController;
@RestController
public class Example3 {
    @GetMapping("/info/{name}")
    public String info(@PathVariable String name) {
        return "customer name :-" + name;
    }
    @GetMapping("/info1/{cid}")
    public String info(@PathVariable int cid) {
        return "customer id :-" + cid;
    }
    @GetMapping("/info1/{cid}/{name}")
    public String info(@PathVariable int cid, @PathVariable String name) {
        return "customer id :-" + cid + " name:" + name;
    }
}

```

Difference between @pathvariable and @requestparam annotation ?

@RequestParam:

Purpose: Used to extract values from query parameters in the URL's query string.

Syntax: @RequestParam("parameterName")

Example: If you have a URL like /example?paramName=value, you can use @RequestParam("paramName") to extract the value "value."

@PathVariable:

Purpose: Used to extract values from dynamic parts of the URL's path.

Syntax: @PathVariable("variableName")

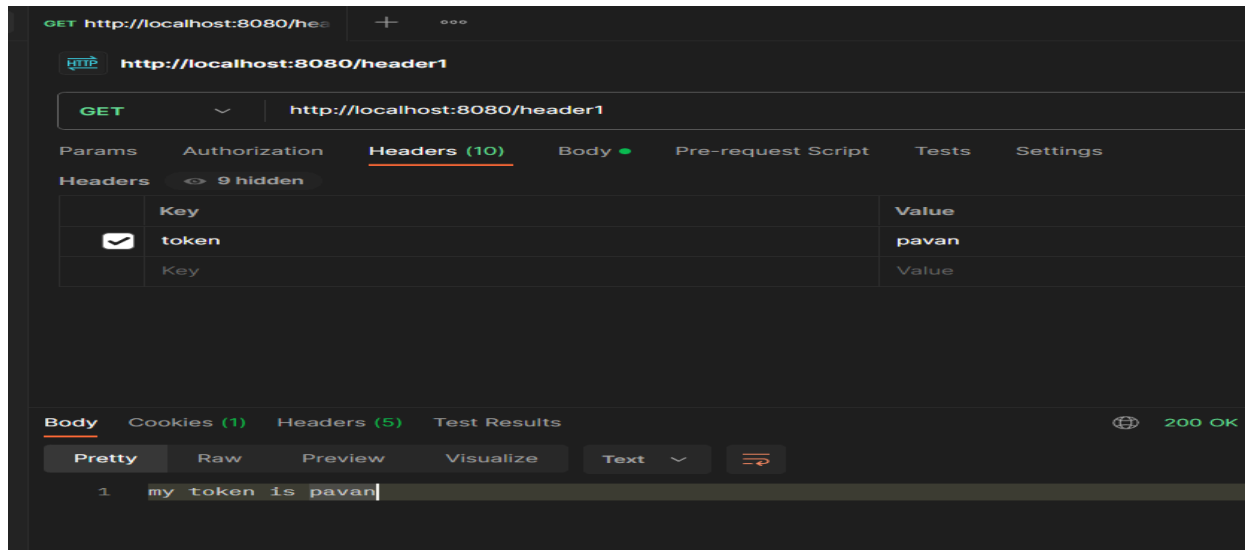
Example: If you have a URL like /example/{id}, you can use @PathVariable("id") to extract the value from the "id" path variable.

Program 5 :-(RequestHeader)

@RequestHeader:- the @RequestHeader annotation is used to access and extract values from HTTP request headers

In Spring Boot, request headers are an integral part of handling HTTP requests within your application. Spring Boot makes it easy to work with request headers through its web framework, which is built on top of the Spring MVC (Model-View-Controller) framework.

```
package com.example.simplebootapp;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestHeader;
import org.springframework.web.bind.annotation.RestController;
@RestController
public class RequestHeaderEx4 {
    @GetMapping("/header1")
    public String info(@RequestHeader String token) {
        return "my token is " + token;
    }
}
```



Media-Type

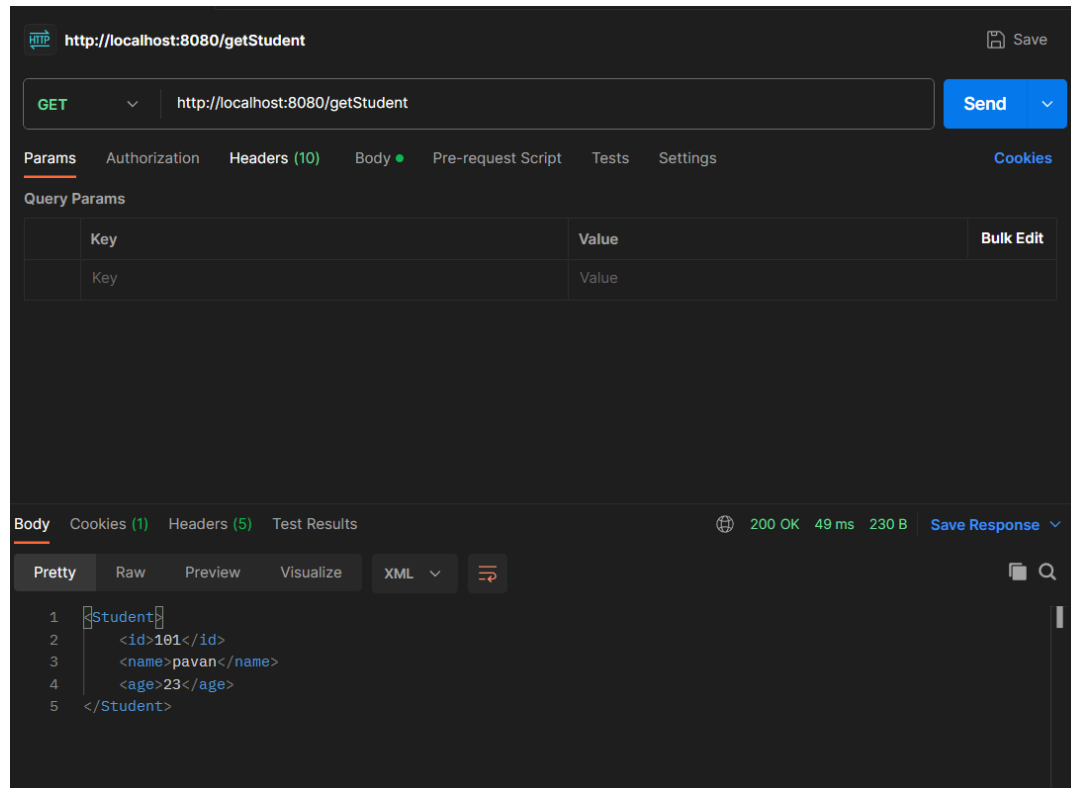
Media types, also known as MIME (Multipurpose Internet Mail Extensions) types, are identifiers used to specify the format and structure of data exchanged over the internet

```
package com.ty.simplebootmedia;
import org.springframework.http.MediaType;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;
@RestController
public class StudentController {

    @GetMapping(value="/getStudent",produces=MediaType.APPLICATION_ATOM_X
ML_VALUE)
    public Student getStudent() {
        Student student = new Student();
        student.setId(101);
        student.setName("divya");
        student.setAge(28);
        return student;
    }
}
```

}

OutPut:-



Response Structure:-

- What my response should contain to my customers.
- For my programs I'll design my response structure.

To build a response structure

1. We should build a class (class name respond structure (industry standards))
2. In this class we will define all parameters that we want to send to customers.

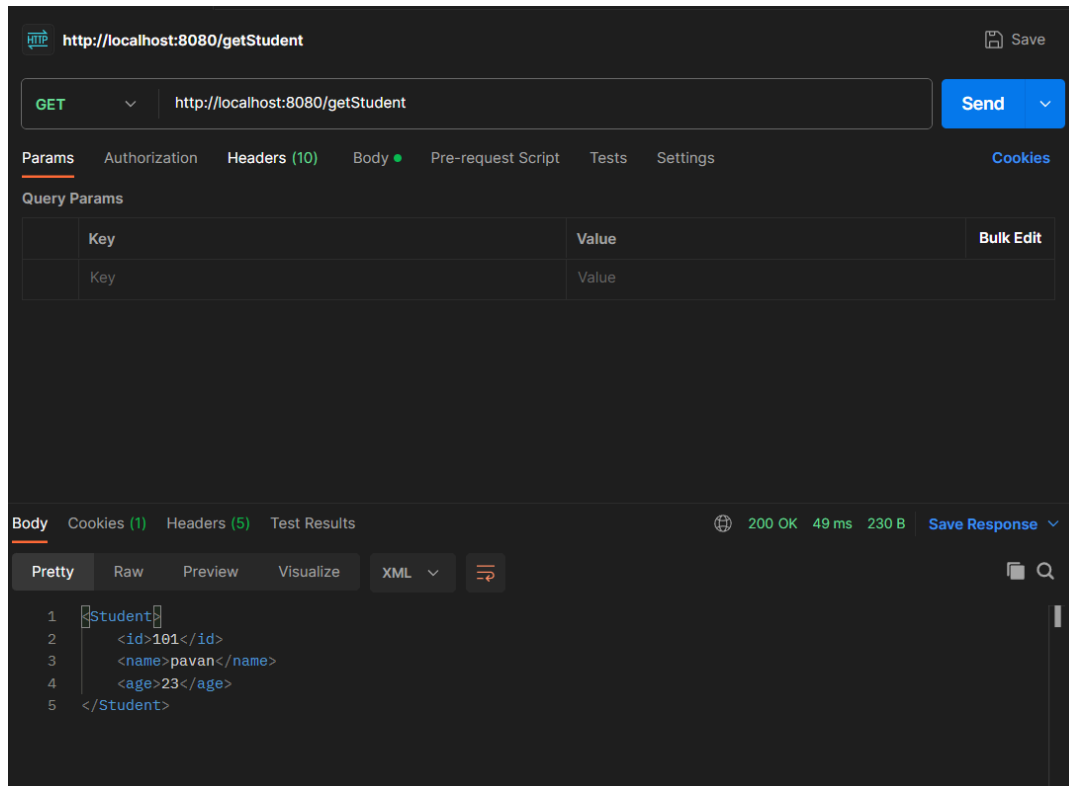
Media-Type

Media types, also known as MIME (Multipurpose Internet Mail Extensions) types, are identifiers used to specify the format and structure of data exchanged over the internet

```
package com.ty.simplebootmedia;
import org.springframework.http.MediaType;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;
@RestController
public class StudentController {

    @GetMapping(value="/getStudent",produces=MediaType.APPLICATION_ATOM_X
ML_VALUE)
    public Student getStudent() {
        Student student = new Student();
        student.setId(101);
        student.setName("pavan");
        student.setAge(23);
        return student;
    }
}
```

OutPut:-



Response Structure:-

- What my response should contain to my customers.
- For my programs I'll design my response structure.

To built a response structure

1. We should build a class (class name respond structure (industry standards))
2. In this class we will define all parameters that we want to send to customers.

EX:-

```
Class ResponseStructure<T>{  
Private int statusCode;  
Private String message;
```

Private T data;

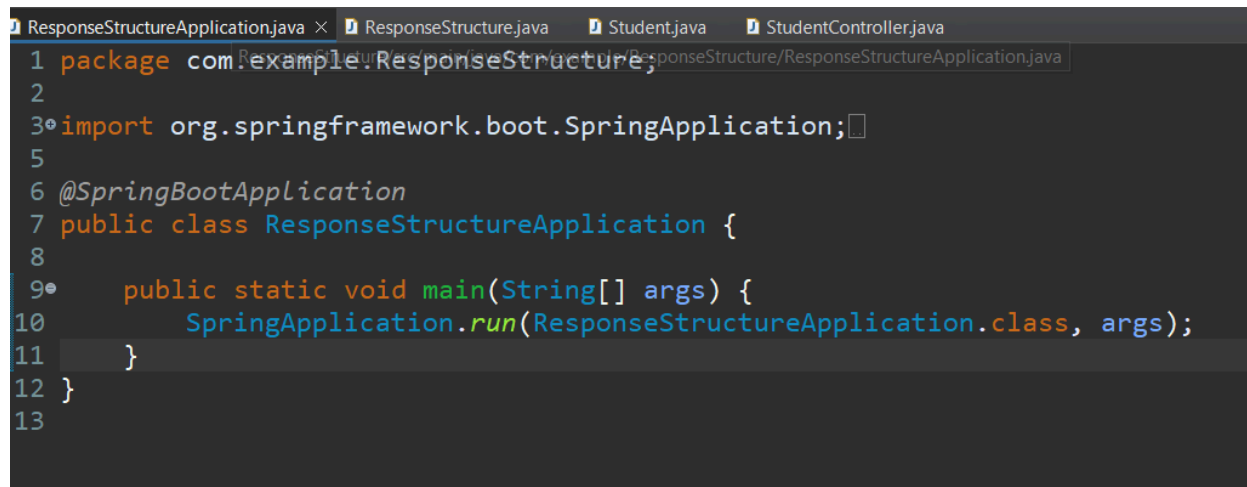
//data type of data is generic

//setter and getter

}

- If your response structure is Employee type your data will be of type Employee
- If your response structure is Student type your data will be of type Student
- If your response structure is Collection type your data will be of type Collection
- If your response structure is String type your data will be of type String

Code:-



```
1 package com.example.ResponseStructure;
2
3 import org.springframework.boot.SpringApplication;
4
5
6 @SpringBootApplication
7 public class ResponseStructureApplication {
8
9     public static void main(String[] args) {
10         SpringApplication.run(ResponseStructureApplication.class, args);
11     }
12 }
13
```

```

ResponseStructureApplication.java  *ResponseStructure.java × Student.java StudentController.java
1 package com.example.ResponseStructure;
2
3 public class ResponseStructure<T> {
4
5     private int statusCode;
6     private String message;
7     private T data;
8     //getters and setters
9
10 }
11

```

```

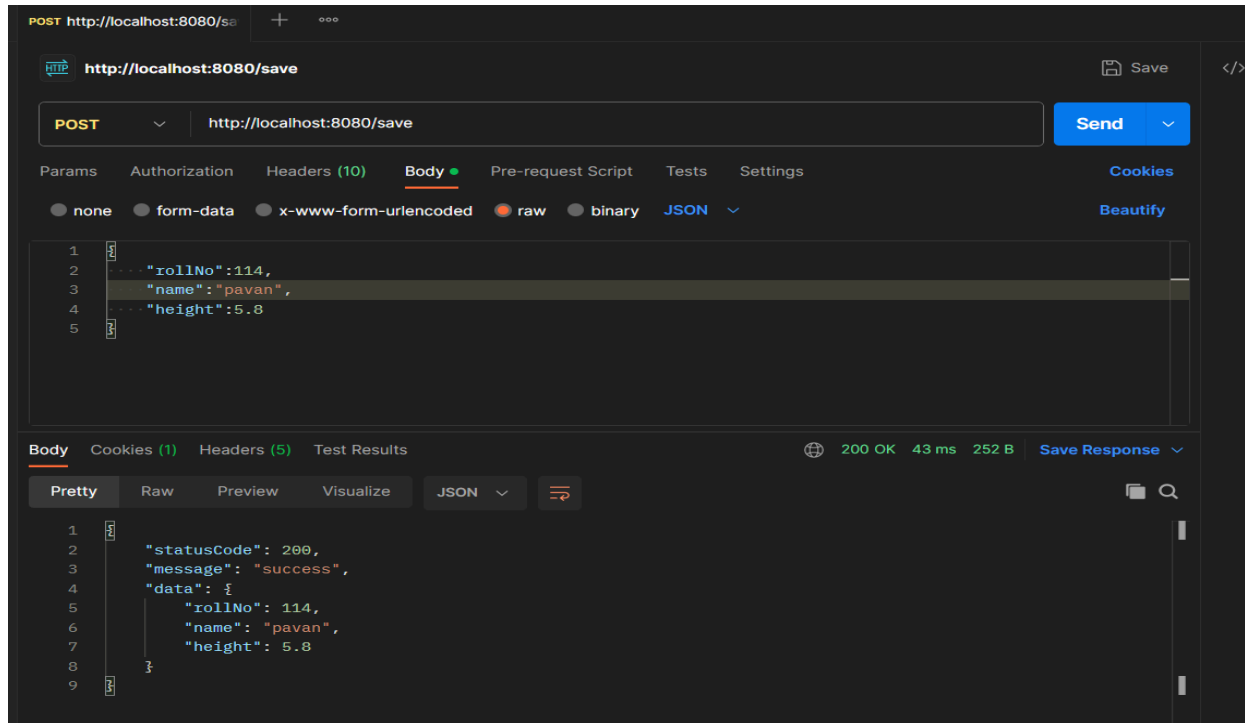
ResponseStructureApplication.java  *ResponseStructure.java × Student.java × StudentController.java
1 package com.example.ResponseStructure;
2
3 public class Student {
4     private int rollNo;
5     private String name;
6     private double height;
7     //getters and setters
8 }

```

```

ResponseStructureApplication.java  ResponseStructure.java Student.java StudentController.java ×
1 package com.example.ResponseStructure;
2
3 import org.springframework.web.bind.annotation.PostMapping;
4 import org.springframework.web.bind.annotation.RequestBody;
5 import org.springframework.web.bind.annotation.RestController;
6
7 @RestController
8 public class StudentController {
9
10     @PostMapping("/save")
11     public ResponseStructure<Student> saveStudent(@RequestBody Student student) {
12
13         ResponseStructure<Student> responseStructure = new ResponseStructure<Student>();
14         responseStructure.setStatusCode(200);
15         responseStructure.setMessage("success");
16         responseStructure.setData(student);
17         return responseStructure;
18     }
19
20 }

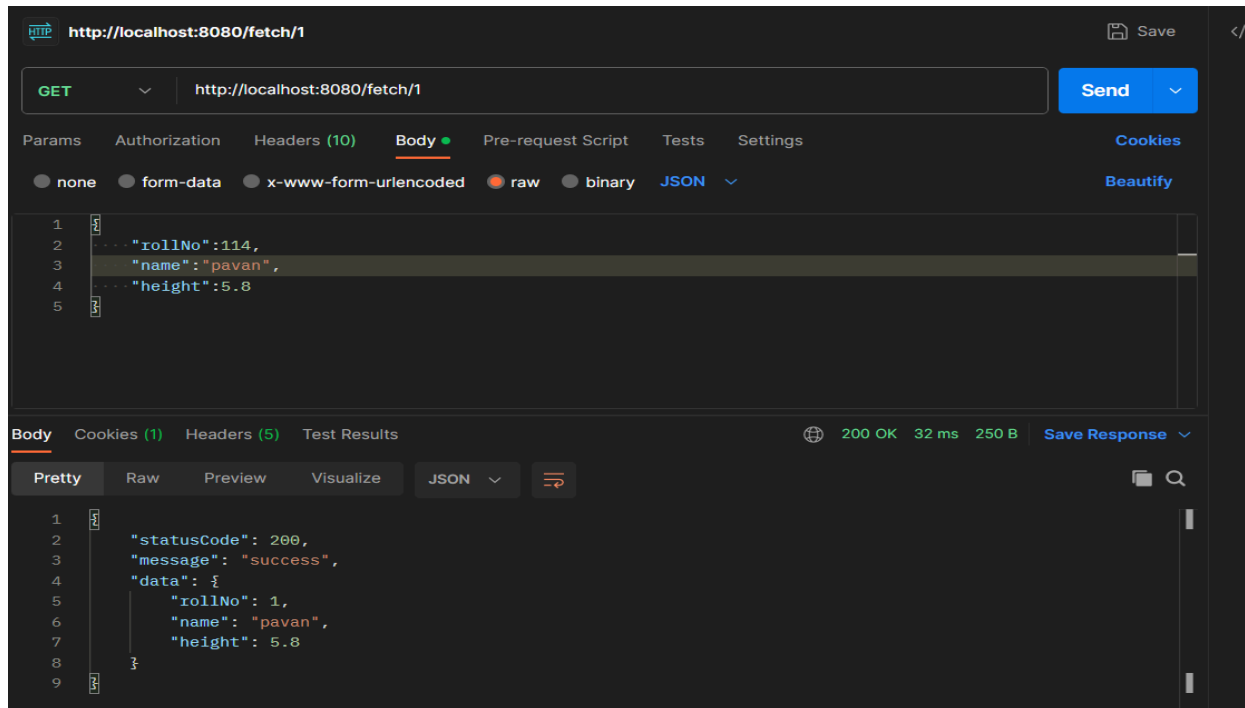
```



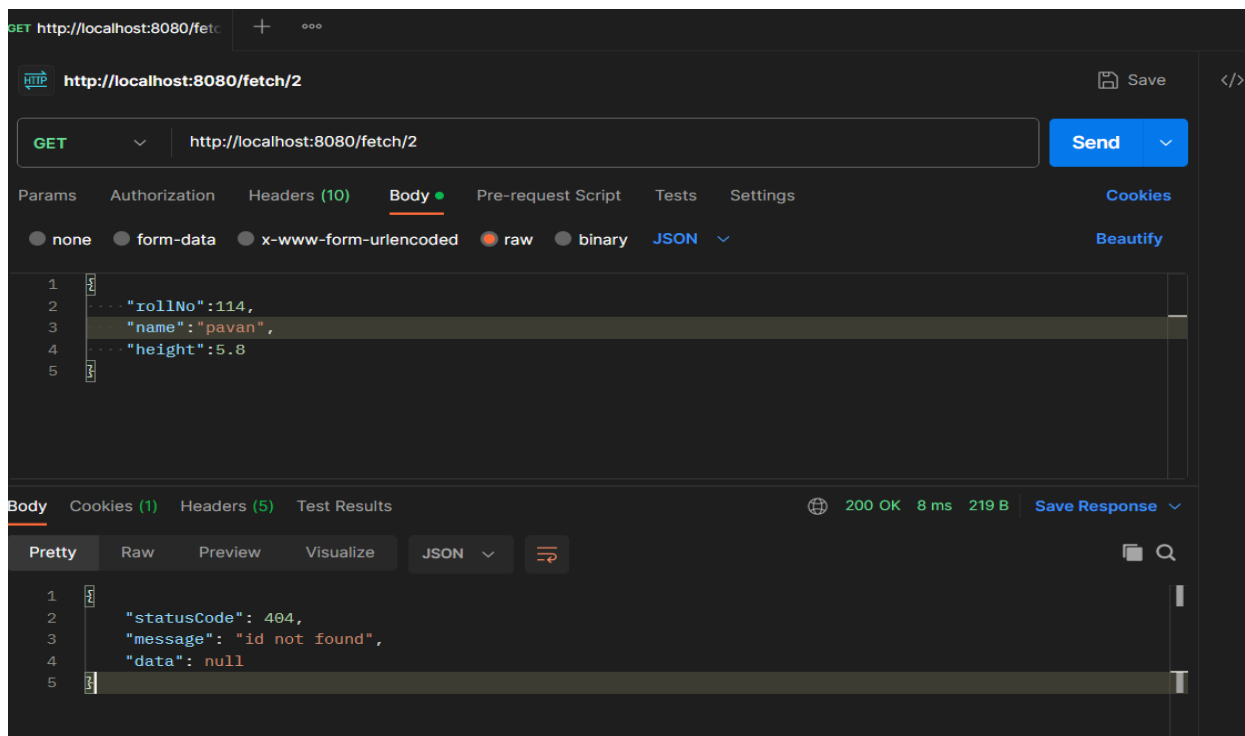
What and Why ResponseEntity:-

```
1 ResponseStructureApplication.java 2 ResponseStructure.java 3 Student.java 4 StudentController.java X
22
23 * @GetMapping("/fetch/{roll}")
24 public ResponseStructure<Student> getByRollNO(@PathVariable int roll) {
25     if (roll == 1) {
26         Student student = new Student();
27         student.setRollNo(1);
28         student.setName("pavan");
29         student.setHeight(5.8);
30         ResponseStructure<Student> responseStructure = new ResponseStructure<Student>();
31         responseStructure.setData(student);
32         responseStructure.setMessage("success");
33         responseStructure.setStatusCode(200);
34         return responseStructure;
35     } else {
36         ResponseStructure<Student> responseStructure = new ResponseStructure<Student>();
37         responseStructure.setData(null);
38         responseStructure.setMessage("id not found");
39         responseStructure.setStatusCode(404);
40         return responseStructure;
41     }
42 }
43 }
44 }
```

Output:- 1

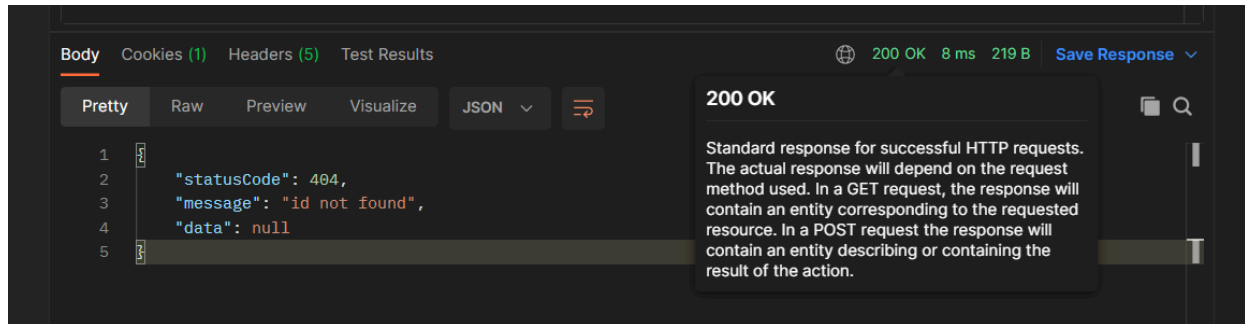


Output 2



NOTE:-

- I Am changing the body of the response structure and I'm not altering the entire HTTP response
- To alter the entire http response I need the help of the response entity.



Response Entity:-

- It contains a status code.
- This response entity contains a body.
- And it contains data.
- It is also going to contain the headers like content type and actual content
- Response entity is a structure of http response.
- Instead of returning http response we will return response entity.
- Headers are optional.

Response entity

- Body of a response entity should be responseStructure.
- Status code can be anything.
- I need to create an object of response Entity of response Structure.

EX:-

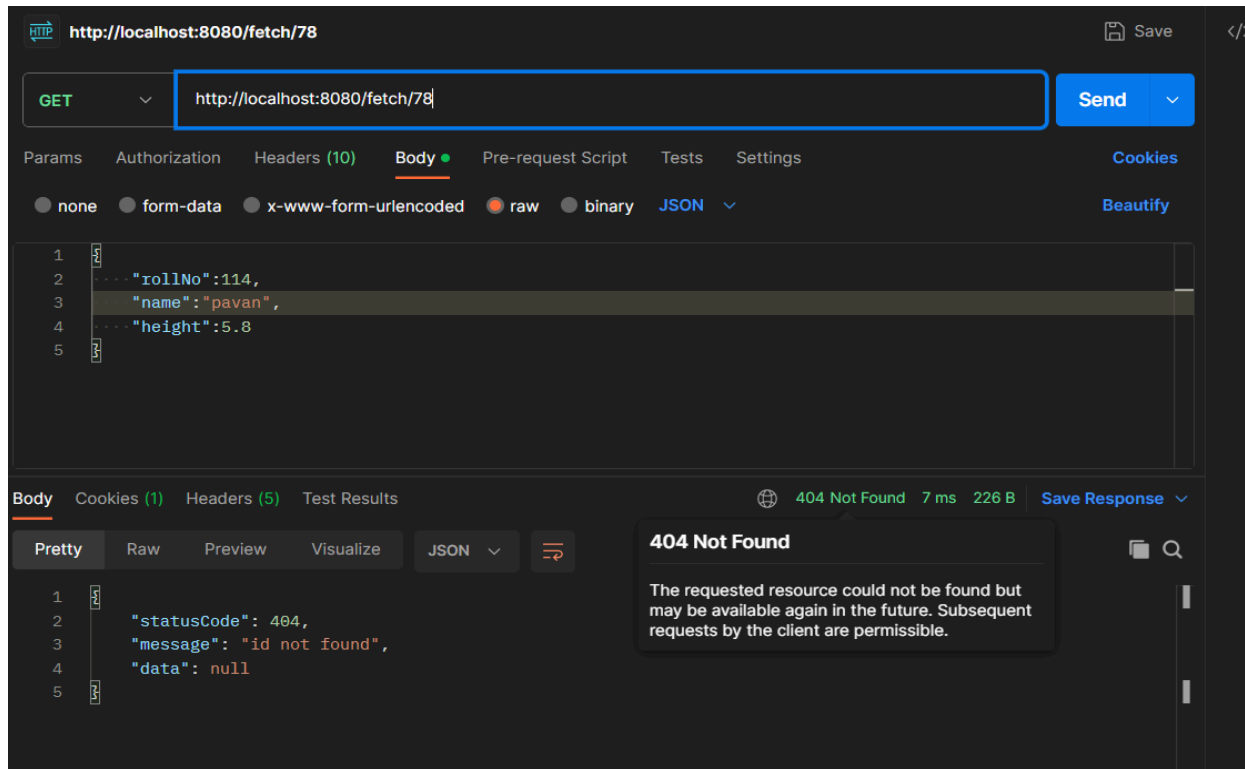
1. New ResponseEntity <ResponseStructure>(data,statusCode)
2. New ResponseEntity<ResponseStructure<Student>>(data,statusCode)
3. New ResponseEntity<ResponseStructure<Employee>>(data,statusCode)

This is how we are creating response entity

Program for ResponseEntity:-

```
ResponseStructureApplication.java  ResponseStructure.java  Student.java  StudentController.java X
5
6  @GetMapping("/fetch/{roll}")
7  public ResponseEntity<ResponseStructure<Student>> getByRollNO(@PathVariable int roll) {
8      if (roll == 1) {
9          Student student = new Student();
10         student.setRollNo(1);
11         student.setName("pavan");
12         student.setHeight(5.8);
13         ResponseStructure<Student> responseStructure = new ResponseStructure<Student>();
14         responseStructure.setData(student);
15         responseStructure.setMessage("success");
16         responseStructure.setStatusCode(200);
17         return new ResponseEntity<ResponseStructure<Student>>(responseStructure, HttpStatus.OK);
18     }
19     else {
20         ResponseStructure<Student> responseStructure = new ResponseStructure<Student>();
21         responseStructure.setData(null);
22         responseStructure.setMessage("id not found");
23         responseStructure.setStatusCode(404);
24         return new ResponseEntity<ResponseStructure<Student>>([responseStructure, HttpStatus.NOT_FOUND]);
25     }
26 }
```

Output:-



NOTE:-

- It is recommended to return responseEntity and using of responseStructure is a bad practice.
- The response entity of capability or capacity to alter the entire http response
- Coming to the response structure we can alter only the body part of the http response.

Creating and handling custom exception:-

```

Student.java StudentController.java × ResponseStructure.java HandlingCustomExceptionApplication.java MyAppExceptionHandler.java
3•import org.springframework.http.HttpStatus;
4 import org.springframework.http.ResponseEntity;
5 import org.springframework.web.bind.annotation.GetMapping;
6 import org.springframework.web.bind.annotation.RequestBody;
7 import org.springframework.web.bind.annotation.RestController;
8
9 @RestController
10 public class StudentController {
11
12• @GetMapping("/fetch")
13 public ResponseEntity<ResponseStructure<Student>> getStudentByrollNo(@RequestBody Student student) {
14     String stName = student.getName();
15     stName = stName + ":" + stName.length();
16 // student.setName(stName);
17     ResponseStructure<Student> responseStructure = new ResponseStructure<Student>();
18     responseStructure.setMessage("success");
19     responseStructure.setStatusCode(200);
20     responseStructure.setData(student);
21     return new ResponseEntity<ResponseStructure<Student>>(responseStructure, HttpStatus.OK);
22 }
23 }
24

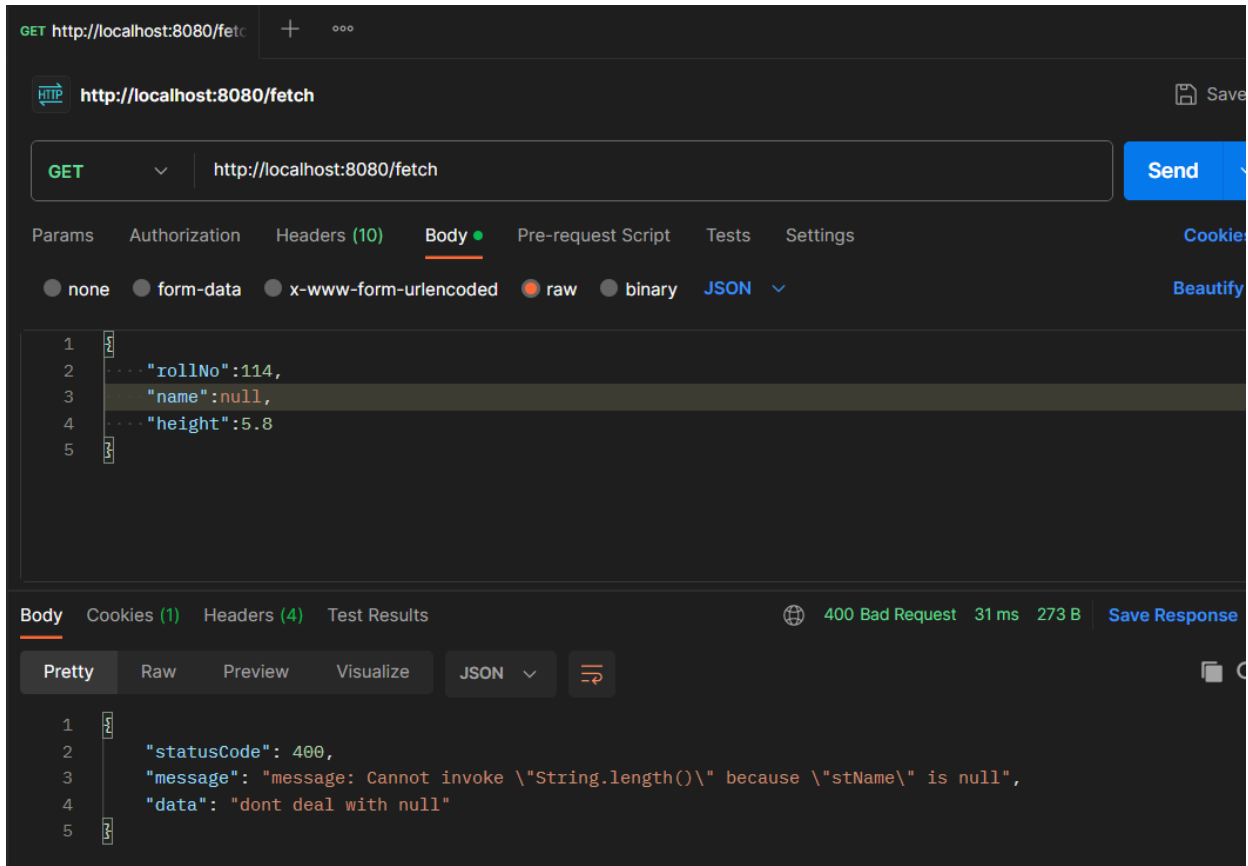
```

```

Student.java StudentController.java ResponseStructure.java HandlingCustomExceptionApplication.java MyAppExceptionHandler.java ×
1 package com.custom.handlingcustomexception;
2
3•import org.springframework.http.HttpStatus;
4 import org.springframework.http.ResponseEntity;
5 import org.springframework.web.bind.annotation.ControllerAdvice;
6 import org.springframework.web.bind.annotation.ExceptionHandler;
7 import org.springframework.web.servlet.mvc.method.annotation.ResponseEntityExceptionHandler;
8
9 @ControllerAdvice
10 public class MyAppExceptionHandler extends ResponseEntityExceptionHandler {
11
12• @ExceptionHandler({NullPointerException.class})
13 public ResponseEntity<ResponseStructure<String>> handleNullPointerException(
14     NullPointerException nullPointerException) {
15     ResponseStructure<String> responseStructure = new ResponseStructure<String>();
16     responseStructure.setStatusCode(HttpStatus.BAD_REQUEST.value());
17     responseStructure.setData("dont deal with null");
18     responseStructure.setMessage("message: " + nullPointerException.getMessage());
19     return new ResponseEntity<ResponseStructure<String>>(responseStructure, HttpStatus.BAD_REQUEST);
20 }
21
22 }

```

Output:-



Custom methods in repository:-

```
package com.custom.custommethods;
import java.util.List;
import org.springframework.data.jpa.repository.JpaRepository;
public interface EmployeeRepository extends JpaRepository<Employee, Integer> {
    public Employee findByEmail(String email);
    public Employee findByPhone(long phone);
    public List<Employee> findByName(String name);
    public Employee findByEmailAndPassword(String email, String password);
    public List<Employee> findByAgeGreaterThan(int ageValue);
}
```

We can write our customised methods in our employeeRepository class and our spring container will write the queries in the backend .

Ex:- **findByPhone**(long phone)

- And the method name should start with findBy itself.
- And spring container will create the query.

Program 1:-

```
@RestController
public class EmployeeController {
    @Autowired
    private EmployeeRepository employeeRepository;
    @GetMapping("/getByEmail/{email}")
    public Employee findByEmail(@PathVariable String email) {
        return employeeRepository.findByEmail(email);
    }
}
```

Program 2:-

Input:-

```
@GetMapping("/getByEmail/{phone}")
public Employee findByPhone(@PathVariable String phone) {
    return employeeRepository.findByEmail(phone);
}
```

Program 3:-

```
@GetMapping("/getByName/{name}")
    public List<Employee> findByName(@PathVariable String name) {
        return employeeRepository.findByName(name);
    }
```

Program 4:-

```
@GetMapping("/valid/{pass}/{email}")
    public Employee findByPhone(@PathVariable String email,
    @PathVariable String pass) {
        return employeeRepository.findByEmailAndPassword(email,
pass);
    }
```

Output:-

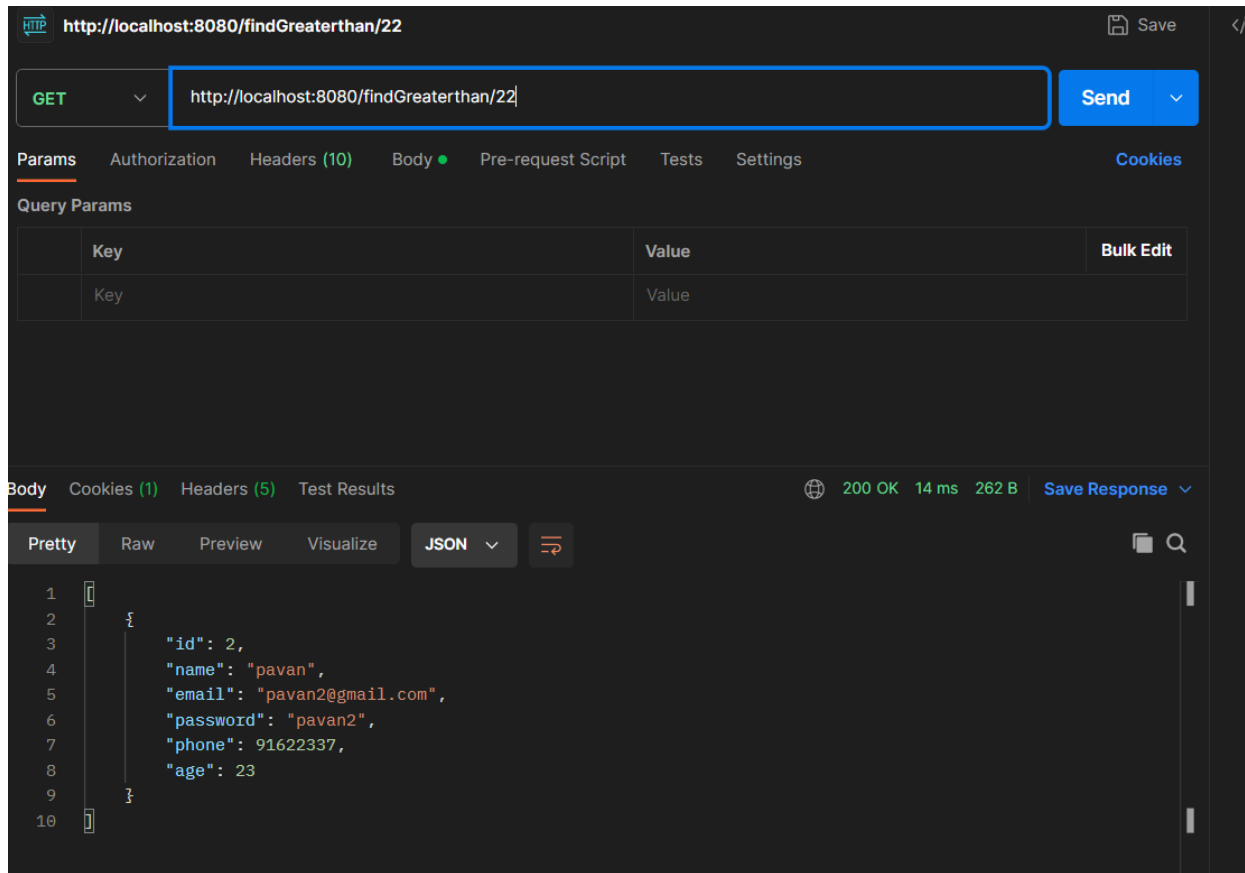
The screenshot shows a web browser interface for an HTTP client. The address bar displays the URL `http://localhost:8080/valid/pavan/pavan@gmail.com`. The request method is `GET`. The response status is `200 OK` with a response time of `31 ms` and a size of `304 B`. The response body is displayed in JSON format:

```
{
  "id": 1,
  "name": "pavan",
  "email": "pavan@gmail.com",
  "password": "pavan",
  "phone": 9160004067,
  "age": 22
}
```

Program 5:-

```
@GetMapping("/findGreaterthan/{age}")
public List<Employee> findGreaterAge(@PathVariable int age) {
    return employeeRepository.findByAgeGreaterThan(age);
}
```

Output:-



Custom Query In Repository

Using @query annotation i can write customised queries in the format of string

Example:-

```
Interface studentRepository extends  
jpaRepository<Student,Integer>{  
    @Query("select s from Student s where s.roll=1")  
    list<student>getInfo(){
```

}

}

You have to write it with in repository at the time i can write customised query

- We can write our customised queries in two ways

1) named parameter.

Scenario-1

```
@Query("select s from Student s where s.name=:name")  
List<Student> getByName1(String name);
```

Scenario-2

```
@Query("select s from Student s where s.email=:email and  
s.passsword=:password")  
Student validateStudent(String email, String password);
```

Scenario-3

```
@Query("select s from Student s where s.name=:name")  
List<Student> getByName2(@Param("name") String nm);
```

Note:- when we want to give different name for the variable name instead of same name of the parameter in the query.

->index parameter.

Program 1:-

```
@Query("select s from Student s")
```

```
List<Student> getAllStudents();
```

Program 2:-

```
@Query("select s from Student s where s.name=?1")  
List<Student> getByName(String name);
```

Here it will follow 1 based index based on the position of variables .

Program 3:-

```
@Query("select s from Student s where s.name=?1")  
List<Student> getByroll(String name, int roll);
```

14 8 34

Annotations

Deployment descriptor

- **@Autowired:** Spring provides annotation-based auto-wiring by providing @Autowired annotation. It is used to autowire spring bean on setter methods, instance variable, and constructor. When we use @Autowired annotation, the spring container auto-wires the bean by matching data-type.
- **@Configuration:** It is a class-level annotation. The class annotated with @Configuration used by Spring Containers as a source of bean definitions.
- **@ComponentScan:** It is used when we want to scan a package for beans. It is used with the annotation

@Configuration. We can also specify the base packages to scan for Spring Components.

- @Component: It is a class-level annotation. It is used to mark a Java class as a bean. A Java class annotated with @Component is found during the classpath. The Spring Framework pick it up and configure it in the application context as a Spring Bean.
- @Controller: The @Controller is a class-level annotation. It is a specialisation of @Component. It marks a class as a web request handler. It is often used to serve web pages. By default, it returns a string that indicates which route to redirect. It is mostly used with @RequestMapping annotation.
- @Service: It is also used at class level. It tells the Spring that the class contains business logic.
- @Repository: It is a class-level annotation. The repository is a DAOs (Data Access Object) that access the database directly. The repository does all the operations related to the database.
- Spring Boot Annotations
- @EnableAutoConfiguration: It auto-configures the bean that is present in the classpath and configures it to run the methods. The use of this annotation is reduced in Spring

Boot 1.2.0 release because developers provided an alternative of the annotation, i.e. `@SpringBootApplication`.

- `@SpringBootApplication`: It is a combination of three annotations `@EnableAutoConfiguration`, `@ComponentScan`, and `@Configuration`.

Spring MVC and REST Annotations

- `@RequestMapping`: It is used to map the web requests. It has many optional elements like consumes, header, method, name, params, path, produces, and value. We use it with the class as well as the method.
- `@GetMapping`: It maps the HTTP GET requests on the specific handler method. It is used to create a web service endpoint that fetches It is used instead of using: `@RequestMapping(method = RequestMethod.GET)`
- `@PostMapping`: It maps the HTTP POST requests on the specific handler method. It is used to create a web service endpoint that creates It is used instead of using: `@RequestMapping(method = RequestMethod.POST)`
- `@PutMapping`: It maps the HTTP PUT requests on the specific handler method. It is used to create a web service endpoint that creates or updates It is used instead of using: `@RequestMapping(method = RequestMethod.PUT)`
- `@DeleteMapping`: It maps the HTTP DELETE requests on the specific handler method. It is used to create a web service endpoint that deletes a resource. It is used instead of using: `@RequestMapping(method = RequestMethod.DELETE)`

- **@PatchMapping**: It maps the HTTP PATCH requests on the specific handler method. It is used instead of using: `@RequestMapping(method = RequestMethod.PATCH)`
- **@RequestBody**: It is used to bind HTTP request with an object in a method parameter. Internally it uses HTTP `MessageConverters` to convert the body of the request. When we annotate a method parameter with `@RequestBody`, the Spring framework binds the incoming HTTP request body to that parameter.
- **@ResponseBody**: It binds the method return value to the response body. It tells the Spring Boot Framework to serialize a return an object into JSON and XML format.
- **@PathVariable**: It is used to extract the values from the URL. It is most suitable for the RESTful web service, where the URL contains a path variable. We can define multiple `@PathVariable` in a method.
- **@RequestParam**: It is used to extract the query parameters from the URL. It is also known as a query parameter. It is most suitable for web applications. It can specify default values if the query parameter is not present in the URL.
- **@RequestHeader**: It is used to get the details about the HTTP request headers. We use this annotation as a method parameter. The optional elements of the annotation are `name`, `required`, `value`, `defaultValue`. For each detail in the header, we should specify separate annotations. We can use it multiple time in a method
- **@RestController**: It can be considered as a combination of `@Controller` and `@ResponseBody` annotations. The `@RestController` annotation is itself annotated with the

`@ResponseBody` annotation. It eliminates the need for annotating each method with `@ResponseBody`.

- `@RequestAttribute`: It binds a method parameter to request attribute. It provides convenient access to the request attributes from a controller method. With the help of `@RequestAttribute` annotation, we can access objects that are populated on the server-side.

HTTP response status codes

HTTP response status codes indicate whether a specific [HTTP](#) request has been successfully completed. Responses are grouped in five classes:

1. [Informational responses](#) (100 – 199)
2. [Successful responses](#) (200 – 299)
3. [Redirection messages](#) (300 – 399)
4. [Client error responses](#) (400 – 499)
5. [Server error responses](#) (500 – 599)

Jakarta EE vs J2EE vs Java EE

- Jakarta EE (Jakarta Platform, Enterprise Edition)
- Java EE (Java Platform, Enterprise Edition)
- J2EE (Java 2 Platform, Enterprise Edition)

Jakarta EE

- Jakarta Server Pages (JSP): Used to create dynamic web pages.

- Jakarta Standard Tag Library (JSTL): Used to show dynamic information on web pages.
- Jakarta Enterprise Beans (EJB): Used to create enterprise applications.
- Jakarta Restful Web Services (JAX-RS): Used to create RESTful web services.
- Jakarta Bean Validation (JSR-303): Used to validate Java objects.
- Jakarta Contexts and Dependency Injection (CDI): Used to inject dependencies into Java objects.
- Jakarta Persistence (JPA): Used to persist Java objects into a database.
- Jakarta Server Faces (JSF): Used to create user interfaces for web applications.
- Jakarta Expression Language (EL): Used to evaluate expressions in Java applications.
- Jakarta WebSocket: Used to create WebSocket applications.
- Jakarta Batch: Used to create batch applications.
- Jakarta Concurrency: Used to create concurrent applications.

Spring 5 - Java EE (javax.)

Spring 6 - Jakarta EE (jakarta.)

Spring Framework

Spring Framework is an open source Java platform that provides comprehensive infrastructure support for developing Java applications. Spring Framework is the most popular application framework for enterprise Java. Spring Framework is an open source framework for the Java platform. It is used to build Java enterprise applications.

Tight Coupling vs Loose Coupling

- Tight Coupling: When two classes are tightly coupled, they are dependent on each other. If one class changes, the other class will also change.
- Loose Coupling: When two classes are loosely coupled, they are not dependent on each other. If one class changes, the other class will not change.

In the example below, GameRunner class has a dependency on GamingConsole. Instead of wiring game object to a specific class such as MarioGame, we can use GamingConsole interface to make it loosely coupled. So that, we don't need to change our original code. In the future, we can create classes that implements GamingConsole interface (Polymorphism) and use it.

```
private final GamingConsole game; // Loosely coupled, it's not a specific game anymore. Games implement GamingConsole interface. Polymorphism.
```

```
public GameRunner(GamingConsole game) {
```

```
    this.game = game;
```

```
}
```

```
public void run() {
```

```
    System.out.println("Running game: " + game);
```

```
    game.up();
```

```
    game.down();
```

```
    game.left();
```

```
    game.right();
```

```
}
```

```
}
```

Tightly Coupling



Traveller

```
public class Traveller {  
    // Car car = null; // Tightly coupled with Car. If we want to change the car, we  
    // need to change the Traveller class.  
    Bike bike = null;  
  
    public Traveller() {  
        // this car = new Car();  
        this.bike = new Bike();  
    }  
  
    public void startJourney() {  
        // this car move();  
        this.bike.move();  
    }  
}
```



Car

```
public class Car {  
    public void move() {  
        System.out.println("Car is moving");  
    }  
}
```



Bicycle

```
public class Bicycle {  
    public void move() {  
        System.out.println("Bicycle is moving");  
    }  
}
```



Bike

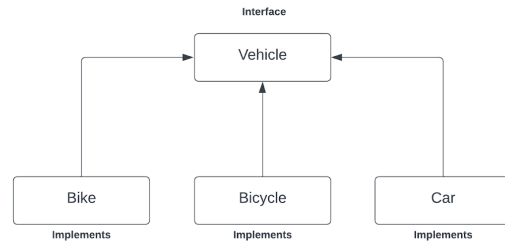
```
public class Bike {  
    public void move() {  
        System.out.println("Bike is moving");  
    }  
}
```

Problems
with
Tightly Coupling

- Whenever we want to change the instance of the Traveller class, we need to go and change the instance for the Vehicle.
- If we are building complex application, it will hard to manage and tightly coupling will cause bugs.
- Classes are highly depends on other class implementations.

Loosely Coupling

To achieve loose coupling, we can use **abstract classes** or **interfaces**.



Bike, Bicycle and Car classes implements Vehicle interface.

```
public interface Vehicle {  
    void move();  
}
```

We implemented Vehicle dynamically to the Traveller class by using interface reference.

```
public class Traveller {  
    private Vehicle vehicle;  
  
    public Traveller(Vehicle vehicle) {  
        this.vehicle = vehicle;  
    }  
  
    public void startJourney() {  
        // this car.move();  
        this.vehicle.move();  
    }  
}
```

With loose coupling, we achieved that whenever **Traveller** wants to change the **Vehicle**, we won't modify the original class but pass new instance of the **Vehicle** traveller wants to use.

```
public class Client {  
    public static void main(String[] args) {  
        // Creating an instance.  
        Vehicle vehicle = new Car();  
        Car car = new Car();  
        Traveller traveller = new Traveller(vehicle);  
        traveller.startJourney();  
    }  
}
```

Spring Container

What is a Spring Container?

Spring Container is the core of the Spring Framework. The Spring Container will create the objects, wire them together, configure them, and manage their complete life cycle from creation till destruction. The Spring Container uses DI to manage the components that make up an application.

The Spring Container manages Spring beans and their life cycle.

We have created POJOs (Plain Old Java Objects) and Configuration file. We passed them as inputs into Spring IoC Container. The Configuration file contains all of the beans. The output of Spring IoC Container is called Ready System.

JVM (Java Virtual Machine) is the container that runs the Java application. Spring Container is the container that runs the Spring application. JVM contains Spring Context.

Spring IoC container creates the runtime system for us. Creates Spring Context and manages beans for us.

Spring Container—Spring Context—Spring IoC Container

Different Types of IoC Containers

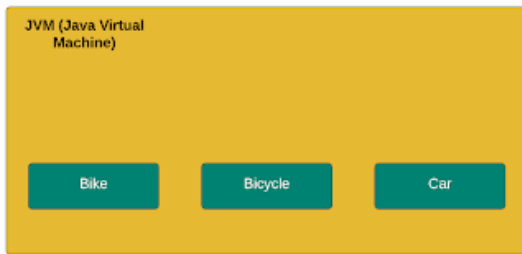
Spring provides two types of IoC containers:

- **BeanFactory Container:** Basic IoC container provided by Spring. It is the root interface for accessing a Spring BeanFactory. It is the simplest container provided by Spring.
- **ApplicationContext Container:** It is the advanced container provided by Spring. It is built on top of the BeanFactory container. It adds more enterprise-specific functionality like the ability to resolve textual messages from a properties file and the ability to publish application events to interested event listeners.

Most of the Spring applications use ApplicationContext container. It is recommended to use ApplicationContext container over BeanFactory container. (Web applications, web services, REST API and microservices)

Diagram Example

Creating Objects Manually



- Whenever we create objects manually by using **new** keyword, those objects will be stored in **JVM**. In the JVM, we have heap memory. When we create new objects, they will be stored in the heap memory.

Problem:

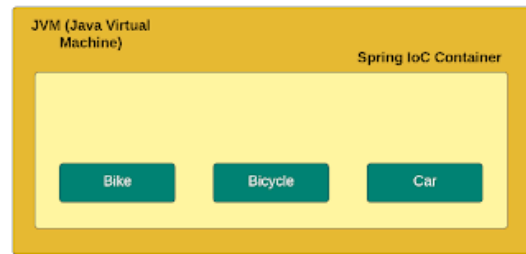
- If we have hundreds of classes, we need to create new instances using **new** keyword. This is not a good practice.

Solution:

- Spring framework will create and manage objects we want to use in our application. **Spring IoC Container** feature will create and manage the lifecycle of the objects.

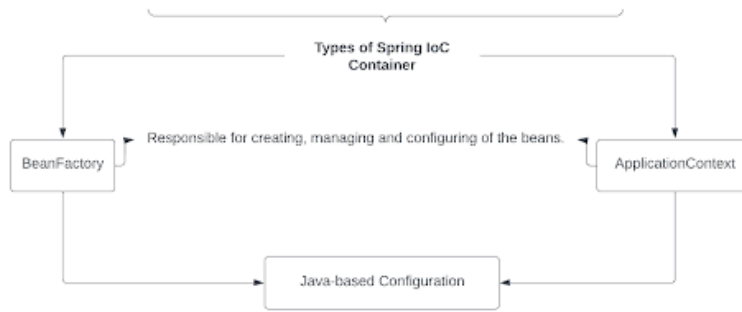
Objects inside JVM called **Java Beans**.

Creation of Objects Managed by Spring Framework



- Spring IoC** is responsible for creating the objects. Whenever object is managed by Spring, it is called **Spring Beans**.
- By using feature called **Dependency Injection (DI)**, Spring IoC container is injecting one object into another object.
- Spring IoC container manages the bean's entire lifecycle from **creation** to **destruction**.
- Container uses the information provided in the configuration file (or annotations) to create the objects, and it uses dependency injection to supply the object with their dependencies.
- The configuration metadata is represented in **XML**, **Java annotations**, or **Java code**.
- IoC container manages object more loosely coupled.

Objects inside Spring IoC called **Spring Beans**.



- Create a configuration class with **@Configuration** annotation.
- Create a method and annotated it with **@Bean** annotation.
- Create a Spring IoC Container (**ApplicationContext**) and retrieve the Spring bean from the Spring IoC container.

@Configuration

- We can configure bean definitions inside this class that uses this annotation.

@Bean

- We created a method that returns the object of the class we want Spring to manage. We need to create only one time, Spring will manage it. We don't have to create the instance of the object again.

ApplicationContext

- Creates the **Spring IoC Container**.
- Reads the **configuration** class with **@Configuration** annotation.
- Creates and manages the Spring beans.

In **Java-based Configuration**, we need to create object of the class manually. We also have to inject dependencies manually.

```
@Configuration
public class AppConfig {
```

```
    @Bean
    public Vehicle car() {
        return new Car();
    }
```

```
    @Bean
    public Vehicle bike() {
        return new Bike();
    }
}
```

```
public class Client {
```

```
    public static void main(String[] args) {
        // Creating Spring IoC Container
        // Read the configuration class
        // Create and manage the Spring Beans
        ApplicationContext applicationContext = new
        AnnotationConfigApplicationContext(AppConfig.class); //
        Providing the configuration class.

        // Retrieve Spring Beans from the Spring IoC
```

Difference between Java Bean, POJO and Spring Bean

POJO (Plain Old Java Object)

- A POJO is a simple Java object, without any dependency. It does not extend any class and does not implement any interface. It is a simple Java object that is used to transfer data from one layer to another.
- POJOs do not have any business logic. They are just simple data objects.
- POJOs are not thread-safe. They are not synchronized.
- POJOs do not have any dependency. They do not have any reference to any other object.

Java Bean

- A Java Bean is a POJO that follows some additional rules.
- Java Beans are serializable.
- Java Beans have a no-argument constructor.
- Java Beans have getter and setter methods.
- Java Beans are thread-safe. They are synchronized.
- Java Beans have a dependency. They have a reference to other objects.

Spring Bean

- A Spring Bean is a POJO that is managed by Spring IoC Container (ApplicationContext or BeanFactory).

Spring Bean Configuration

Spring Bean Configuration is the process of defining beans. The Spring Bean Configuration can be done in two ways:

- XML Based Configuration
- Annotation Based Configuration

How to list all beans managed by Spring Container?

We can list all beans managed by Spring Container using the following code:

```
public class ExampleClass {  
  
    public static void main(String[] args) {
```

```
ApplicationContext context = new
AnnotationConfigApplicationContext(AppConfig.class);

String[] beanNames = context.getBeanDefinitionNames();

for (String beanName : beanNames) {

    System.out.println(beanName);

}

}
```

What if multiple matching beans are found?

If multiple matching beans are found, Spring will throw an exception. We can resolve this issue by using `@Qualifier` annotation.

Another option is to use `@Primary` annotation. If we use `@Primary` annotation, Spring will use the bean that is marked with `@Primary` annotation if nothing else is specified.

Primary vs Qualifier

`@Primary` annotation is used to specify the default bean to be used when multiple beans are available. `@Qualifier` annotation is used to specify the bean to be used when multiple beans are available.

What is the difference between `@Component` and `@Bean`?

- `@Component` annotation is used to mark a class as a bean so that the component-scanning mechanism of Spring can pick it up and pull it into the application context. `@Component` annotation is used with classes that we have written.
- `@Bean` annotation is used to mark a method as a bean so that the bean definition is generated and managed by the Spring container. `@Bean` annotation is used with methods that we have written.
- `@Component` annotation is used with classes that we have written.
- `@Bean` annotation is used with methods that we have written.

Dependency Injection

Dependency Injection is a design pattern that allows us to remove the hard-coded dependencies and make our code loosely coupled. It is a process whereby objects define their dependencies, that is, the other objects they work with, only through constructor arguments, arguments to a factory method, or properties that are set on the object instance after it is constructed or returned from a factory method. The container then injects those dependencies when it creates the bean. This process is fundamentally the inverse (hence the name, Inversion of Control) of the bean itself controlling the instantiation or location of its dependencies by using direct construction of classes, or a mechanism such as the Service Locator pattern.

Constructor Injection

Constructor Injection using @Autowired

```
public class Employee {  
  
    private String name; private String email;  
  
    @Autowired public Employee(String name, String email) {  
  
        this.name = name; this.email = email;    }  
  
}
```

Constructor Injection using @Autowired and @Qualifier

```
public class Employee {  
  
    private String name; private String email;  
  
    @Autowired public Employee(@Qualifier("name") String name,  
    @Qualifier("email") String email) {  
  
        this.name = name; this.email = email;    }  
  
}
```

Constructor Injection using @Autowired and @Primary

```
public class Employee {  
  
    private String name; private String email;
```



```
@Autowired public Employee(@Primary String name, String email) {  
  
    this.name = name; this.email = email; }  
  
}
```

Setter Injection

```
public class BusinessService {  
  
    @Autowired  
  
    public void setDataService(DataService dataService) {  
  
        System.out.println("Setter injection");  
  
        this.dataService = dataService;  
  
    }  
  
}
```

Field Injection

```
public class BusinessService {  
  
    @Autowired  
  
    private DataService dataService;  
  
}
```

Injection Example

```
@Component  
  
class YourBusinessClass {  
  
    Dependency1 dependency1;  
  
    // @Autowired // Field injection is not recommended  
  
    Dependency2 dependency2;  
  
  
    // Constructor injection -> Autowired is not necessary, it automatically  
    // injects dependencies.
```

```
@Autowired

public YourBusinessClass(Dependency1 dependency1, Dependency2 dependency2) {

    System.out.println("Constructor injection");

    this.dependency1 = dependency1;

    this.dependency2 = dependency2;

}


@Override

public String toString() {

    return "YourBusinessClass{" +

        "dependency1=" + dependency1 +

        ", dependency2=" + dependency2 +

        '}';

}


// @Autowired // Setter injection

public void setDependency1(Dependency1 dependency1) {

    System.out.println("Setter injection");

    this.dependency1 = dependency1;

}


public void setDependency2(Dependency2 dependency2) {

    this.dependency2 = dependency2;

}

}
```

Inversion of Control (IoC)

Inversion of Control (IoC) is a design principle in which the control of objects or portions of a program is transferred to a container or framework. Inversion of Control is a principle in software engineering by which the control of objects or portions of a program is transferred to a container or framework. The framework is responsible for managing the life cycle and the flow of control of the application.

In regular programming, the control of objects is in the hands of the programmer. The programmer creates objects, wires them together, puts them into a configuration, and then the objects are ready to be used by the application. Inversion of Control reverses this process. The objects are created by a framework, and the framework wires them together and puts them into a configuration. The application then uses the objects from the framework.

Dependency Injection (DI)

Dependency Injection (DI) is a software design pattern that implements Inversion of Control for software applications. The basic idea behind DI is to provide the required dependencies to a class through external sources rather than creating them inside the class. This external source is called a container. The container is responsible for creating the dependencies and injecting them into the class.

Example:

```
@Service

public class BusinessCalculationService {

    // BusinessCalculationService depends on DataService.

    // BusinessCalculationService does not know which implementation of
    DataService is used. BusinessCalculationService needs to talk to DataService.

    // DataService is a dependency of BusinessCalculationService.

    private final DataService dataService;

    @Autowired // Constructor injection

    public BusinessCalculationService(DataService dataService) {
```

```
super(); // Not necessary

this.dataService = dataService; }

public int findMax() {

return Arrays.stream(dataService.retrieveData()).max().orElse(0);

}

}
```

Auto Wiring

Auto Wiring is a process in which Spring automatically wires beans together by inspecting the beans and matching them with each other.

Eager vs Lazy Initialization

Eager initialization is the process of initializing a bean as soon as the Spring container is created. Lazy initialization is the process of initializing a bean when it is requested for the first time.

The default Spring behaviour is Eager initialization. We can change the default behaviour to Lazy initialization by using `@Lazy` annotation.

Eager initialization is the recommended approach. Because errors in the configuration are discovered immediately at application startup. We should use Lazy initialization only when we are sure that the bean will not be used.

Bean Scopes

Bean Scopes are used to define the lifecycle of a bean. There are five different bean scopes in Spring:

- Singleton
- Prototype
- Request
- Session
- Global Session

Singleton Scope

Singleton scope is the default scope of a bean. It means that only one instance of the bean will be created and shared among all the clients.

It creates only one instance of the bean, and that instance is shared among all the clients.

Stateless beans. (Doesn't hold user information)

Prototype Scope

Prototype scope means that a new instance of the bean will be created every time a request is made for the bean.

It creates a new instance of the bean every time the bean is requested.

Stateful beans. (Holds user information)

To use Prototype scope, we need to use `@Scope` annotation with `@Component` annotation.

```
// Singleton: The reference in memory is the same. It creates only one instance. (Hash code is the same)
```

```
System.out.println(context.getBean(NormalClass.class));
```

```
System.out.println(context.getBean(NormalClass.class));
```

```
// Prototype: The reference in memory is different. It creates a new instance every time. (Hash code is different)
```

```
System.out.println(context.getBean(PrototypeClass.class));
```

```
System.out.println(context.getBean(PrototypeClass.class));
```

```
System.out.println(context.getBean(PrototypeClass.class));
```

Request Scope

Request scope means that a new instance of the bean will be created for each HTTP request.

Session Scope

Session scope means that a new instance of the bean will be created for each HTTP session.

Global Session Scope

Global Session scope means that a new instance of the bean will be created for each global HTTP session.

Java Singleton (GOF) vs Spring Singleton

- Java Singleton is a design pattern that restricts the instantiation of a class to one object. (One object instance per JVM)
- Spring Singleton is a bean scope that restricts the instantiation of a bean to one object. (One object instance per Spring IoC Container)

PostConstruct vs PreDestroy

- `@PostConstruct` annotation is used on a method that needs to be executed after dependency injection is done to perform any initialization. (ex: Fetching data from a database)
- `@PreDestroy` annotation is used on methods as a callback notification to signal that the instance is in the process of being removed by the container. (ex: Closing a database)

Jakarta Contexts and Dependency Injection (CDI)

Jakarta Contexts and Dependency Injection (CDI) is a standard for dependency injection and contextual lifecycle management for Java EE applications. It is a part of Jakarta EE. Spring Framework implements CDI specification.

- Named: `@Named` -> `@Named("name")` is used to specify the name of the bean. Alternatively, we can use `@Component` annotation.
- Inject: `@Inject` -> `@Inject` is used to inject a dependency. Alternatively, we can use `@Autowired` annotation.
- Qualifier: `@Qualifier` -> `@Qualifier("name")` is used to specify the name of the bean. Alternatively, we can use `@Qualifier` annotation.
- Scope: `@Scope` -> `@Scope("name")` is used to specify the scope of the bean. Alternatively, we can use `@Scope` annotation.

```
}
```

Spring XML Configuration

Spring XML Configuration is a way of configuring Spring beans using XML files. We can configure beans using XML files instead of using annotations.

Steps to configure Spring beans using XML files:

1. Create a Spring configuration file (ex: `beans.xml`) under `src/main/resources` folder.
2. Add the following XML code to the `beans.xml` file.

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="

        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd"> <!-- bean
        definitions here -->

    <bean id="name" class="java.lang.String">

        <constructor-arg value="Onur" />

    </bean>

    <context:component-scan
    base-package="com.onurcansever.learnspringframework.game" />
```

</beans>

- constructor-arg tag is used to inject a dependency to a constructor.
- property tag is used to inject a dependency to a setter method.
- context:component-scan tag is used to scan for components.

Spring Stereotype Annotations

- **@Component**: @Component is a generic stereotype for any Spring-managed component. It is a general-purpose annotation. It is a meta-annotation that serves as a specialization of @Component for different use cases.
- **@Service**: @Service is a specialization of @Component for service layer. Indicates that an annotated class has business logic.
- **@Controller**: @Controller is a specialization of @Component for presentation layer. Indicates that an annotated class is a "Controller" (e.g. a web controller, REST API).
- **@Repository**: @Repository is a specialization of @Component for persistence layer. Indicates that an annotated class is a "Repository" (e.g. a DAO). Used to retrieve and/or manipulate data from a database.

Why is Spring Ecosystem popular?

- **Loose Coupling**: Spring manages creating and wiring of beans and dependencies. It makes it maintainable and writing unit tests easily.
- **Reduced Boilerplate Code**: Spring provides a lot of annotations to reduce boilerplate code. No exception handling.
- **Architectural Flexibility**: Spring Modules and Projects.
- **Evolution with Time**: Microservices and Cloud. (Spring Boot, Spring Cloud etc.)

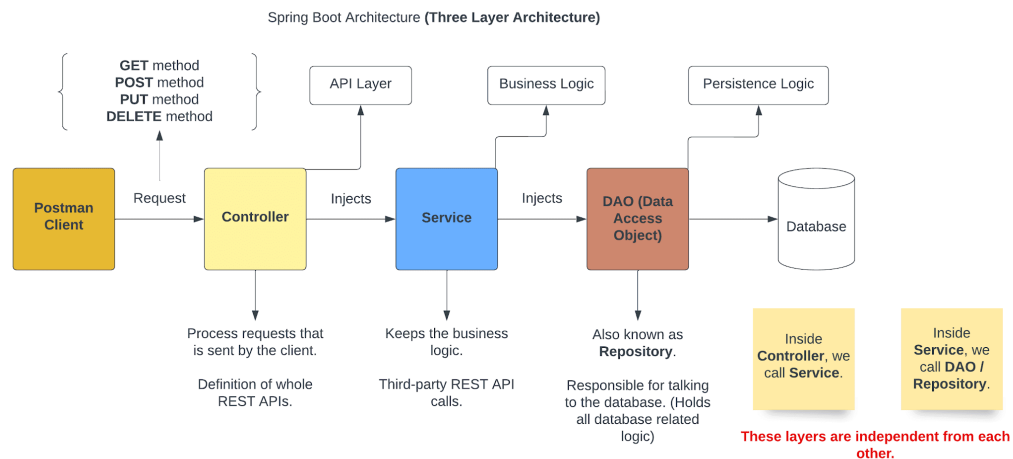
Spring Projects

- Application architectures evolve continuously:
- Web > Rest API > Microservices > Cloud > ...
- Spring evolves through Spring Projects:

- First Project: Spring Framework
- Spring Security: Secure your web application or REST API or microservice.
- Spring Data: Integrate the same way with different types of databases: NoSQL and Relational.
- Spring Integration: Address challenges with integration with other applications.
- Spring Boot: Popular framework to build microservices.
- Spring Cloud: Build cloud-native applications.

Three Layer Architecture

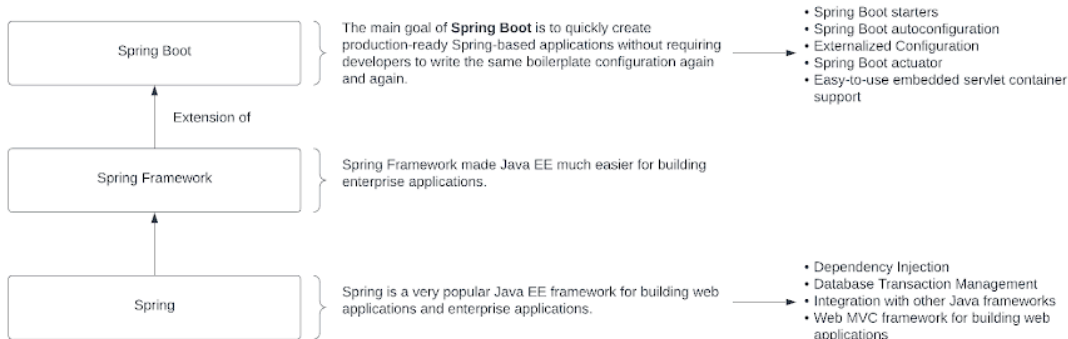
Flow of REST API HTTP Request Through Spring Boot Application



Spring Boot vs Spring MVC vs Spring

- Spring Framework: Dependency Injection and other Spring framework features. (Spring Modules and Spring Projects)
- Spring MVC (Spring Module): Simplify the development of web applications and RESTful web services. (@Controller, @RestController, @RequestMapping)
- Spring Boot (Spring Project): Quickly build production-ready applications. (Starter projects and autoconfiguration)
- Enables non functional requirements (NFRs):
- Embedded Servers
- Actuator
- Logging and Error Handling
- Profiles and Configuration Properties

What is Spring Boot?



What happens if we create Spring application without Spring Boot?

Spring based applications require lots of configuration.

- **Spring MVC:** Component Scan, Dispatcher Servlet, View Resolver, Web Jars (for delivering static content) among other things.
- **Hibernate / JPA:** Data Source, Entity Manager Factory / Session Factory
- **Transaction Manager**
- **Cache:** Cache Configuration
- **Message Queue:** Message Queue Configuration
- **NoSQL Database:** NoSQL Database Configuration

Spring Boot helps us to get rid of the boilerplate and configuration.

Spring Boot helps developers to focus on business logic.

Spring Boot Auto-Configuration

Attempts to automatically configure the Spring application based on the JAR dependencies that is provided in class path.

Spring Boot automatically configures. (JAR file)

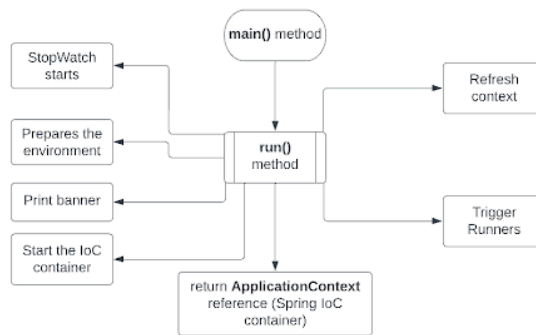
Externalized Configuration

Allows developer to work on different environments such as dev, prod, etc... Configuration is created with **application.properties**.

Spring Boot Actuator

Provides REST endpoints to monitor our application. (/health, /metrics, etc...)

Spring Boot App Execution Process



```

@SpringBootApplication
public class SpringbootDemoApplication {

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

`run()` method returns bootstraps the Spring Boot application. It runs all the auto configuration and executes them. It creates the Spring `IoC` container and returns it.

1. Spring Boot application execution will start from the `main()` method.
2. The `main()` method will internally call `SpringApplication.run()` method.
3. `SpringApplication.run()` method performs bootstrapping of the Spring Boot application.
4. Starts `StopWatch` to identify time taken to bootstrap the Spring Boot application.
5. Prepares the environment to run the Spring Boot application. (dev, prod, qa, uat)
6. Print banner (Spring Boot logo prints on the console)
7. Start the `IoC` container (`ApplicationContext`) based on the class path. (default, Web Servlet / Reactive)
8. Refresh context.
9. Triggers Runners (`ApplicationRunner` or `CommandLineRunner`) -> Execute logic only once only startup phase of the application.
10. Return `ApplicationContext` reference (Spring `IoC` container)

Types of Spring Boot Application

`spring-boot-starter` → If we have `spring-boot-starter` dependency in a classpath, then the Spring boot application comes under **default** category. We can use it for "utilities, standalone projects, desktop-based (GUI) projects".

`spring-boot-starter-web` → If we have `spring-boot-starter-web` dependency in a classpath, then the Spring boot application comes under **servlet** category. We can use it for "Spring MVC web applications, Spring MVC REST API applications".

If we have `spring-boot-starter` dependency in a classpath, then the Spring boot

Spring JDBC, H2 Database, Hibernate, JPA Notes

- H2 Database: In-memory database.
- Connection to H2 Database: `conn0:`
`url=jdbc:h2:mem:bdfa2bb8-4131-4dce-96c4-550927009ec5 user=SA`
- In `application.properties`, set `spring.h2.console.enabled=true`
- To access console, go to `http://localhost:8080/h2-console/`
- Copy and paste `jdbc:h2:mem:f30911dd-9715-4a83-9d4a-880d582be6a7` into JDBC URL. This is a dynamic URL which changes every restart. We can configure static URL.
- To configure static URL, add `spring.datasource.url=jdbc:h2:mem:testdb` in `application.properties`. This will create a database called `testdb` in memory.

If we want to use JDBC, JPA, Spring Data JPA, Hibernate etc. To do that, we need to create tables in the H2 database.

We need to create a file called `schema.sql` in `src/main/resources` folder. This file will contain the SQL statements to create the tables.

```
CREATE TABLE course
(
    id BIGINT NOT NULL,
    name VARCHAR(255) NOT NULL,
    author VARCHAR(255) NOT NULL,
    PRIMARY KEY (id)
);
```

Spring Data JPA

- Spring Data JPA is a framework that provides a simple, lightweight, and fast way to access the database.

- It makes JPA easy to use. It configures EntityManager.

```
// This is a Spring Data JPA Repository

// We need to create an interface and extend JpaRepository.

// We need to pass the entity class (Course) and the primary key type.

public interface CourseSpringDataJpaRepository extends JpaRepository<Course,
Long> {

    // We can add custom methods to this interface.

    // Spring Data JPA will implement these methods.

    // Spring Data JPA will create a proxy object for this interface.


    // Search by author.

    // Follow naming conventions. We are searching by author, so we need to
name the method findByAuthor.

    List<Course> findByAuthor(String author);

    List<Course> findByName(String name);

}
```

Hibernate vs JPA

- JPA is a specification. It is an API. (How to define entities, How to map attributes, Who manage the entities)
- Hibernate is an implementation of JPA.
- Using Hibernate will lock us to Hibernate. We cannot use other implementations of JPA. (Toplink, EclipseLink, OpenJPA)

Spring MVC

Spring MVC(Model-View-Controller) provides a convenient way to develop a java based web application.

It has a central servlet called as `DispatcherServlet` which is well known as front controller that intercepts all the requests, identify the appropriate handler i.e. controllers and render views to the client.

Spring MVC flow

In Spring Web MVC, `DispatcherServlet` class works as the front controller. It is responsible to manage the flow of the spring mvc application.

The `@Controller` annotation is used to mark the class as the controller in Spring 3.

The `@RequestMapping` annotation is used to map the request url. It is applied on the method.

Spring Web Annotations

@RequestMapping

it can be configured using:

- path, or its aliases, name, and value: which URL the method is mapped to
- method: compatible HTTP methods
- params: filters requests based on presence, absence, or value of HTTP parameters
- headers: filters requests based on presence, absence, or value of HTTP headers
- consumes: which media types the method can consume in the HTTP request body
- produces: which media types the method can produce in the HTTP response body

Example:

```
@Controller

class VehicleController {

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

    String home() {
```

```

        return "home";
    }
}

```

this configuration has the same effect :

```

@Controller

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

class VehicleController {

    @RequestMapping("/home")

    String home() {

        return "home";
    }

}

```

Moreover, `@GetMapping`, `@PostMapping`, `@PutMapping`, `@DeleteMapping`, and `@PatchMapping` are different variants of `@RequestMapping` with the HTTP method already set to GET, POST, PUT, DELETE, and PATCH respectively.

These are available since Spring 4.3 release.

@RequestBody

maps the body of the HTTP request to an object. The deserialization is automatic and depends on the content type of the request.

```

@PostMapping("/save")

void saveVehicle(@RequestBody Vehicle vehicle) {

    // ...

}

```

@PathVariable

This annotation indicates that a method argument is bound to a URI template variable. We can specify the URI template with the `@RequestMapping` annotation and bind a method argument to one of the template parts with `@PathVariable`.

We can achieve this with the name or its alias, the value argument:

```
@RequestMapping("/{id}")

Vehicle getVehicle(@PathVariable("id") long id) {

    // ...

}
```

If the name of the part in the template matches the name of the method argument, we don't have to specify it in the annotation:

```
@RequestMapping("/{id}")

Vehicle getVehicle(@PathVariable long id) {

    // ...

}
```

Moreover, we can mark a path variable optional by setting the argument required to `false`:

```
@RequestMapping("/{id}")

Vehicle getVehicle(@PathVariable(required = false) long id) {

    // ...

}
```

@RequestParam

We use `@RequestParam` for accessing HTTP request parameters:

```
@RequestMapping

Vehicle getVehicleByParam(@RequestParam("id") long id) {

    // ...

}
```

It has the same configuration options as the `@PathVariable` annotation.

In addition to those settings, with `@RequestParam` we can specify an injected value when Spring finds no or empty value in the request. To achieve this, we have to set the `defaultValue` argument.

Providing a default value implicitly sets `required` to `false`:

```
@RequestMapping("/buy")

Car buyCar(@RequestParam(defaultValue = "5") int seatCount) {

    // ...

}
```

Response Handling Annotations

@ResponseBody

If we mark a request handler method with `@ResponseBody`, Spring treats the result of the method as the response itself:

```
@ResponseBody

@RequestMapping("/hello")

String hello() {

    return "Hello World!";

}
```

@ExceptionHandler

With this annotation, we can declare a custom error handler method. Spring calls this method when a request handler method throws any of the specified exceptions.

The caught exception can be passed to the method as an argument:

```
@ExceptionHandler(IllegalArgumentException.class)

void onIllegalArgumentException(IllegalArgumentException exception) {

    // ...

}
```



```
}
```

@ResponseStatus

We can specify the desired HTTP status of the response if we annotate a request handler method with this annotation. We can declare the status code with the code argument, or its alias, the value argument.

Also, we can provide a reason using the reason argument.

We also can use it along with @ExceptionHandler:

```
@ExceptionHandler(IllegalArgumentException.class)

@ResponseStatus(HttpStatus.BAD_REQUEST)

void onIllegalArgumentException(IllegalArgumentException exception) {

    // ...

}
```

Other Web Annotations

@Controller

We can define a Spring MVC controller with @Controller. @Controller is a class level annotation which tells the Spring Framework that this class serves as a controller in Spring MVC:

```
@Controller

public class VehicleController {

    // ...

}
```

@RestController

The @RestController combines @Controller and @ResponseBody.

```
@Controller
```

```

@ResponseBody

class VehicleRestController {

    // ...

}

```

is same as :

```

@RestController

class VehicleRestController {

    // ...

}

```

@ModelAttribute

With this annotation we can access elements that are already in the model of an MVC `@Controller`, by providing the model key:

```

@PostMapping("/assemble")

void assembleVehicle(@ModelAttribute("vehicle") Vehicle vehicleInModel) {

    // ...

}

```

Like with `@PathVariable` and `@RequestParam`, we don't have to specify the model key if the argument has the same name:

```

@PostMapping("/assemble")

void assembleVehicle(@ModelAttribute Vehicle vehicle) {

    // ...

}

```

Besides, `@ModelAttribute` has another use: if we annotate a method with it, Spring will automatically add the method's return value to the model:

```

@ModelAttribute("vehicle")

```

```
Vehicle getVehicle() {  
  
    // ...  
  
}
```

Like before, we don't have to specify the model key, Spring uses the method's name by default:

```
@ModelAttribute  
  
Vehicle vehicle() {  
  
    // ...  
  
}
```

Project Lombok is a java library that automatically plugs into your editor and build tools, spicing up your java. Never write another getter or equals method again, with one annotation your class has a fully featured builder, automate your logging variables,