Spring Boot

# Setting up Spring Boot Project

* Go to ***start.spring.io***
* Select the below:
  + **Project:** Maven
  + **Language:** Java
  + **Spring Boot Version:** 2.7.11
  + Fill up Project Metadata
  + **Packaging:** Jar
  + **Java Version:** 17 (Select the installed java version)
* Add the **dependencies** below in the project:
  + Spring Web
  + Spring Boot Actuator
* Run the project on intellij and visit ***locahost:port/actuator*** to view the site.
* Install JSON formatter extension to view the result beautifully.

# MAVEN setup

* Download Maven Binary files.
* Copy to Program Files.
* Setup environment variable ***M2\_HOME*** (Just the path to main directory of Maven).
* Setup Bin path to the environment path variable.
* Check the installation using command ***mvn***.

# Postgres setup

* **Default user:** postgres
* **Password:** abc123
* **Port:** 5432
* To check if the postgres is installed use command: ***psql -U postgres***
* Install DBEAVER for GUI access.

# Port Checking

If your port is being used by another application then the command to view which port are currently in use in windows: ***netstat -ano***

# Changing the port in Spring-Boot Project

* Go to ***src/main/resources.***
* Open ***application.properties***
* Add the line: ***server.port = 8081*** (Or any port that is not used).
* Save it and run : ***localhost:newPort***.

# HTTP Request and Response

HTTP (Hypertext Transfer Protocol) is a protocol used for communication between web clients (such as web browsers) and web servers. When a client sends an HTTP request to a server, the server responds with an HTTP response.

**HTTP requests and responses have several components:**

## HTTP Request:

* **Request line:** Includes the HTTP method (GET, POST, etc.) and the URL path of the resource the client wants to access.
* **Request headers:** Provide additional information about the request, such as the type of data that the client can accept in response, authentication credentials, and more.
* **Request body:** Contains optional data that the client can send to the server, such as form data or JSON data.

## HTTP Response:

* **Status line:** Includes the HTTP status code, which indicates whether the request was successful or not, and a short message describing the status code.
* **Response headers:** Provide additional information about the response, such as the content type of the data that the server is sending, the date and time the response was generated, and more.
* **Response body:** Contains the data that the server is sending to the client in response to the request, such as an HTML document, a JSON object, or an image file.

When a client sends an HTTP request to a server, the server processes the request and generates an HTTP response. The response is then sent back to the client, which can then parse the response and display the data or content to the user.

HTTP requests and responses are at the heart of the web and are used for many different types of interactions between clients and servers, including retrieving web pages, submitting forms, and interacting with web APIs.

# HTTP Methods

HTTP (Hypertext Transfer Protocol) methods define the actions that clients (such as web browsers) can perform on resources (such as web pages) located on web servers. There are several HTTP methods that are commonly used, each of which has a specific purpose and behavior:

## Get Method

The GET method is one of the HTTP (Hypertext Transfer Protocol) **methods used by web clients** (such as web browsers) **to request resources** (such as web pages) **from web servers.** *When a client sends a GET request to a server, the server retrieves the requested resource and sends it back to the client in the response*.

The **basic structure of a GET request** includes a *request line, request headers, and an optional request body*. The ***request line includes*** the ***HTTP method*** (in this case, GET) and the ***URL path of the resource that the client wants to retrieve***. The ***request headers*** provide additional information about the request, such as the user agent making the request and the type of data that the client can accept in response. The ***request body is not used in a GET request***.

When the server receives a GET request, it processes the request by locating the requested resource and sending it back to the client in the response. The ***response typically includes*** a ***status line, response headers***, and a ***response body.*** The status line includes the HTTP status code (such as 200 OK if the request was successful) and a short message describing the status code. The response headers provide additional information about the response, such as the content type of the data that the server is sending. The response body contains the actual data (such as an HTML document or an image file) that the client requested.

The ***GET method is commonly used to request web pages, images, and other static content from web servers***. Since GET requests only retrieve data from the server and do not modify server-side data, they are ***considered "safe" and "idempotent"***, meaning that multiple identical requests will produce the same response without changing the state of the server.

## POST Method

The POST method is one of the HTTP (Hypertext Transfer Protocol) methods used by web clients (such as web browsers) **to submit data to web servers**. When a client sends a POST request to a server, the request includes a body that contains the data that the client wants to send to the server.

The ***basic structure of a POST request*** includes a ***request line, request headers***, and ***a request body***. The ***request line includes the HTTP method*** (in this case, POST) and ***the URL path of the resource that the client wants to submit data to***. The ***request headers*** provide additional information about the request, such as the user agent making the request and the type of data that the client is sending in the request body. The ***request body contains the actual data that the client is submitting to the server***.

***When the server receives a POST request***, it processes the request by extracting the data from the request body and using it to create or update a resource on the server. The server then sends a response back to the client, which typically includes a status line, response headers, and a response body. The status line includes the HTTP status code (such as 200 OK if the request was successful) and a short message describing the status code. The response headers provide additional information about the response, such as the content type of the data that the server is sending. The response body may contain data that the server wants to send back to the client as a result of the POST request.

***The POST method is commonly used to submit form data, upload files, and perform other actions that modify server-side data***. Since POST requests can modify server-side data, they are ***not considered "safe" or "idempotent" like GET requests***. This means that multiple identical POST requests may produce different results and may modify the state of the server.

## PUT Method

The PUT method is one of the HTTP (Hypertext Transfer Protocol) methods used by web clients (such as web browsers) ***to update or replace an existing resource on a web server***. When a client sends a PUT request to a server, the request includes a body that contains the updated representation of the resource.

The ***basic structure*** of a PUT request includes ***a request line, request headers, and a request body***. The request line includes the HTTP method (in this case, PUT) and the URL path of the resource that the client wants to update or replace. The request headers provide additional information about the request, such as the user agent making the request and the type of data that the client is sending in the request body. The request body contains the updated representation of the resource that the client wants to store on the server.

When the server receives a PUT request, it processes the request by storing the updated representation of the resource sent in the request body at the specified URL path. If the resource does not exist, the server may create a new resource at that URL path. The server then sends a response back to the client, which typically includes a status line, response headers, and a response body. The status line includes the HTTP status code (such as 200 OK if the request was successful) and a short message describing the status code. The response headers provide additional information about the response, such as the content type of the data that the server is sending. The response body may contain data that the server wants to send back to the client as a result of the PUT request.

***The PUT method is commonly used to update or replace resources, such as documents or media files, on web servers***. Since PUT requests can modify server-side data, ***they are not considered "safe" like GET requests***. However, PUT requests ***are considered "idempotent",*** meaning that multiple identical PUT requests will produce the same result without changing the state of the server beyond the initial PUT request.

## DELETE Method

The DELETE method is one of the HTTP (Hypertext Transfer Protocol) methods used by web clients (such as web browsers) ***to request the removal of a resource on a web server***. When a client sends a DELETE request to a server, the request includes the URL path of the resource that the client wants to delete.

The ***basic structure*** of a DELETE request includes a ***request line, request headers, and no request body***. The ***request line*** includes the ***HTTP method*** (in this case, DELETE) and the ***URL path of the resource that the client wants to delete***. The ***request headers*** provide additional information about the request, such as the user agent making the request and the type of data that the client expects in the response.

***When the server receives a DELETE request, it processes the request by removing the specified resource from the server***. The server then sends a ***response*** back to the client, which typically includes a ***status line, response headers, and a response body***. The ***status line*** includes the HTTP status code (such as 200 OK if the request was successful) and a short message describing the status code. The ***response headers*** provide additional information about the response, s***uch as the content type of the data that the server is sending***. The ***response body*** ***may contain data that the server wants to send back to the client as a result of the DELETE request***.

***The DELETE method is commonly used to remove resources, such as documents or media files, from web servers.*** Since DELETE requests can modify server-side data, they are ***not considered "safe" or "idempotent" like GET requests***. This means that multiple identical DELETE requests may produce different results and may modify the state of the server.

## PATCH Method

The PATCH method is one of the HTTP (Hypertext Transfer Protocol) methods used by web clients (such as web browsers) ***to update or modify a portion of an existing resource on a web server***. ***When a client sends a PATCH request to a server, the request includes a body that contains a partial update to the resource***.

The ***basic structure*** of a PATCH request includes a ***request line, request headers, and a request body***. The ***request line*** includes the HTTP method (in this case, PATCH) and the URL path of the resource that the client wants to update. The ***request headers*** provide additional information about the request, such as the user agent making the request and the type of data that the client is sending in the request body. The ***request body*** contains the partial update to the resource that the client wants to apply.

***When the server receives a PATCH request, it processes the request by applying the partial update to the specified resource***. The server may ***use various strategies*** to apply the update, such as replacing or adding specific fields or attributes within the resource. The server then sends a ***response back to the client***, which typically includes a ***status line, response headers, and a response body***. The ***status line*** includes the HTTP status code (such as 200 OK if the request was successful) and a short message describing the status code. The ***response headers*** provide additional information about the response, such as the content type of the data that the server is sending. The ***response body*** may contain data that the server wants to send back to the client as a result of the PATCH request.

***The PATCH method is commonly used to modify specific portions of resources, such as documents or data records, on web servers***. Since PATCH requests can modify server-side data, they are ***not considered "safe" or "idempotent" like GET requests***. However, PATCH requests are considered "idempotent" if the same request is applied multiple times to the same resource and produces the same result each time.

## HEAD Method

The HEAD method is one of the HTTP (Hypertext Transfer Protocol) methods used by web clients (such as web browsers) to ***retrieve only the headers of a resource on a web server without actually retrieving the resource itself***. When a client sends a HEAD request to a server, the request includes the URL path of the resource that the client wants to retrieve.

The ***basic structure*** of a HEAD request includes a ***request line, request headers, and no request body***. The ***request line*** includes the HTTP method (in this case, HEAD) and the URL path of the resource that the client wants to retrieve. The ***request headers*** provide additional information about the request, such as the user agent making the request and the type of data that the client expects in the response.

When the server receives a HEAD request, ***it processes the request by retrieving only the headers of the specified resource, rather than the entire resource itself***. The server then sends a ***response back to the client***, which typically includes a ***status line, response headers, and no response body***. The ***status line*** includes the HTTP status code (such as 200 OK if the request was successful) and a short message describing the status code. The ***response headers*** provide additional information about the resource, such as its size, last modified date, and content type.

***The HEAD method is commonly used to retrieve metadata about resources, such as their size or last modified date, without actually retrieving the resource itself.*** This can be useful in situations where a client wants to check whether a resource has been updated since it was last accessed, without actually downloading the entire resource again. Since HEAD requests do not retrieve the entire resource, they are considered ***"safe" and "idempotent" like GET requests***. This means that multiple identical HEAD requests will produce the same result and will not modify the state of the server.

# HTTP Status Codes

HTTP status codes are a three-digit code that is returned by a server in response to a client's request to indicate the status of the requested resource. These codes are an important part of the HTTP protocol and provide useful information to clients, developers, and network administrators.

There are five main classes of status codes, each represented by the first digit of the code:

* **1xx (Informational**): This class of status codes indicates that the request was received and understood but requires further action to proceed.
* **2xx (Successful):** This class of status codes indicates that the request was successfully received, understood, and processed by the server.
* **3xx (Redirection):** This class of status codes indicates that further action is needed by the client to complete the request.
* **4xx (Client Error):** This class of status codes indicates that the request was malformed or invalid and cannot be fulfilled by the server.
* **5xx (Server Error):** This class of status codes indicates that an error occurred on the server and the request cannot be fulfilled.

Some common HTTP status codes include:

* **200 OK:** The request was successful and the server returned the requested data.
* **201 Created:** The server has successfully created a new resource as a result of the request.
* **400 Bad Request:** The request was malformed or invalid and the server could not understand it.
* **401 Unauthorized:** The client is not authorized to access the requested resource.
* **403 Forbidden:** The client is authorized to access the requested resource, but the server refuses to fulfill the request.
* **404 Not Found:** The requested resource could not be found on the server.
* **500 Internal Server Error:** An error occurred on the server and the request could not be fulfilled.

In a Spring Boot application, the server will return an appropriate HTTP status code based on the outcome of a request, and the client can use this code to determine the success or failure of the request.

# HTTP request handling in Spring Boot

When a user enters a path in browser or clicks some anchor in webpage, Spring boot does something that is called handling HTTP request.

Spring Boot then processes the request and generates an HTTP response which is sent back to the browser. The response typically includes the requested data or content which can be HTML, JSON, images or any other type of data.

***In Spring Boot***, ***handling HTTP requests is typically done using controllers which are classes that define methods to handle specific URL paths.*** The methods in the controller are annotated with specific annotations as ***‘@GetMapping’***, ***‘@PostMapping’***, etc. to indicate the HTTP methods (GET, POST etc.) that the method should handle.

# MVC Architecture of Spring Boot

The ***Model-View-Controller (MVC)*** architectural pattern is a widely-used design pattern in web development. Spring Boot is a popular Java framework that implements the MVC pattern to build web applications.

In the MVC architecture of Spring Boot, the application is divided into three main components: the Model, the View, and the Controller.

## Model

The Model ***represents the application's data and business logic***. In Spring Boot, the Model is typically ***implemented as a set of Java classes that encapsulate the application's data and business logic***. The ***Model classes can interact with a database, an API, or any other data source***. The Model is responsible for providing data to the Controller and for storing data that has been modified by the Controller.

## View

The View is ***responsible for rendering the user interface***. In Spring Boot, the View is typically ***implemented as a set of HTML templates or JSP pages***. The View receives data from the Controller and generates HTML output that can be displayed in the user's browser.

## Controller

The Controller is ***responsible for processing incoming HTTP requests and generating responses***. In Spring Boot, the Controller is typically ***implemented as a set of Java classes that receive HTTP requests, invoke business logic in the Model, and return data to the View***. The Controller is ***responsible for controlling the flow of the application and for handling user input***.

## Flow of data in MVC architecture

The flow of data in the MVC architecture of Spring Boot can be summarized as follows:

1. The user sends an HTTP request to the Controller.
2. The Controller receives the request and extracts any data that was sent with the request.
3. The Controller invokes business logic in the Model to process the data and generate a response.
4. The Controller returns data to the View.
5. The View generates HTML output based on the data it received from the Controller.
6. The View sends the HTML output to the user's browser.

# Controller

A controller is a component that ***handles incoming HTTP requests and generates responses to the client***. The ***controller is responsible for processing user input, invoking business logic, and returning a response to the user***.

In the Model-View-Controller (MVC) architectural pattern, the controller is one of the three core components, alongside the model and the view. The model represents the application's data and business logic, while the view is responsible for rendering the user interface. The controller acts as an intermediary between the model and the view, processing user input, updating the model, and generating an appropriate view to present to the user.

## Controller in Spring Boot

A Controller in Spring Boot is typically implemented as a Java class and is annotated with the ***@Controller*** or ***@RestController*** annotation. The ***@Controller*** annotation indicates that the class is responsible for handling HTTP requests and returning responses, while the ***@RestController*** annotation is a specialized version of ***@Controller*** that is used to create RESTful web services.

Controllers typically define methods that are mapped to specific HTTP requests. These methods are annotated with one or more mapping annotations, such as @RequestMapping, ***@GetMapping, @PostMapping, @PutMapping, @DeleteMapping,*** or ***@PatchMapping***. These annotations indicate which HTTP method and URL path the method is mapped to.

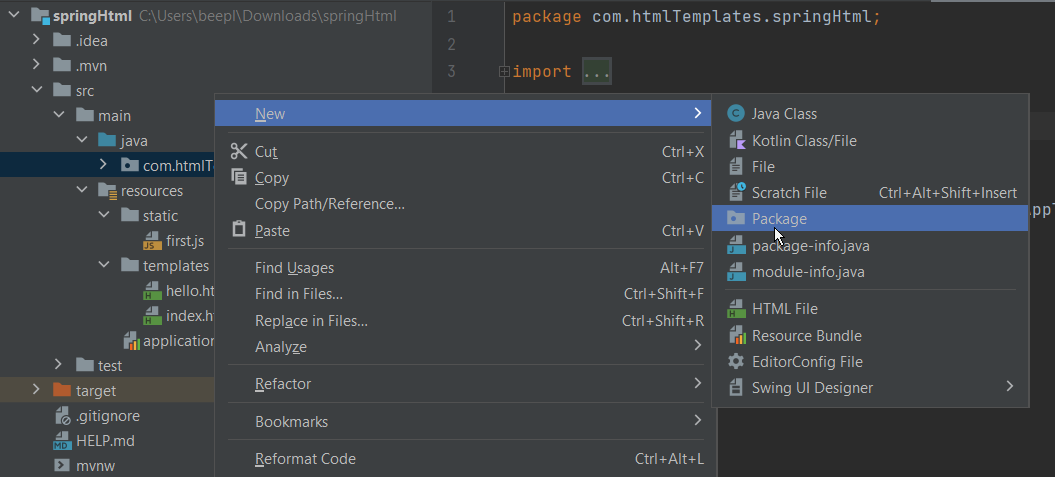
Inside the controller method, you can access request and response objects through method parameters. You can also use annotations such as ***@RequestParam***, ***@PathVariable***, and ***@RequestBody*** to extract parameters from the request, including URL parameters, form data, and JSON payloads.

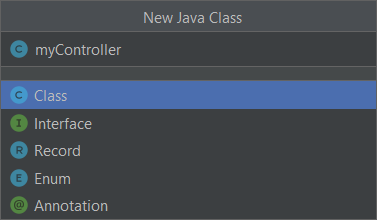
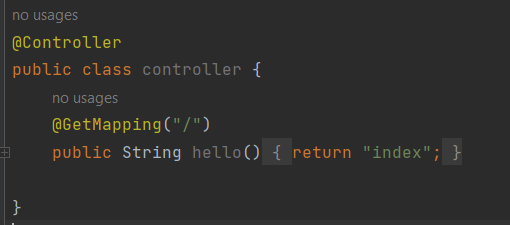
The method in the controller can return a variety of types, including a view name, a ModelAndView object, a ResponseEntity object, or a custom object that can be serialized to JSON. The returned object will be used to generate the response to the client.

## Creating a controller in Spring Boot

Below is a summarized step for creating a controller in Spring Boot Project.

1. Create a package called ***‘controller’*** in your project’s ***main> java> com.example***



1. Inside the ***‘controller’*** package create a new Java Class. Note that the name of Java Class must end with controller like ***HomePageController***, ***UserController***.
2. Give the class an annotation of ***‘@Controller’*** or ***‘@RestController’*** based on requirement.

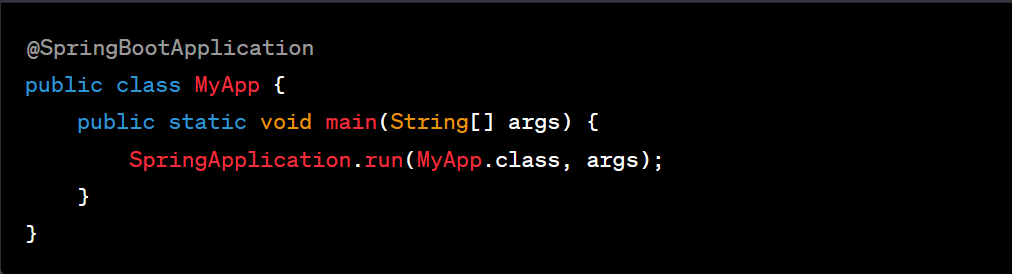
# Annotations in Spring Boot

Annotations in Spring Boot are used to ***provide additional metadata to your code, allowing the Spring framework to automatically configure and manage your application***. Annotations ***can be used to*** ***mark classes, methods, or fields with special meanings that are interpreted by Spring Boot at runtime***.

Some of the common annotations used in Spring Boot are:

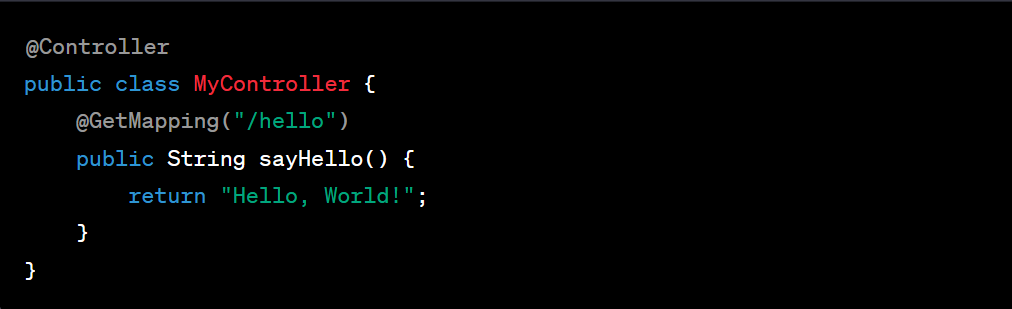
## @SpringBootApplication

This annotation is used to indicate the main class of a Spring Boot application. It is a combination of three other annotations: **@Configuration**, **@EnableAutoConfiguration**, and **@ComponentScan**. The **@Configuration** annotation indicates that the class contains configuration information, the **@EnableAutoConfiguration** annotation enables Spring Boot's auto-configuration feature, and the **@ComponentScan** annotation tells Spring Boot to scan the package for components such as controllers, services, and repositories.



## @Controller

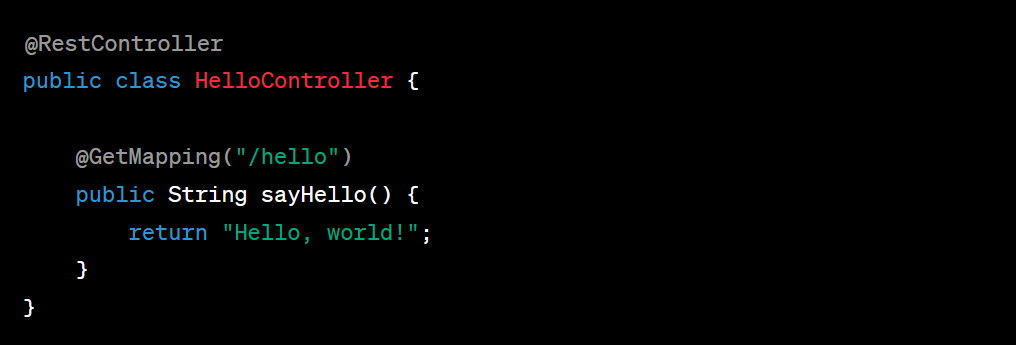
This annotation is used to mark a class as a controller. Controllers handle incoming HTTP requests and generate responses. This can be used to render static HTML pages.



## @RestController

In Spring Boot, the **@RestController** annotation is used to mark a class as a controller that handles HTTP requests and returns responses in a RESTful manner. It combines the functionality of **@Controller** and **@ResponseBody** annotations, making it easier to write RESTful web services.

It means that the return value of the method will be automatically converted to JSON or XML and sent back to the client.



In this example, we have created a **HelloController** class and annotated it with **@RestController**. This indicates that the class is a REST controller that handles incoming HTTP requests.

We have also defined a single method, **sayHello()**, that handles a GET request at the URL "***/hello***" and returns the string "Hello, world!" as the response.

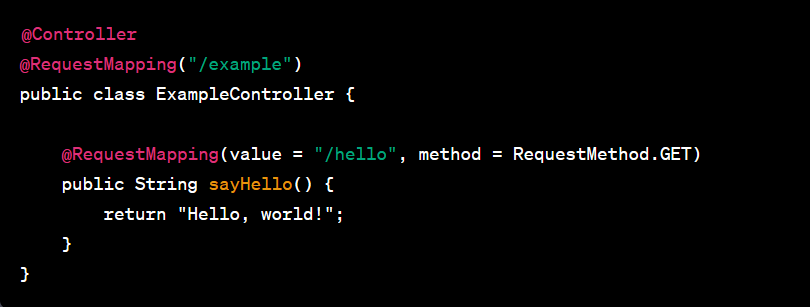
### Example done in class



## @RequestMapping

This annotation is used to map incoming HTTP requests to a specific method in a controller. You can specify the URL path, HTTP method, and other request parameters using this annotation.

**@RequestMapping** is an annotation in Spring Boot used to map a URL request to a particular method or a controller. It is used to handle HTTP requests based on the HTTP method, URL pattern, and other criteria.



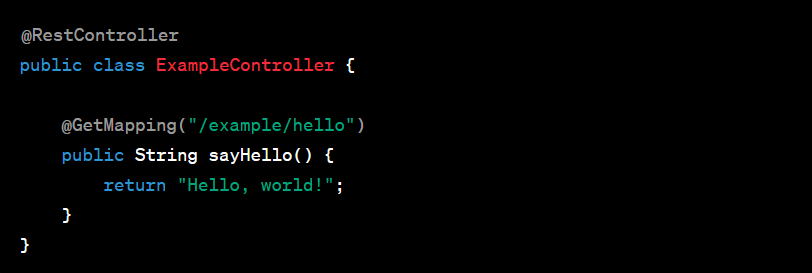
In this example, we have created an **ExampleController** class and annotated it with **@Controller**. We have also used **@RequestMapping** to specify that all requests handled by this controller should start with "***/example***".

We have then used **@RequestMapping** again to specify the URL mapping for the **sayHello()** method. In this case, the method handles a GET request at the URL ***"/example/hello***". We have used **RequestMethod.GET** to specify that this method should only handle GET requests.

We could also use other HTTP methods like **RequestMethod.POST**, **RequestMethod.PUT**, **RequestMethod.DELETE**, or **RequestMethod.PATCH** to handle different HTTP methods.

## @GetMapping

**@GetMapping** is a shortcut annotation in Spring Boot that is used to map HTTP GET requests to specific controller methods. It is essentially a combination of **@RequestMapping(method = RequestMethod.GET)** and **@ResponseBody**.

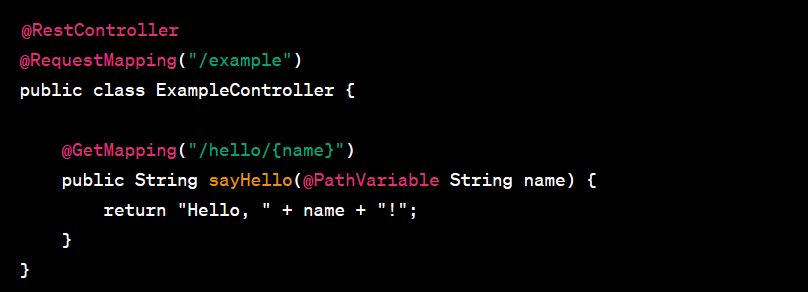


In this example, we have created an **ExampleController** class and annotated it with **@RestController**. This indicates that the class is a REST controller that handles incoming HTTP requests.

We have then used **@GetMapping** to specify that the **sayHello()** method should handle HTTP GET requests at the URL ***"/example/hello***". The method returns the string "Hello, world!" as the response.

## @PathVariable

**@PathVariable** is a Spring Boot annotation that is used to map dynamic values from the URI of a request to a method parameter in a controller.



In this example, we have created an **ExampleController** class and annotated it with **@RestController** and **@RequestMapping("/example")**.

We have then used **@GetMapping** to specify that the **sayHello()** method should handle HTTP GET requests at the URL ***"/example/hello/{name}".*** The **{name}** part of the URL is a dynamic value that will be replaced with an actual value at runtime.

We have used **@PathVariable** to specify that the **name** parameter of the **sayHello()** method should be mapped to the dynamic value in the URL. The value of **name** will be extracted from the URL and passed to the method as a parameter.

For example, if we make a GET request to ***"/example/hello/John***", the **sayHello()** method will be called with the parameter **name** set to "John", and it will return the string "Hello, John!".

### Arguments

The **@PathVariable** annotation in Spring Boot takes the following arguments:

1. **value** or **name**: The name of the path variable. If the name of the path variable in the URL and the method parameter name are the same, this argument is optional.
2. **required**: A boolean value indicating whether the path variable is required or not. If set to **true** (default), Spring Boot will throw an exception if the path variable is missing from the request URL.
3. **defaultValue**: The default value to be used for the path variable if it is not present in the request URL. This argument is optional.

## @RequestParam

**@RequestParam** is a Spring Boot annotation that is used to extract query parameters or form data from an HTTP request and pass them as arguments to a method in a controller.



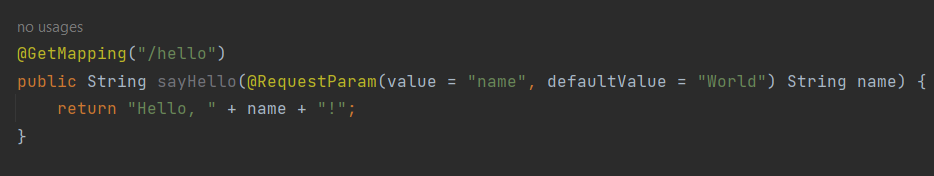
In this example, we have created an **ExampleController** class and annotated it with **@RestController** and **@RequestMapping("/example")**.

We have then used **@GetMapping** to specify that the **sayHello()** method should handle HTTP GET requests at the URL ***"/example/hello***".

We have used **@RequestParam** to specify that the **name** parameter of the **sayHello()** method should be extracted from the query parameters of the HTTP request. The value of the **name** parameter in the query string will be passed to the method as a parameter.

For example, if we make a GET request to ***"/example/hello?name=John"***, the **sayHello()** method will be called with the parameter **name** set to "John", and it will return the string "Hello, John!". If the **name** parameter is not present in the query string, a **MissingServletRequestParameterException** will be thrown. However, we can specify the **required** attribute of **@RequestParam** to be **false** to make the parameter optional.

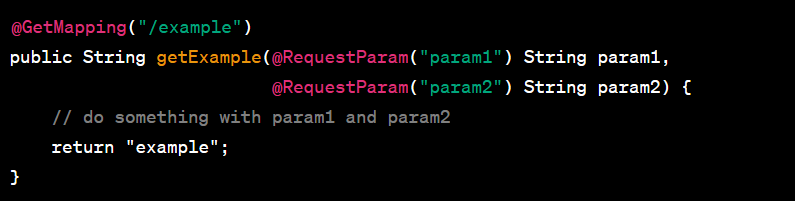
We can also provide default values for optional parameters using the **defaultValue** attribute of **@RequestParam**. For example:



In this example, if the **name** parameter is not present in the query string, the value "World" will be used as the default value.

### Sending Multiple Data with @RequestParam

We can specify multiple query parameters by specifying multiple @RequestParam annotations or by separating them with an ampersand (&) in the URL.



In the above example, two query parameters are specified with the @RequestParam annotation, "param1" and "param2". The values for these parameters can be passed in the URL like this:

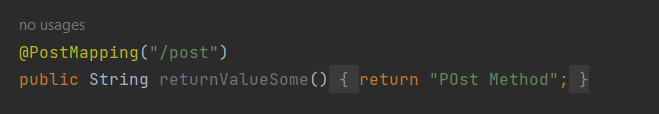


In this case, the value of "param1" will be "value1" and the value of "param2" will be "value2". You can add more @RequestParam annotations to handle additional query parameters.

## @PostMapping

**@PostMapping** is an annotation in Spring Boot that is used to map HTTP POST requests to specific handler methods in a Spring controller.

When you annotate a method with **@PostMapping**, you are telling Spring Boot that this method should handle HTTP POST requests to a specific URL mapping.



In this example, we have defined a **MyController** class that is annotated with **@RestController** and **@RequestMapping("/test")**.

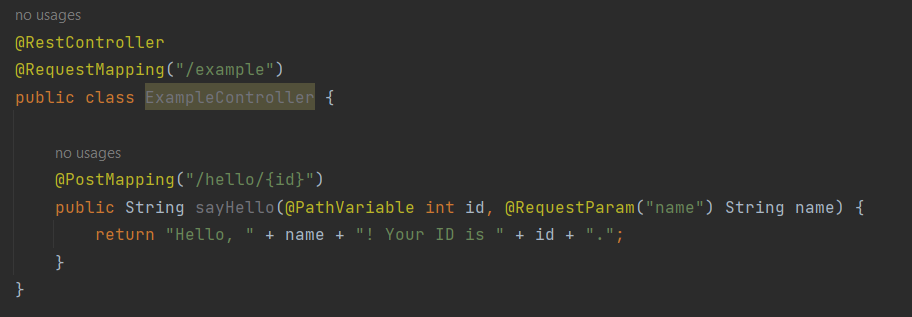
We have then defined a method **returnValueSome()** that is annotated with **@PostMapping("/post")**. This means that this method will handle HTTP POST requests to the ***"/test/post***" URL mapping.

The method takes no parameter. It simply returns the string “Post Method”.

To test this endpoint, you can make a POST request to [**http://localhost:8080/test/post**](http://localhost:8080/test/post).

## @PostMapping with @PathVariable and @RequestParam

**@RequestParam** can be used to extract query parameters or form data from an HTTP POST request, while **@PathVariable** can be used to extract dynamic values from the URL path of the request.



In this example, we have created an **ExampleController** class and annotated it with **@RestController** and **@RequestMapping("/example")**.

We have then used **@PostMapping** to specify that the **sayHello()** method should handle HTTP POST requests at the URL ***"/example/hello/{id}".*** The **{id}** part of the URL is a dynamic value that will be replaced with an actual value at runtime.

We have used **@PathVariable** to specify that the **id** parameter of the **sayHello()** method should be mapped to the dynamic value in the URL.

We have used **@RequestParam** to specify that the **name** parameter of the **sayHello()** method should be extracted from the form data of the HTTP POST request. The value of the **name** parameter in the form data will be passed to the method as a parameter.

For example, if we make a POST request to ***"/example/hello/123***" with a form data parameter **name** set to "John", the **sayHello()** method will be called with the parameters **id** set to 123 and **name** set to "John", and it will return the string "Hello, John! Your ID is 123.".

## ResponseEntity<>

**ResponseEntity** is a class in Spring Boot that represents an HTTP response, including the response body, headers, and status code. It allows you to customize the HTTP response that your Spring Boot application sends back to the client.

With **ResponseEntity**, you can set the HTTP status code, response headers, and response body. This provides more flexibility than simply returning the response body as an object, as you can customize other aspects of the response.

For example, you can use **ResponseEntity** to return a 404 Not Found response when a resource is not found, or a 201 Created response when a new resource is successfully created. You can also add custom headers to the response, such as content-type or cache-control headers.

### Syntax:

***ResponseEntity<T> responseEntity = new ResponseEntity<>(body, headers, status);***

Here,

* **T** is the type of the response body, which can be any object.
* **body** is the response body object that you want to send to the client.
* **headers** is an instance of **HttpHeaders** that contains any custom headers that you want to include in the response.
* **status** is an instance of **HttpStatus** that represents the HTTP status code that you want to return in the response.

You can also create a **ResponseEntity** without headers or status code like this:

***ResponseEntity<T> responseEntity = new ResponseEntity<>(body, HttpStatus.OK);***

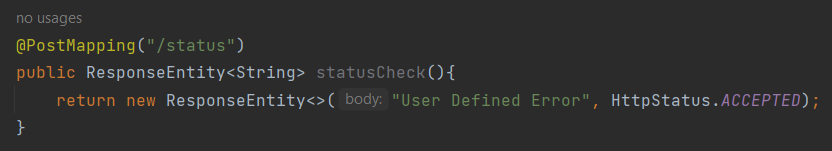
In this case, the **ResponseEntity** will have a default HTTP status code of 200 OK.

Note that you can also use static methods of **ResponseEntity** class to create instances of **ResponseEntity** without creating a new object, for example:

***return ResponseEntity.ok().body("Hello, World!");***

This creates a new **ResponseEntity** with a body of "Hello, World!" and an HTTP status code of 200 OK.

### Example



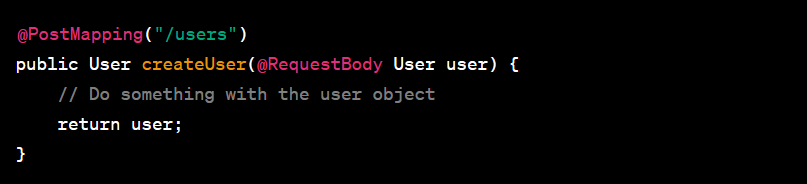
## @RequestBody

**@RequestBody** is an annotation in Spring Boot that ***binds the HTTP request body to a Java object***. When a client sends an HTTP POST or PUT request with a ***JSON or XML payload***, the **@RequestBody** annotation is used to extract the payload and convert it into a Java object.

The data sent in the **@RequestBody** can be in various formats like JSON, XML, or even plain text. Spring Boot can automatically convert the incoming request body into a corresponding Java object using a message converter. The message converter is chosen based on the Content-Type header of the incoming request.

For example, if the Content-Type header is set to application/json, Spring Boot will use a JSON message converter to convert the request body to a corresponding Java object. Similarly, if the Content-Type header is set to application/xml, Spring Boot will use an XML message converter to convert the request body.

(Note: The keys of JSON are same as the Java Object properties.)

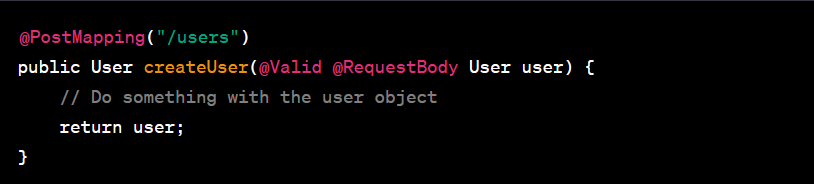


In this example, we define a **@PostMapping** endpoint that creates a new user. The **@RequestBody** annotation is used to extract the user object from the request body. The user object is then used to create a new user, and the created user is returned as the response.

This tells Spring Boot to deserialize the JSON payload of the HTTP POST request into a **User object** and pass it as a parameter to the **createUser()** method.

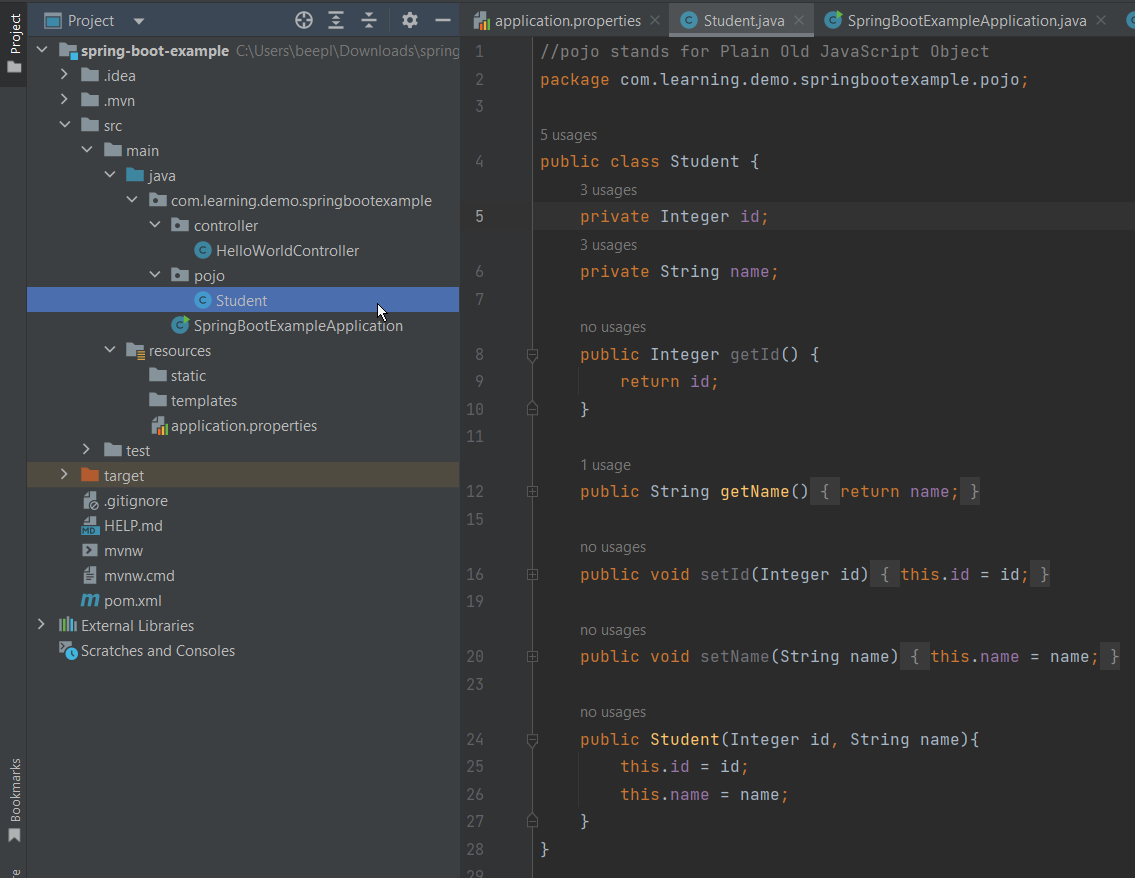
Note that the **@RequestBody** annotation can be used with any HTTP method that has a request body, such as POST, PUT, and PATCH.

You can also use the **@Valid** annotation along with **@RequestBody** to perform data validation on the request body. For example:

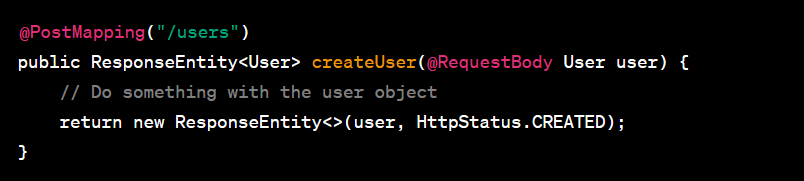


In this example, the **@Valid** annotation is used to perform validation on the user object before it is processed. If the user object fails validation, Spring Boot will return a 400 Bad Request response.

***Keep all the Java Objects in a package called ‘pojo’ which stand for “Plain Old Java Object”.***



## Another Example

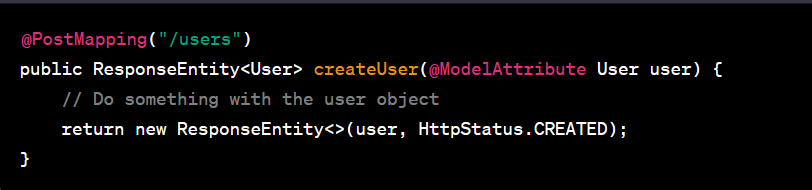


In this example, we define a **@PostMapping** endpoint that creates a new user. The **@RequestBody** annotation is used to extract the user object from the request body. The user object is then used to create a new user, and the created user is returned as the response in a **ResponseEntity** object with a 201 Created status code.

## @ModelAttribute

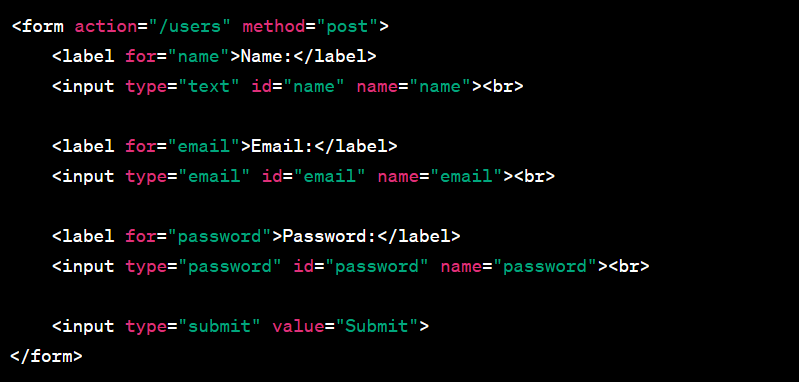
**@ModelAttribute** is an annotation in Spring Boot that is ***used to bind request parameters to a model attribute***. ***When used in a controller method, it binds the request parameter values to a model object before the method is executed***.

The request parameters are typically sent as HTML form data or query string parameters.



In this example, we define a **@PostMapping** endpoint that creates a new user. The **@ModelAttribute** annotation is used to bind the request parameters to the **User** object. When the method is called, Spring will automatically populate the **User** object with the values of the request parameters.

To send data from the client to the server using **@ModelAttribute**, you can use a form with input fields that correspond to the fields in the model object. For example, if you have a **User** model with **name**, **email**, and **password** fields, you can create a form like this:

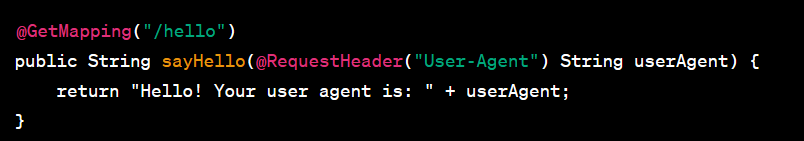


When the form is submitted, the values of the **name**, **email**, and **password** fields will be sent as request parameters to the server. Spring Boot will then use **@ModelAttribute** to bind those values to a **User** object in the controller method.

## @RequestHeader

**@RequestHeader** is a Spring Boot annotation used to access HTTP request headers in a controller method.

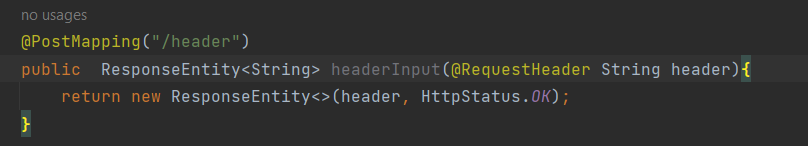
In Spring Boot, the **@RequestHeader** annotation is used to extract data from HTTP request headers. It can be used to get data such as the user-agent, content-type, accept, etc. sent in the headers of the HTTP request.



In this example, we are defining a GET endpoint **/hello** and using **@RequestHeader** to inject the **User-Agent** header value into the **userAgent** parameter.

When a client makes a GET request to **/hello**, Spring Boot will automatically extract the **User-Agent** header from the HTTP request and pass its value to the **sayHello** method. The method will then return a greeting message that includes the user agent value.

Note that if the specified header is not present in the request, Spring Boot will throw an exception at runtime.



## @Service

This annotation is used to mark a class as a service. Services provide business logic for your application.

