**SPRING BOOT**

[**1. HOW TO CREATE A SPRING BOOT PROJECT? 2**](#_3i4y1fskl1va)

[**2. BETTER WAY OF CREATING A PROJECT? 3**](#_1ikl672ym01l)

[**3. HOW TO TEST THE CREATED PROJECT? 4**](#_cpalgz3mon3f)

[**4. HOW TO ADD WEB WIZARD IN STS? 5**](#_o7gs2q8dvubn)

[**5. HOW TO ADD JSP IN STS? 6**](#_56u6686yxdxx)

[**6. HOW TO RUN JSP ON SERVER? 7**](#_5yzwvug2kbh7)

[**7. HOW TO IMPLEMENT JPA ? 8**](#_4wdqitu0114h)

[**8. HOW TO USE JPQL QUERY IN JPA ? 10**](#_jvvbb7mpuc58)

[**9. HOW TO USE NATIVE QUERY IN JPA ? 11**](#_swyhbn6dnswo)

[**10. WHAT IS SPRING BOOT DEV-TOOLS? 12**](#_ktk3mj4e3b7z)

[**11. HOW TO CREATE A GET REST API? 13**](#_bmhiw4nc6z8v)

[**12. HOW TO CREATE A POST REST API? 14**](#_kxhrb5vetxkn)

[**13. HOW TO CREATE A PUT/UPDATE REST API? 15**](#_5826lck60hcv)

[**14. HOW TO CREATE A DELETE REST API? 16**](#_gfnrjncozmtp)

[**15. HOW TO SEND RESPONSE CODE LIKE 200, 404, ETC. ? 17**](#_4qrwoh90papt)

[**16. CREATE REST API AND CONNECT IT TO DATABASE 18**](#_tv3de33xk22p)

[**17. CREATE REST API WITH CUSTOM/NESTED ENTITIES 22**](#_a8b9md9ect5b)

[**18. HOW TO CREATE A BI-DIRECTIONAL MAPPING OF ENTITIES UNSING JACKSON? 23**](#_g64yrbqqgp59)

[**19. HOW TO SETUP THYMELEAF? 24**](#_6h1gdgadabz0)

[**20. HOW TO LOOP IN THYMELEAF? 26**](#_vxy8tvbpu6fi)

[**21. CONDITIONAL STATEMENTS IN THYMELEAF? 27**](#_8rqymy19fch6)

[**22. FRAGMENTS IN THYMELEAF? 28**](#_s3tyuytauig)

[**23. HOW TO PASS DATA TO THE FRAGMENTS FROM CONTROLLER IN THYMELEAF? 29**](#_s9i6kybzazu1)

[**24. HOW TO DO INHERITANCE OF HTML FILES IN THYMELEAF? 30**](#_s4s5ogw72gfk)

# 

# HOW TO CREATE A SPRING BOOT PROJECT?

* + Visit <https://start.spring.io> website
  + Select gradle or maven as project type
  + Select java or kotlin as language
  + Give project name, package name, etc.
  + Select jar or war package type
  + Select java version
  + Select dependencies:
    1. Web
    2. Jasper
    3. Security
    4. Devtools
    5. JPA
    6. MYSQL
  + Click generate. This will create a zip file of the project
  + Open this folder with an IDE

# BETTER WAY OF CREATING A PROJECT?

* + Open spring tool suit STS IDE
  + Click new and select spring starter project
  + Fill required fields mentioned above in point number 1

# HOW TO TEST THE CREATED PROJECT?

* + Make sure that the annotation above the Application class is *@SpringBootApplication*
  + Create a Controller java class
    1. Annotate the class with *@Controller*
    2. Create a public method that returns a string
    3. Annotate this method with @ResponseBody. This will show the returned value on the web page, otherwise no value will be shown on the page.
    4. Annotate the method with @RequestMapping("/test").   
       “/test” could be anything you want.  
       This is required to find this method when we run the server.  
       When we run the server, we will write localhost:portnumber/test to call this method
    5. Save the project and click run spring boot app.
    6. Then the server will start
    7. Look for the post number in the logs
    8. Open a web browser and type localhost:portnumber/test to test the method that we have created. Now the string which we have written in the method shall be visible on the web page.

# HOW TO ADD WEB WIZARD IN STS?

* + Click help
  + Click install new software
  + Select all websites option from the work with drop-down list
  + Uncheck hide items that are already installed option
  + Search “web” word
  + Select eclipse web developer tools package and install it

# HOW TO ADD JSP IN STS?

* + Click help
  + Select eclipse marketplace option
  + Search “jsp” word
  + Install eclipse enterprise java and web developer tools package

# HOW TO RUN JSP ON SERVER?

* + Click plain src folder
  + Click main
  + Inside that create a folder called webapp
  + Inside that create a folder called views
  + Create a jsp file inside this folder
  + Now open src/main/resource
  + Click application.properties = here we will define the view resolver
  + Type following lines:  
    spring.mvc.view.prefix=/views/

spring.mvc.view.suffix=.jsp

* + Google embedded jasper maven library. Copy paste that library in pom.xml file under dependencies section. This is important for embedding our jsp file in the browser on our server.
  + Write some function in controller file. The return value of this function should be the name of our jsp file.   
    For eg. if we have a home.jsp file, then, the return value of the above said function should be return “home”.
  + The name and path of the function could be anything you want.
  + Now run start the server and hit this path, your jsp file will be loaded in the browser.

# HOW TO IMPLEMENT JPA ?

* + Integrate JPA & Mysql dependency
  + Open application.peoperties file and write following configuration:  
    spring.datasource.url=jdbc:mysql://localhost:3306/testb

spring.datasource.username=root

spring.datasource.password=password

spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL8Dialect

spring.jpa.show-sql=true

spring.jpa.hibernate.ddl-auto=update

* + Create a model class
    1. Write @Entity annotation on the class name
    2. Write data members of the class
    3. Annotate id data member with   
       @Id  
       @GeneratedValue(strategy = GenerationType.AUTO)
  + Then create an interface and extend it with CrudRepository<ModelClassName, Integer>
  + Open application class.
    1. In the main function, get the application context like this:  
       ApplicationContext context = SpringApplication.run(JpaExampleApplication.class, args);
    2. Get bean class like this:  
       Interface interface = context.getBean(Interface.class)
    3. Then create an object of the model class with some data in it.
    4. Finally save the model class in database using the interface instance like this:  
       Interface\_instance.save(model object);
    5. Make sure the the dependencies are correctly created in the pom.xml file. This has wasted many hours, because the default mysql dependency was itself written wrongly.
  + To create custom query function, write such method in the repository interface.  
    For example, suppose the User model class has a name data member, and we want to query the table by name, then; create a method like this:  
    public List<User> getUserByName(String name);
    1. To query the table by name and ID, create a method in repo interface like this:  
       public List<User> findByNameAndId(String name, int id);
    2. To know more about how to create such query method, google Mysql query creation. Here, we can learn to create methods with AND, OR, LESS THAN, GREATER THAN, etc. attributes.

# HOW TO USE JPQL QUERY IN JPA ?

* + To create a query function with a JPQL query, write a function in the repo interface.
  + Annotate it with @Query annotation
  + Write following query to get all the elements in the table:  
    @Query("select user from User user")  
      
    Here, User is the name of our model class.  
    Make sure that the spelling and upper-lowercase of the model class is written correctly here.
  + Write following query to get a user by name from the table:  
    @Query("select u from User u where user.name =:n")

public List<User> getUserByNamee(@Param("n") String name);  
  
Here name is the data member of the User model class.

The Param annotation defines the identity of the name parameter.

Identity of the name parameter is ‘n’.

* + Write following query to get a user by name and ID from the table:  
    @Query("select u from User u where u.n =:name and u.id =:id")

public User getUserByNameeAndId(@Param("n") String name, @Param("id") int id);

Here name & id are the data members of the User model class.

The Param annotation defines the identity of the name and id parameter.

Identity of the name parameter is ‘n’.

Identity of the id parameter is ‘id’.

# HOW TO USE NATIVE QUERY IN JPA ?

* + To create a query function with a JPQL query, write a function in the repo interface.
  + Annotate it with @Query annotation
  + Write following query to get all the elements in the table:

@Query(value = "select \* from user", nativeQuery = true)

public List<User> getAllUserss();

* + Write following query to get all the elements by name in the table:  
    @Query(value = "select \* from user where name =:name", nativeQuery = true)

public List<User> getUserByNameee(@Param("name") String name);

# WHAT IS SPRING BOOT DEV-TOOLS?

* Whenever we run some code on the server, and then, we do some code modifications, then we have to restart the server and run the application again, in order to see the modifications in the app.
* This is where the dev tools help us.
* With dev tools installed in our app, we don’t have to restart the server whenever we make some code modifications. The dev tools handles the server updation task automatically. And whatever modifications we have done, get deployed on the server without restarting the server.

# HOW TO CREATE A GET REST API?

* Create a simple model class Eg. Book class
* Create a simple java class and call it BookService class
  + Annotate this class as @Component
  + Populate some books in some function and call this function while creating an object of this class  
    OR   
    populate some books in the static block of this class
  + Then create functions like getAllBooks, getBookByID etc and do CRUD operations on the list of books which we have populated above and return respective results from these functions.
* Create a controller class
* Annotate this class as @Controller or @RestController.  
  @RestController is better.
* Create a data member of the service class like,  
  private BookService bookService;
* Annotate the above data member as @Autowired
* Now create various CRUD functions like getAllBooks, getBookByID, etc.
* To use the GET API, you can either annotate the functions as,  
  @RequestMapping(value = "/some\_name", method = RequestMethod.GET)  
  OR  
  @GetMapping(value = "/some\_name")  
  Second way is better
* Example of getAllBooks function:  
  @GetMapping(value = "/books")

public List<Book> getAllBooks() {

return this.bookService.getBooks();

}

* Example of getBookByID function:

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

public Book getBookById(@PathVariable("id") int id) {

return this.bookService.getBookByID(id);

}

# HOW TO CREATE A POST REST API?

* + Implement all the points explained in GET api above.
  + Create a function in the bookservice class to add/create a book. Here we just add the book to the array list.
  + In the controller create a addBook function. The parameter of the function should be the object of the Book model class. Also, write @RequestBody before the book object. So it would look like this:

public Book addBook(@RequestBody Book book) {

return this.bookService.addBook(book);

}

* Annotate this function as @PostMapping(“/books”)
* So the function finally looks like this:  
  @PostMapping("/books")

public Book addBook(@RequestBody Book book) {

return this.bookService.addBook(book);

}

# HOW TO CREATE A PUT/UPDATE REST API?

* + Implement all the points explained in GET api above.
  + Create a function in the bookservice class to update a book. Here we check if the asked book is actually available or not, if available, then we update its data.
  + In the controller create a updateBook function. The parameter of the function should be the object of the Book model class and an ID.   
    Also, write @RequestBody before the book object.   
    And write @PathVariable before the ID.
  + So it would look like this:

public Book updateBook(@RequestBody Book book, @PathVariable("id") int id) {

return this.bookService.updateBook(book, id);

}

* + Annotate this function as @PutMapping("/books/{id}")

# HOW TO CREATE A DELETE REST API?

* + Implement all the points explained in GET api above.
  + Create a function in the bookservice class to delete a book. Here we check if the asked book is actually available or not, if available, then we delete the book from the array list.
  + In the controller create a deleteBook function. The parameter of the function should be an ID.
  + And write @PathVariable before the ID.
  + So it would look like this:

public Book deleteBookById(@PathVariable("id") int id) {

return this.bookService.deleteBookByID(id);

}

* Annotate this function as @DeleteMapping(value = "/books/{id}")

# HOW TO SEND RESPONSE CODE LIKE 200, 404, ETC. ?

* + Open the controller class.
  + Go to the getBook function. This function returns a List<Book> right now.
  + First of all, change this return type to ResponseEntity<List<Book>>.
  + Now, this function return this : return this.bookService.getBooks();
  + Instead of the above return statement, write a try catch block.
  + In the try section, get the list of books first:  
    List<Book> bookList = this.bookService.getBooks();
  + Then check if this list is empty or not.
  + If the list is ok, return the list of books like this:  
    if (bookList.size() > 0) {

return ResponseEntity.of(Optional.of(bookList));

}

* If the list is empty, return the appropriate response code like this:  
  else {

return ResponseEntity.status(HttpStatus.NOT\_FOUND).build();

}

* In the catch section, print the stack trace and return the appropriate response like this:  
  catch (Exception e) {

e.printStackTrace();

return ResponseEntity.status(HttpStatus.NOT\_FOUND).build();

}

* So finally the getBooks function look like this:

@GetMapping(value = "/books")

public ResponseEntity<List<Book>> getBooks() {

List<Book> bookList = this.bookService.getBooks();

try {

if (bookList.size() > 0) {

return ResponseEntity.of(Optional.of(bookList));

} else {

return ResponseEntity.status(HttpStatus.NOT\_FOUND).build();

}

} catch (Exception e) {

e.printStackTrace();

return ResponseEntity.status(HttpStatus.NOT\_FOUND).build();

}

}

* Apply similar functionality to other CRUD functions.

# CREATE REST API AND CONNECT IT TO DATABASE

* + Add following dependencies in the project
    1. Web
    2. Mysql connector
    3. Dev tools
    4. Spring-boot-starter-data-jpa
  + Open the application.properties class and write the following configurations:  
    spring.datasource.url=jdbc:mysql://localhost:3306/testb

spring.datasource.username=root

spring.datasource.password=password

spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL8Dialect

spring.jpa.show-sql=true

spring.jpa.hibernate.ddl-auto=update

* + Create a model class, say, Book
    1. Add data members to this class and set the getter-setter, tostring, constructor, etc
    2. Annotate this class with @Entity
    3. By default the class name would be the table name, but, if you want to change the table name, annotate the class with @Table(name = "books")
    4. Annotate the ID data member as

@Id

@GeneratedValue(strategy = GenerationType.AUTO)

* Create a repository interface like this:  
  public interface BookRepository extends CrudRepository<Book, Integer> {

}

* Create a BookService class and annotate it with @Component
* Write book repository as the data member as:  
  @Autowired

private BookRepository bookRepository;

* Then write different CRUD operations and use the repository class for CRUD tasks like this:

public List<Book> getAllBooks() {

return (List<Book>) bookRepository.findAll();

}

public Book getBookByID(int id) {

return bookRepository.findById(id).get();

}

public Book addBook(Book book) {

return bookRepository.save(book);

}

public void deleteBook(Book book) {

bookRepository.delete(book);

}

public void updateBook(Book book, int id) {

book.setId(id);

bookRepository.save(book);

}

* Create a BookController class and annotate it with @RestController
* Write book book service as the data member as:

@Autowired

private BookService bookService;

* Then write different CRUD operations and use the book service class for CRUD tasks like this:

@GetMapping(value = "/books")

public ResponseEntity<List<Book>> getBooks() {

List<Book> bookList = this.bookService.getAllBooks();

try {

if (bookList.size() > 0) {

return ResponseEntity.of(Optional.of(bookList));

} else {

return ResponseEntity.status(HttpStatus.NOT\_FOUND).build();

}

} catch (Exception e) {

e.printStackTrace();

return ResponseEntity.status(HttpStatus.NOT\_FOUND).build();

}

}

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

public ResponseEntity<Book> getBookById(@PathVariable("id") int id) {

Book book = this.bookService.getBookByID(id);

try {

if (book != null) {

return ResponseEntity.of(Optional.of(book));

} else {

return ResponseEntity.status(HttpStatus.NOT\_FOUND).build();

}

} catch (Exception e) {

e.printStackTrace();

return ResponseEntity.status(HttpStatus.NOT\_FOUND).build();

}

}

@PostMapping("/books")

public ResponseEntity<Void> addBook(@RequestBody Book book) {

try {

this.bookService.addBook(book);

return ResponseEntity.status(HttpStatus.OK).build();

} catch (Exception e) {

e.printStackTrace();

return ResponseEntity.status(HttpStatus.INTERNAL\_SERVER\_ERROR).build();

}

}

@DeleteMapping(value = "/books/{id}")

public ResponseEntity<Book> deleteBookById(@PathVariable("id") int id) {

try {

Book book = this.bookService.getBookByID(id);

if (book != null) {

this.bookService.deleteBook(book);

return ResponseEntity.status(HttpStatus.OK).build();

} else {

return ResponseEntity.status(HttpStatus.NO\_CONTENT).build();

}

} catch (Exception e) {

e.printStackTrace();

return ResponseEntity.status(HttpStatus.INTERNAL\_SERVER\_ERROR).build();

}

}

@PutMapping("/books/{id}")

public ResponseEntity<Book> updateBook(@RequestBody Book book, @PathVariable("id") int id) {

try {

Book \_book = this.bookService.getBookByID(id);

if (\_book != null) {

this.bookService.updateBook(book, id);

return ResponseEntity.status(HttpStatus.OK).build();

} else {

return ResponseEntity.status(HttpStatus.NO\_CONTENT).build();

}

} catch (Exception e) {

e.printStackTrace();

return ResponseEntity.status(HttpStatus.INTERNAL\_SERVER\_ERROR).build();

}

}

# CREATE REST API WITH CUSTOM/NESTED ENTITIES

* + Create a model class, say, Author
    1. Annotate this class with @Entity
    2. Write the data members of this class
    3. Annotate the id member with:  
       @Id

@GeneratedValue(strategy = GenerationType.AUTO)

* Create a model class, say, Book
  1. Annotate this class with @Entity
  2. Write the data members of this class
  3. Annotate the id member with:

@Id

@GeneratedValue(strategy = GenerationType.AUTO)

* 1. Write Author class instance as the data member of this Book class
     + Annotate this Author member with:

@OneToOne(cascade = CascadeType.ALL)

* Do all other things as before.

# HOW TO CREATE A BI-DIRECTIONAL MAPPING OF ENTITIES UNSING JACKSON?

* + Jackson is by default present in the spring boot project, no need for any extra dependency.
  + We have two entities Book and Author
  + Book entity has Author as the data member
  + To create bi-directional mapping of these entities do the following:
    1. Write Book data member in the Author class
    2. Annotate this book member with   
       @OneToOne(mappedBy = "author")

@JsonBackReference  
Here “author” is the name of the author data member in the book class, write the correct spelling.

* + 1. Open Book class and annotate the author data member with

@JsonManagedReference

# HOW TO SETUP THYMELEAF?

* + Install Thymeleaf plugin in IDE, for that:
    - Clik help
    - Install new software
    - Enter following URL in the field :   
      <http://www.thymeleaf.org/eclipse-plugin-update-site/>
    - This will show thymeleaf plugin option, install it.
  + Install emmet plugin in IDE, for that:
    - Clik help
    - Install new software
    - Enter following URL in the field :   
      <http://download.emmet.io/eclipse/updates/>
    - This will show emmet plugin option, install it.
  + Create a controller file.
    - Annotate the class name with @Controller
    - Don’t use @RestController annotation
    - Create a simple function and annotate it with @GetMapping("/about")
    - Write some end point in the @GetMapping() function, like, “/about” written above.
    - Write (org.springframework.ui.Model model) as the parameter of the simple function.
    - Add some data in the model like this:  
      model.addAttribute("name", "abcd");

model.addAttribute("age", 50);

* Return <some\_name> (We are going to create an HTML file, its name should be this some\_name)
* The entire function should look something like this:  
  @GetMapping("/about")

public String about(org.springframework.ui.Model model) {

model.addAttribute("name", "abcd");

model.addAttribute("age", 50);

return "about";

}

* Open src > main > resources > templates
* Create some\_name.html file
* In the <html> tag write lang="en" xmlns:th="http://www.thymeleaf.org"
* Write some basic HTML tags.
* To display the model data in the HTML tags, write:  
  <h1 th:text="${name}"></h1>

<h1 th:text="${age}"></h1>

* To perform string operations on the text, write like this:  
  For example, if you want to make the text upper case do like this:  
  <p th:text="${#strings.toUpperCase(name)}"></p>
* There are other options like ${#strings.} ${#array} etc.

# HOW TO LOOP IN THYMELEAF?

* + Write a simple GetMapping function as we did earlier.
  + Create a list of string elements in this function.
  + Add this list to the model.
  + So the final function would look like this:  
    @GetMapping("/loop")

public String loop(Model model) {

List<String> list = List.of("a", "b", "c", "d");

model.addAttribute("listData", list);

return "loop";

* Remember that, this Model is not some class created by us, but it is an in-built class of “org.springframework.ui” package.
* Then create an HTML file with name “loop.html”
* Since we are going to display list data, create a list tag in the HTML like this:

<ul>

<li th:each="n, status : ${listData}">

<p th:text="${status.index+1} + ' ' + ${n}"></p>

</li>

</ul>

* Above, we have created an un-ordered list. In the <li> tag we have used th:each attribute for looping.   
  listData is the name of the key which we have used in the model.addAttribute(), in the controller class.  
  “N” variable catches the data from the list one by one. So to print the list data, we just have print “n” in a th:text attribute as shown above.
* Th.each attribute comes with some handy helpers like finding index, even, odd, first, last, etc.  
  To do that, write another variable called status in the th.each attribute.  
  Then using this status variable we can call different functionalities like, index, even, odd, etc.
* Following code shows how to find if the element of the list is an odd element or not, and then accordingly coloring it red.  
  <ul>

<li th:each="n, status : ${listData}">

<p th:style="${status.odd} ? 'color:red' : ''"

th:text="${status.index+1} + ' ' + ${n}"></p>

</li>

</ul>

* Above, we have used the th.style attribute. Then we check if the current element is at odd index or not by using the status.odd. If it is at odd index, we color it red.
* Beware of double and single quotes here, it could be confusing.

# CONDITIONAL STATEMENTS IN THYMELEAF?

* + Create some function as discussed above in previous themeleaf chapters.
  + Store a boolean tru/false value in the model.
  + Now create an HTML file.
  + Create some tag and check if that boolean is true or not. If true the color the text green, else color it red.   
    The function looks something like this:  
    @GetMapping("/conditionalStatements")

public String conditionalStatements(Model model) {

boolean isActive = true;

model.addAttribute("isActive", isActive);

return "conditionalStatements";

}

* Then the HTML code looks like this:  
  <h1 th:text="${isActive}"

th:style="${isActive} ? 'color:green' : 'color:red'"></h1>

* Also, we can use if..unless statement.  
  This is how the html code looks like:  
  <p th:if="${isActive} == true ">Active</p>

<p th:unless="${isActive} == true ">Inactive</p>  
If the isActive variable is true, then Active word will be displayed.

If the isActive variable is false, then Inactive word will be displayed .

# FRAGMENTS IN THYMELEAF?

* + We can create reusable HTML codes in separate html files and include them in our various html pages as fragments.
  + There are 3 ways to include these fragments.
    1. Include attribute
    2. Insert attribute
    3. Replace attribute
  + Create a simple function in the controller as did earlier and create its html file accordingly.
  + Suppose we want to have a footer in this above created html file. Since every or many such html files would want a footer, we will create a separate html file for this footer UI. this file is our fragment.
  + So create a separate html file for this footer html code. Create a footer in it.
  + In the footer tag write the fragment attribute like this:  
    <footer id= "fragment-id" th:fragment="fragment\_name">

<h1>This is a footer</h1>

</footer>

* Then open the main html file, create a footer tag there and use either include, insert or replace attribute in it. Like this:  
  <footer id= "host-id" th:insert="footer\_fragment::fragment\_name">

</footer>

* Here, the footer\_fragment is the html file name in which we have our footer fragment code written.   
  &  
  Fragment\_name is the name we have given to the fragment in the footer\_fragment.html file.
* When we use the include attribute, the content of the fragment is included in the footer tag of the host file. While the footer tag in the fragment file is excluded, only the content inside that tag is included.
* When we use insert attribute, the entire footer tag along with its content is inserted in the footer tag of the host file. So in the host file, we get one footer tag inside another footer tag, and then the content is inside that.
* When we use replace attribute, the host footer tag along with its content is totally replaced by the fragment footer tag along with its content.

# HOW TO PASS DATA TO THE FRAGMENTS FROM CONTROLLER IN THYMELEAF?

* + In the controller function which we created earlier for the fragment example, put some data in the model.  
    @GetMapping("/footer")

public String footer(Model model) {

model.addAttribute("title","THIS IS A TITLE");

return "footer";

}

* + Now go to the host html file, and create a parentheses after the fragment name and pass the data in it, which we have just added to the model in the controller:  
    <footer id= "host-id" th:insert="footer\_fragment::fragment\_name(${title})">

</footer>

* Now go to the fragment html file, and create a parentheses after the fragment name and accept the data in it and display that data in any html tag, like this:  
  <footer id= "fragment-id" th:fragment="fragment\_name(title)">

<h1 th:text=${title}></h1>

</footer>

# HOW TO DO INHERITANCE OF HTML FILES IN THYMELEAF?

* + Create a base.html file, which will have all those common UI things which many html files will need.
  + In the html tag write this:  
    <html lang="en" xmlns:th="http://www.thymeleaf.org"

th:fragment="layout(content)">

* If you want a common css for all the html files, then create a style.css file in the static folder.
* Now to include that css file in this base html file, write the following tag in the head tag:  
  <link rel="stylesheet" th:href="@{/style.css}"/>
* In the body tag, write those UI things which are common for all the other html files.
* Then write a tag which will load individual html files, like this:  
  <body>

<h1>THIS IS COMMON HEADER</h1>

<div th:replace="${content}"></div>

<h1>THIS IS COMMON FOOTER</h1>

</body>

Those html files which inherit this base.html file, will be loaded in this div tag.

* Then create some html file which will inherit this base.html file.
* Create the UI of this file. Suppose the main tag of this UI is a <p> tag.
* Then, write following lines in the html tag, so that this <p> tag will be loaded in the base.html file.  
  <html lang="en" xmlns:th="http://www.thymeleaf.org"

th:replace="base::layout(~{::p})">

* Finally, write a Getmapping function in the controller which will load this html file on the server. When you run the localhost URL, you will see that, the basic common things mentioned in the base.html have been loaded along with the specific UI things which are designed in the specific html file.
* So now this base.html file could be used as a common UI file for many html files.