Spring Start Here

Chapter-8:
Implementing web apps with Spring Boot and Spring MVC

Upcode Software
Engineer Team

-2024-

CONTENT

Introduction

- **What is dynamic view**

Dynamic views offer users the ability to directly control both the analytic sources they want to explore and when data is refreshed in visualizations

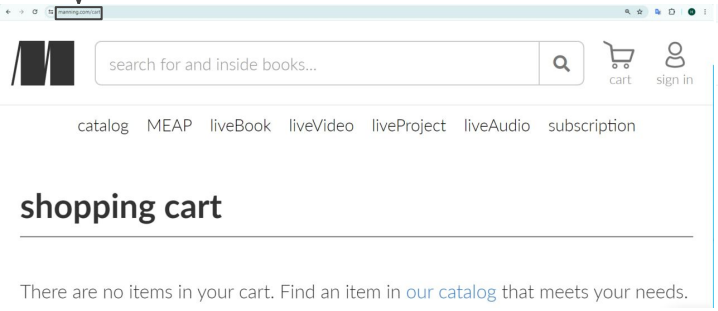
Dynamic views enable you to query and view relevant subsets of large data sets in charts that can be dynamically refreshed as selections are made.

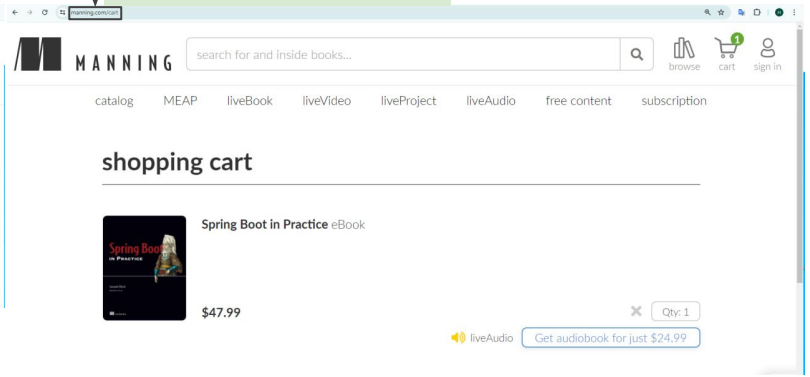
Implementing web apps with a dynamic view

The same web manning.com/cart displays different data. This is a dynamic view.

manning.com/cart

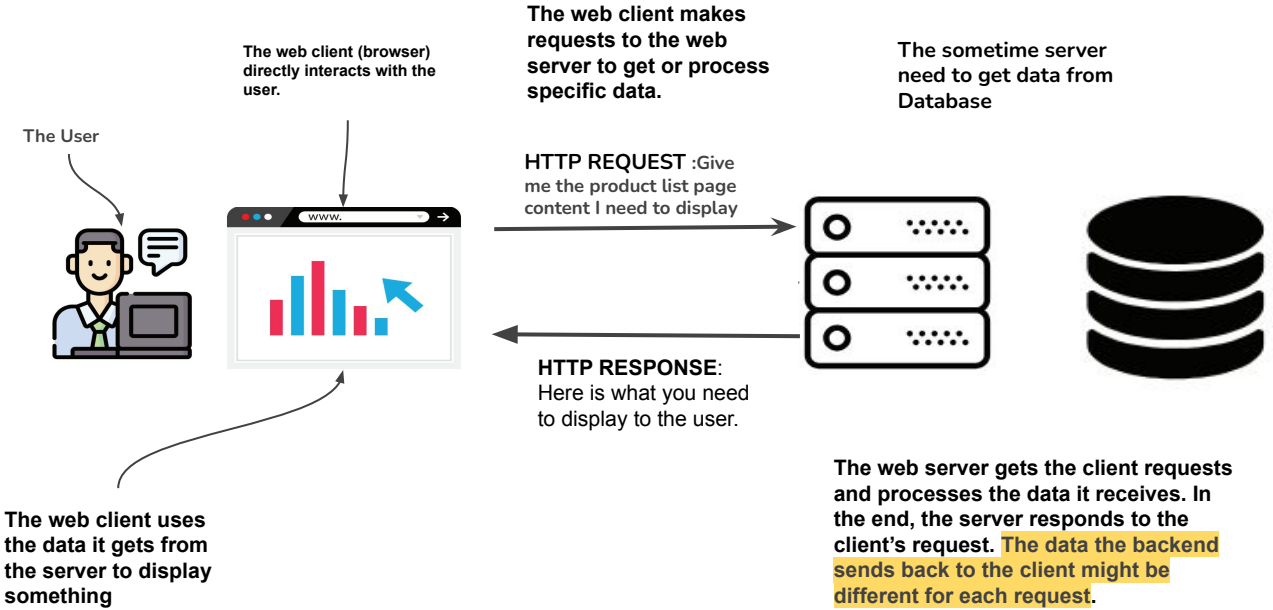
manning.com/cart





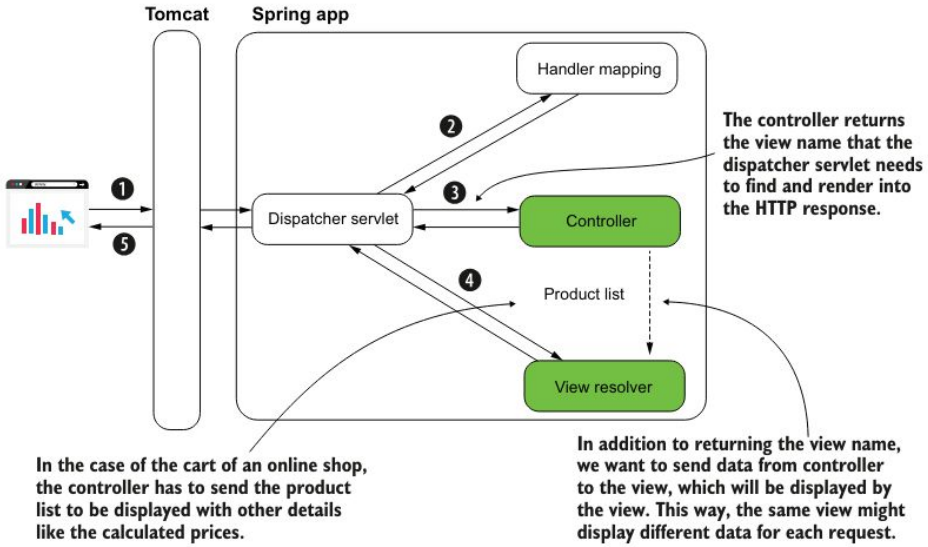
Getting data on the HTTP request(1/5)

- A client sends data through the HTTP request. The backend processes this data and builds a response to send back to the client. Depending on how the backend processed the data, different requests may result in other data displayed to the user.



Getting data on the HTTP request(2/5)

The Spring MVC flow. To define a dynamic view, the controller needs to send data to the view. The data the controller sends can be different for each request.



Getting data on the HTTP request(3/5)

1. The client sends an HTTP request to the web server.

2. The dispatcher servlet uses the handler mapping to find out what controller action to call.

3. The dispatcher servlet calls the controller's action.

4. After executing the action associated with the HTTP request, the controller returns the view name the dispatcher servlet needs to render into the HTTP response.

5. The response is sent back to the client.

Getting data on the HTTP request(4/5)

The home.html file representing the dynamic view of the app

```
@Controller
public class MainController {
    @RequestMapping("/home")
    public String home(Model page) {
        page.addAttribute( attributeName: "username",  attributeValue: "Katy");
        page.addAttribute( attributeName: "color",  attributeValue: "red");
        return "home.html";
    }
}
```

The displayed name took the value sent by the controller. The style is in red, which is the value sent by the controller

Getting data on the HTTP request(4/5)

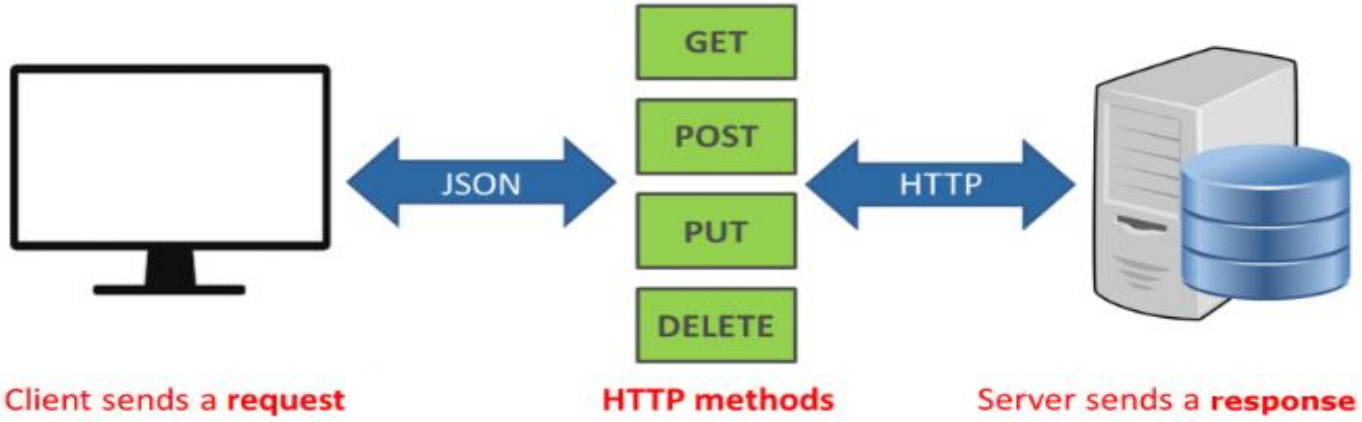
- We will create .html file on resources.templetes folder

Defines the Thymeleaf “th” prefix

Uses the “th” prefix to use the values sent by the controller

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>Home Page</title>
</head>
<body>
<h1>Welcome
    <span th:style="'color:' + ${color}"
          th:text="${username}"></span>!</h1>
</body>
</html>
```

Getting data on the HTTP request(5/5)



Adding dependencies

- **Thymeleaf is a modern server-side Java template engine that emphasizes natural HTML templates**

The dependency starter that needs to be added to use Thymeleaf as a template engine

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-thymeleaf</artifactId>
</dependency>
```

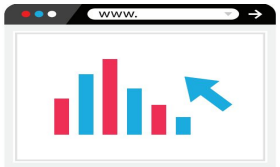
- Maven: org.thymeleaf:thymeleaf:3.1.2.RELEASE
- Maven: org.thymeleaf:thymeleaf-spring6:3.1.2.RELEASE
- Maven: org.unbescape:unbescape:1.1.6.RELEASE
- Maven: org.xmlunit:xmlunit-core:2.9.1
- Maven: org.yaml:snakeyaml:2.2

Using request parameters to send data (1/2)

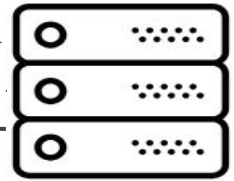
This is a query parameter expression. You use it to define request parameters in the path. We discuss the syntax later in this section

Here, the client sends a parameter identified with a key “brand” having the value “honda.”

http://example.com/products?brand=honda



HTTP REQUEST :: Show me all the motorbikes produced by Honda.



HTTP RESPONSE: : Here are the products produced by Honda.

The server returns all the products produced by the brand the user requested.

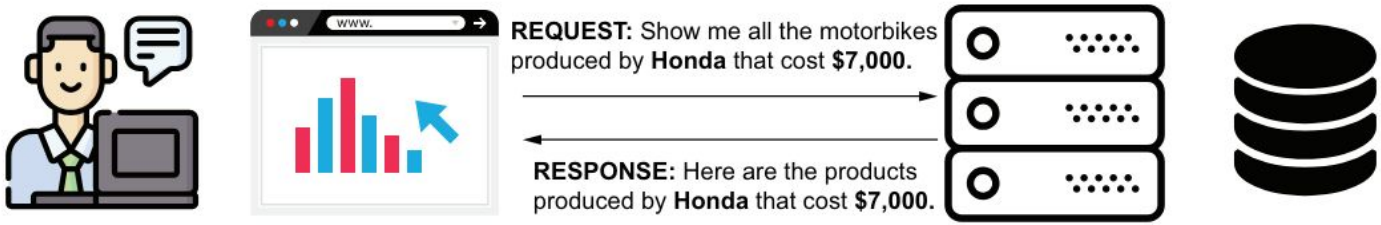
The web server gets the client requests and processes the data it receives. In the end, the server responds to the client’s request. The data the backend sends back to the client might be different for each request.

Using request parameters to send data (2/2)

- Request parameter can be optional. A common scenario for using request parameters is implementing a search functionality where the search criteria are optional. The client sends only some of the request parameters, and the server knows to use only the values it receives. You implement the server to consider it might not get values for some of the parameters

The client can also send the price request parameter. This parameter is optional. The server searches by its value only if the client sends it.

http://example.com/products**?brand=honda&price=7000**



Getting a value through a request parameter (1/2)

- **Color parameter value travels from the client to the controller's action on the backend to be used by the view**

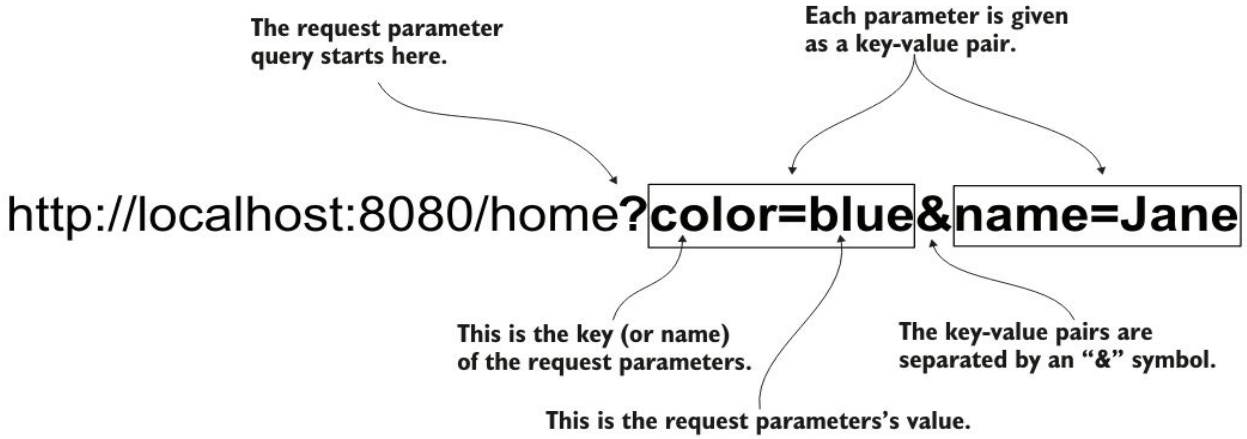
```
@Controller
public class MainController {
    @RequestMapping("/home")
    public String home(@RequestParam String color, Model page) {
        page.addAttribute( attributeName: "username",  attributeValue: "Katy");
        page.addAttribute( attributeName: "color", color);
        return "home.html";
    }
}
```

We define a new parameter for the controller's action method and annotate it with @RequestParam.

The controller passes the color sent by the client to the view.

Getting a value through a request parameter (1/2)

- Sending data through request parameters. Each request parameter is a key-value pair. You provide the request parameters with the path in a query starting with the question mark symbol. If you set more than one request parameter, you separate each key-value pair with the “and” (&) symbol.



Getting a value through a request parameter (2/2)

- It is a result of our HTTP request

Using path variables to get values from client(1/3)

```
@Controller
public class MainController {
    @RequestMapping("/home/{color}")
    public String home(
            @PathVariable String color,
            Model page) {
        page.addAttribute( attributeName: "username",  attributeValue: "Katy");
        page.addAttribute( attributeName: "color", color);
        return "home.html";
    }
}
```

To define a path variable, you assign it a name and put it in the path between curly braces.

We mark the parameter where we want to get the path variable value with the @PathVariable annotation. The name of the parameter must be the same as the name of the variable in the path.

Using path variables to get values from client(2/3)

- It is a result of our HTTP request

Using path variables to get values from client(3/3)

http://localhost:8080/home/green

```
@Controller
public class MainController {
    @RequestMapping("/home/{color}")
    public String home(
            @PathVariable String color,
            Model page) {
        page.addAttribute( attributeName: "username",  attributeValue: "Katy");
        page.addAttribute( attributeName: "color", color);
        return "home.html";
    }
}
```

The {color} path variable represents the value provided in the path.

The action method's parameter with the name of the path variable, annotated with @PathVariable, gets the value from the path

Path variables vs Request parameters

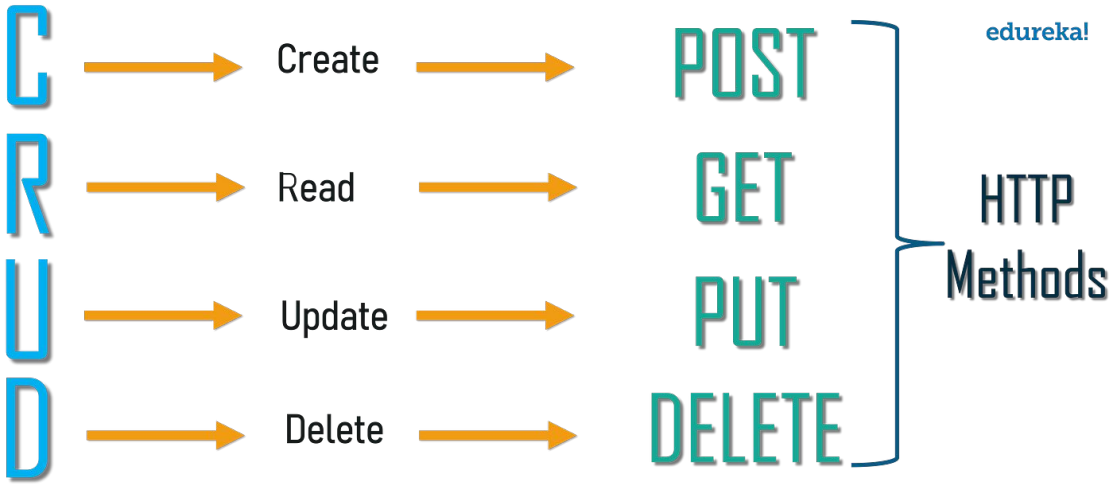
Request parameters	Path variables
1 Can be used with optional values.	1 Should not be used with optional values.
2 It is recommended that you avoid a large number of parameters. If you need to use more than three, I recommend you use the request body, as you'll learn in chapter 10. Avoid sending more than three query parameters for readability.	2 Always avoid sending more than three path variables. It's even better if you keep a maximum of two.
3 Some developers consider the query expression more difficult to read than the path expression.	3 Easier to read than a query expression. For a publicly exposed website, it's also easier for search engines (e.g., Google) to index the pages. This advantage might make the website easier to find through a search engine.

Using the GET and POST HTTP methods (1/8)

- Be careful! You can use an HTTP method against its designed purpose, but this is incorrect. For example, you could use HTTP GET and implement a functionality that changes data. Technically, this is possible, but it’s a bad, bad choice. Never use an HTTP method against its designed purpose.

Using the GET and POST HTTP methods (2/8)

Basic HTTP methods we will often encounter in web apps

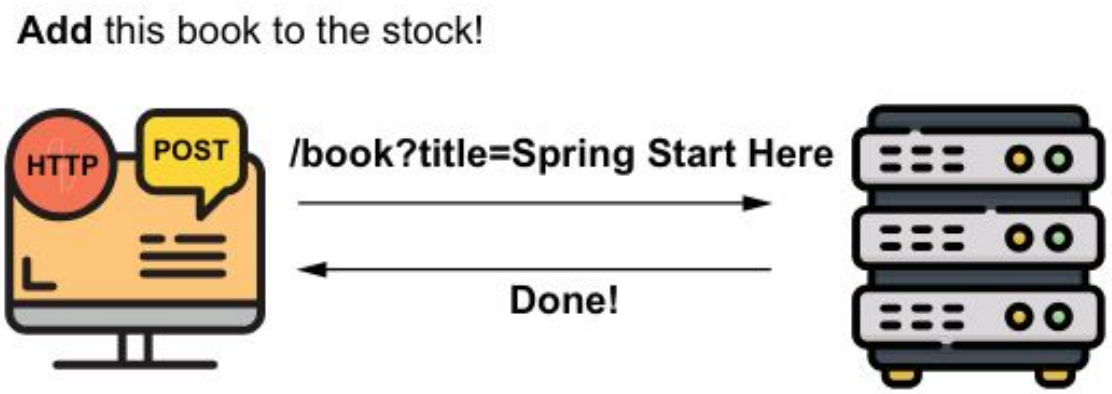


Using the GET and POST HTTP methods (3/8)

GET HTTP method - for retrieving data

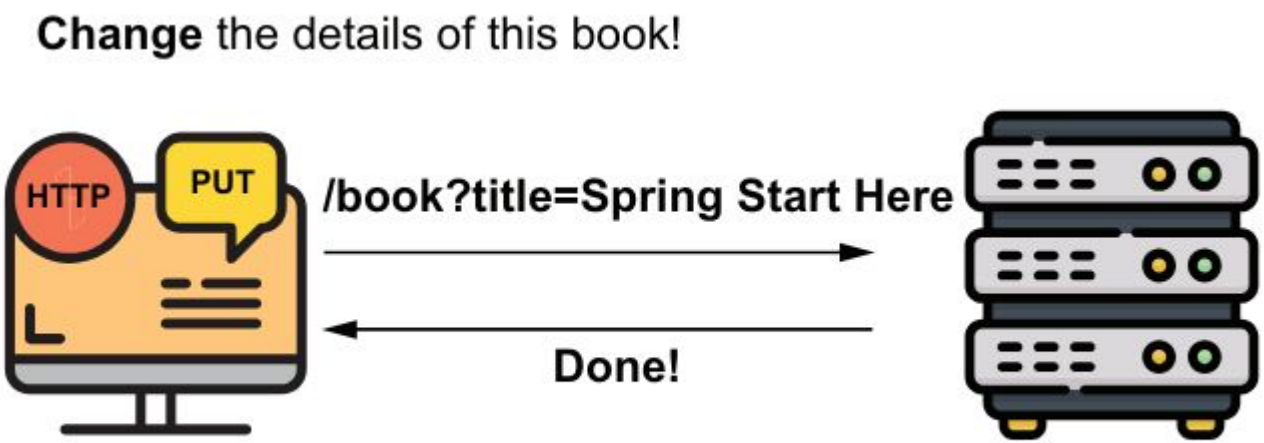
Using the GET and POST HTTP methods (4/8)

POST HTTP method - for adding data



Using the GET and POST HTTP methods (5/8)

PUT HTTP method - for changing a record



Using the GET and POST HTTP methods (6/8)

DELETE HTTP method - to remove data

Using the GET and POST HTTP methods (7/8)

```java
import com.example.demo.model.Product;
import com.example.demo.service.ProductService;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RequestParam;

@Controller
public class ProductsController {
    private final ProductService productService;  4 usages
    public ProductsController(
            ProductService productService) {
        this.productService = productService;
    }
    @RequestMapping(path = "/products",
            method = RequestMethod.POST)
    public String addProduct(
            @RequestParam String name,
            @RequestParam double price,
            Model model
    ) {
        Product p = new Product();
        p.setName(name);
        p.setPrice(price);
        productService.addProduct(p);
        var products = productService.findAll();
        model.addAttribute( attributeName: "products", products);
        return "products.html";
    }
    @GetMapping("/products")
    public String viewProducts(Model model) {
        var products = productService.findAll();
        model.addAttribute( attributeName: "products", products);
        return "products.html";
    }
}
```

add new product

getting all products

```java
package com.example.demo.model;

import lombok.Getter;
import lombok.Setter;

@Getter  7 usages
@Setter
public class Product {
    private String name;
    private double price;
}
```

ProductService.java products.html

```java
package com.example.demo.service;

import com.example.demo.model.Product;
import org.springframework.stereotype.Service;

import java.util.ArrayList;
import java.util.List;

@Service  3 usages
public class ProductService {
    private List<Product> products = new ArrayList<>();  2 usages
    public void addProduct(Product p) {  1 usage
        products.add(p);
    }
    public List<Product> findAll() {  2 usages
        return products;
    }
}
```

Using the GET and POST HTTP methods (8/8)

```html
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>Home Page</title>
</head>
<body>
<h1>Products</h1>
<h2>View products</h2>
<table>
    <tr>
        <th>PRODUCT NAME</th>
        <th>PRODUCT PRICE</th>
    </tr>
    <tr th:each="p: ${products}" >
        <td th:text="${p.name}"></td>
        <td th:text="${p.price}"></td>
    </tr>
</table>
<h4>   Add a new  product </h4>
<form action="/products" method="post">
    Name: <input
        type="text"
        name="name"><br />
    Price: <input
        type="number"
        step="any"
        name="price"><br />
    <button type="submit">Add product</button>
</form>
</body>
</html>
```

An input component allows the user to set the price of the product. The value in the component is sent as a request parameter with the key “price.”

When submitted, the HTML form makes a POST request for path /products

An input component allows the user to set the name of the product. The value in the component is sent as a request parameter with the key “name.”

The user uses a submit button to submit the form.

Using the GET and POST HTTP methods (8/8)

Products

View products

PRODUCT NAME PRODUCT PRICE

Add a new product

Name: Kitob
Price: 12000
Add product

Products

View products

PRODUCT NAME	PRODUCT PRICE
Kitob	12000.0

Add a new product

Name:
Price:
Add product

When App was run opened this window on http://localhost:8080/products link Product list is Empty now

When we click on this button : the product will be added to the list of products and we can see it here

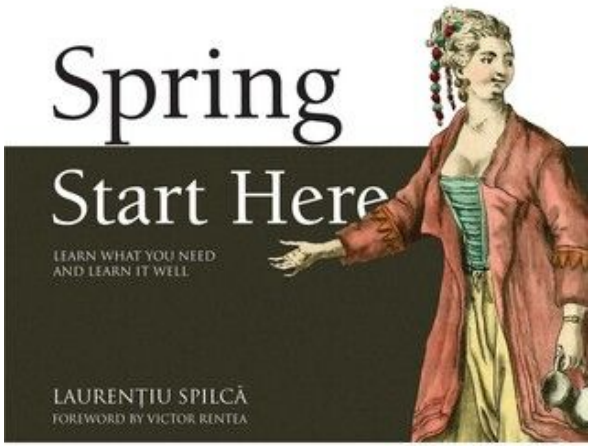
Conclusion

- The client can send data to the server through request parameters or path variables. A controller's action gets the details the client sends in parameters annotated with @RequestParam or @PathVariable

- A request parameter can be optional.

- Through a browser's HTML form process directly, you can use only HTTP GET and HTTP POST. To use other HTTP methods such as DELETE or PUT, you need to implement the call using a client language such as JavaScript

Resources



MANNING

Reference

1. Spring Start Here
2. spring.io
3. kysuit.net

Thank you!

Presented by

Qodirov Xudoyberdi

(qodirovhudoberdi4@gmail.com)