

Web Applications with Spring



JavaTM

Yevhen Berkunskyi, NUoS
eugeny.berkunsky@gmail.com
<http://www.berkut.mk.ua>

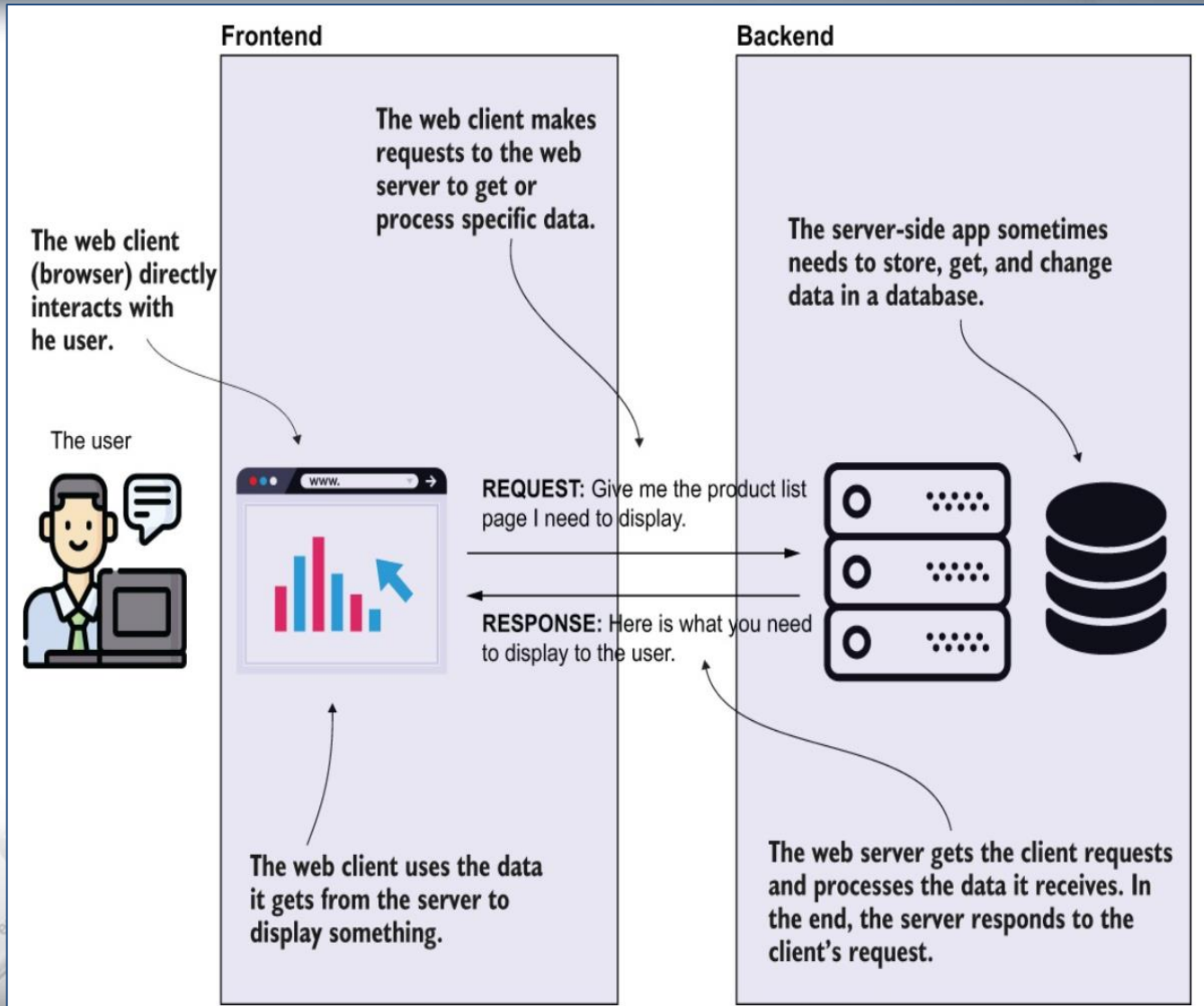
A general overview of a web app

A web app is composed of two parts:

- The **client side** is what the user directly interacts with. We also refer to the client side of a web app as the **frontend**.
- The **server side** receives requests from the client and sends back data in response. We also refer to the server side of a web app as the **backend**



A general overview of a web app

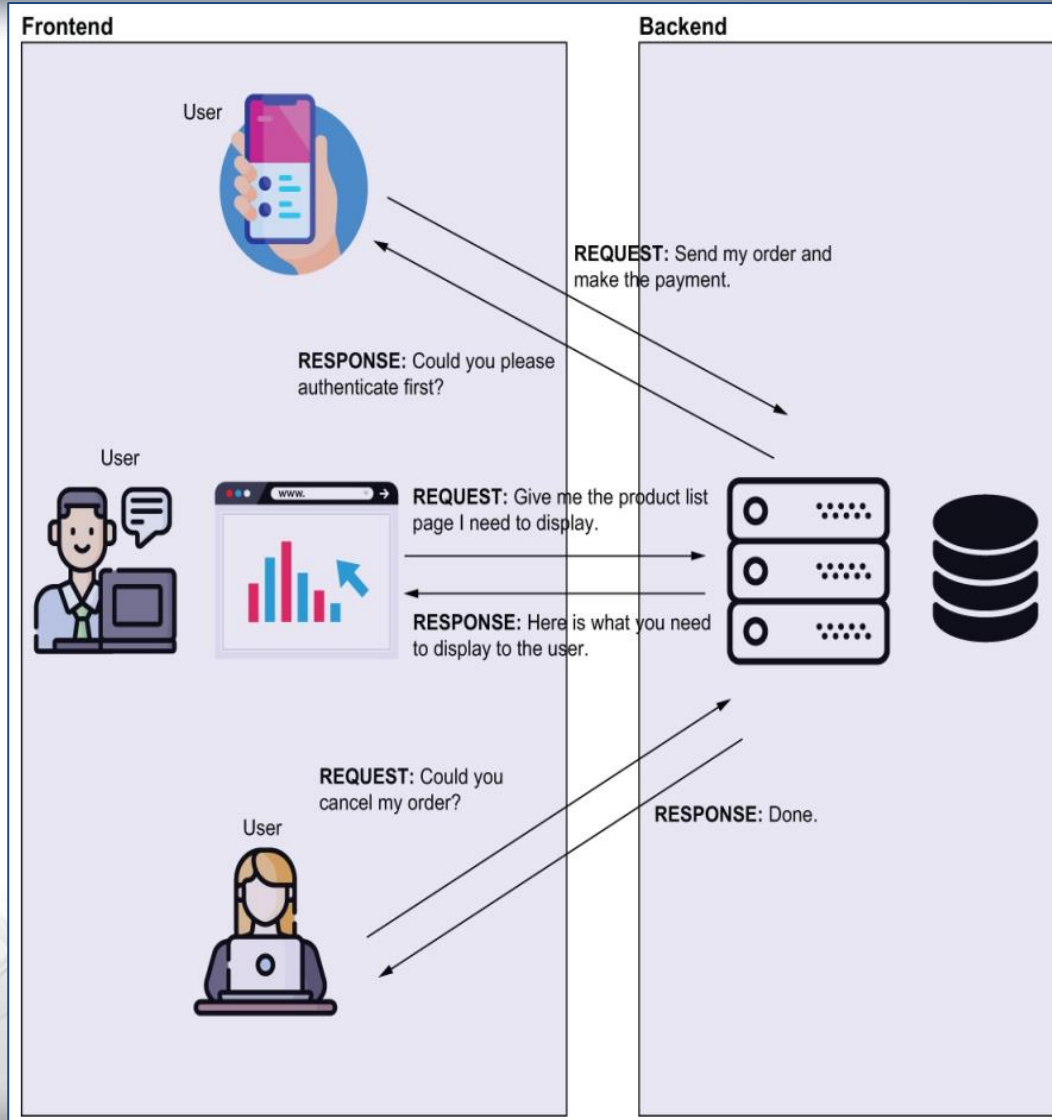


A general overview of a web app

- When discussing web apps, we usually refer to a client and a server, but it's important to keep in mind that the backend serves multiple clients concurrently.
- Numerous people may use the same app at the same time on different platforms.
- Users can access the app through a browser on a computer, phone, tablet, and so on



A general overview of a web app



Different fashions of implementing a web app with Spring

We classify the approaches of creating a web app as the following:

1. Apps where the backend provides the fully prepared view in response to a client's request. The browser directly interprets the data received from the backend and displays this information to the user in these apps.
2. Apps using frontend-backend separation. For these apps, the backend only serves raw data. The browser doesn't display the data in the backend's response directly. The browser runs a separate frontend app that gets the backend responses, processes the data, and instructs the browser what to display.

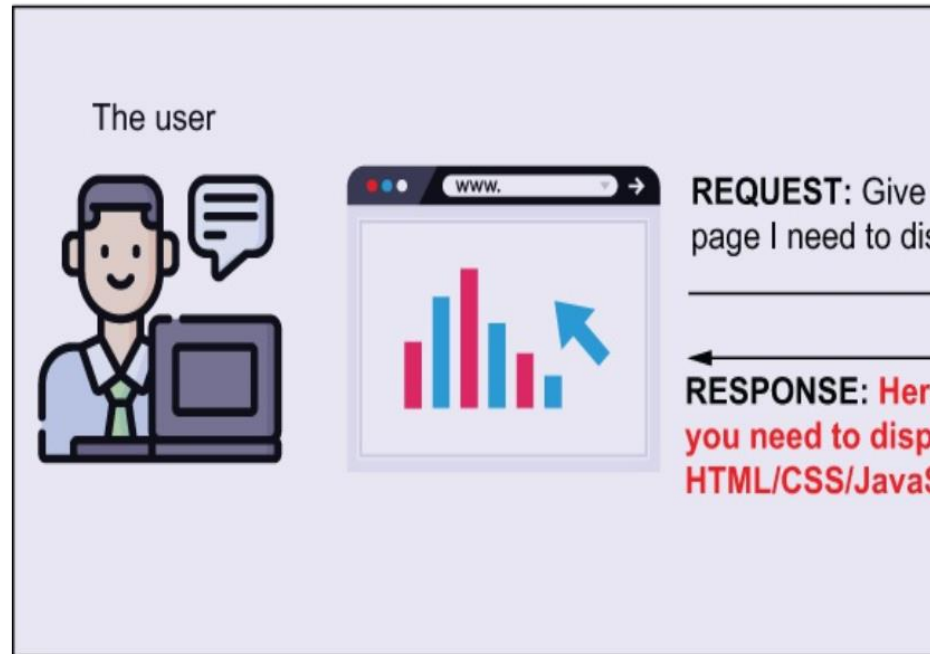
Standard web application

- Next slide presents the first approach in which the app doesn't use a frontend-backend separation.
- For these apps, almost everything happens on the backend side.
- The backend gets requests representing user actions and executes some logic.
- In the end, the server responds with what the browser needs to display.
- The backend responds with the data in formats that the browser can interpret and display, such as HTML, CSS, images, and so on.
- It can also send scripts written in languages that the browser can understand and execute (such as JavaScript)

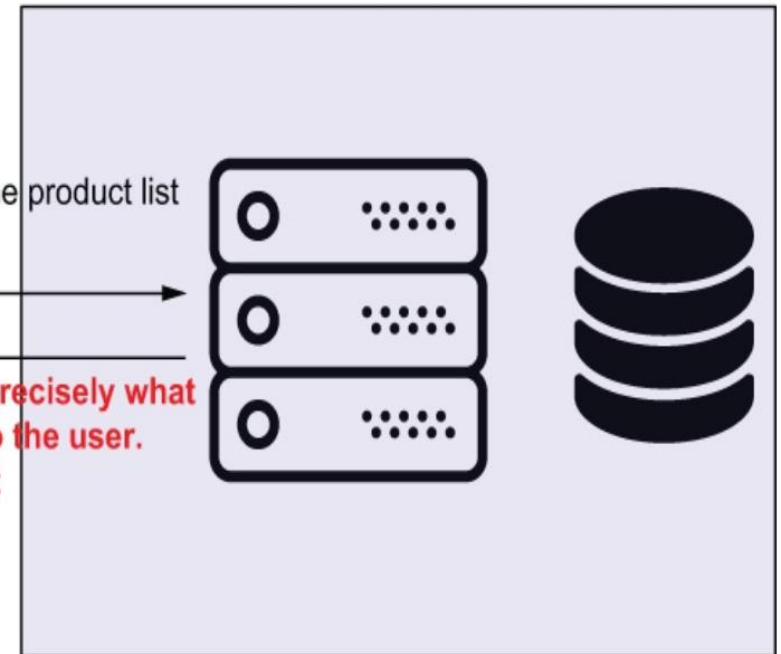


Standard web application

Frontend



Backend



REQUEST: Give me the product list page I need to display.

RESPONSE: Here is precisely what you need to display to the user.
HTML/CSS/JavaScript

In a standard web app, the client receives a response from the server containing exactly what the browser needs to display. The server-side app sends data in HTML, CSS, and JavaScript formats that the browser interprets and displays.

Frontend-backend separation

- Next slide shows an app using frontend-backend separation.
- Compare the server's response in the next slide with the response the server sends back in the previous slide.
- Instead of telling the browser precisely what to display, the server now only sends raw data.
- The browser runs an independent frontend app it loads at an initial request from the server.
- This frontend app takes the server's raw response, interprets it, and decides how the information is displayed.

Frontend-backend separation

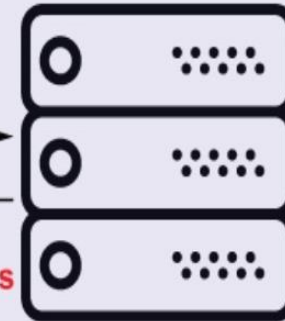
Frontend



REQUEST: Give me the product list page I need to display.

RESPONSE: Here is the JSON-formatted list of products. It's you who decides how to display it.

Backend



When we use frontend-backend separation, the server sends raw data to the client. For example, it might send just the list of products in a specific format. The client uses this data and decides how the browser will display it.

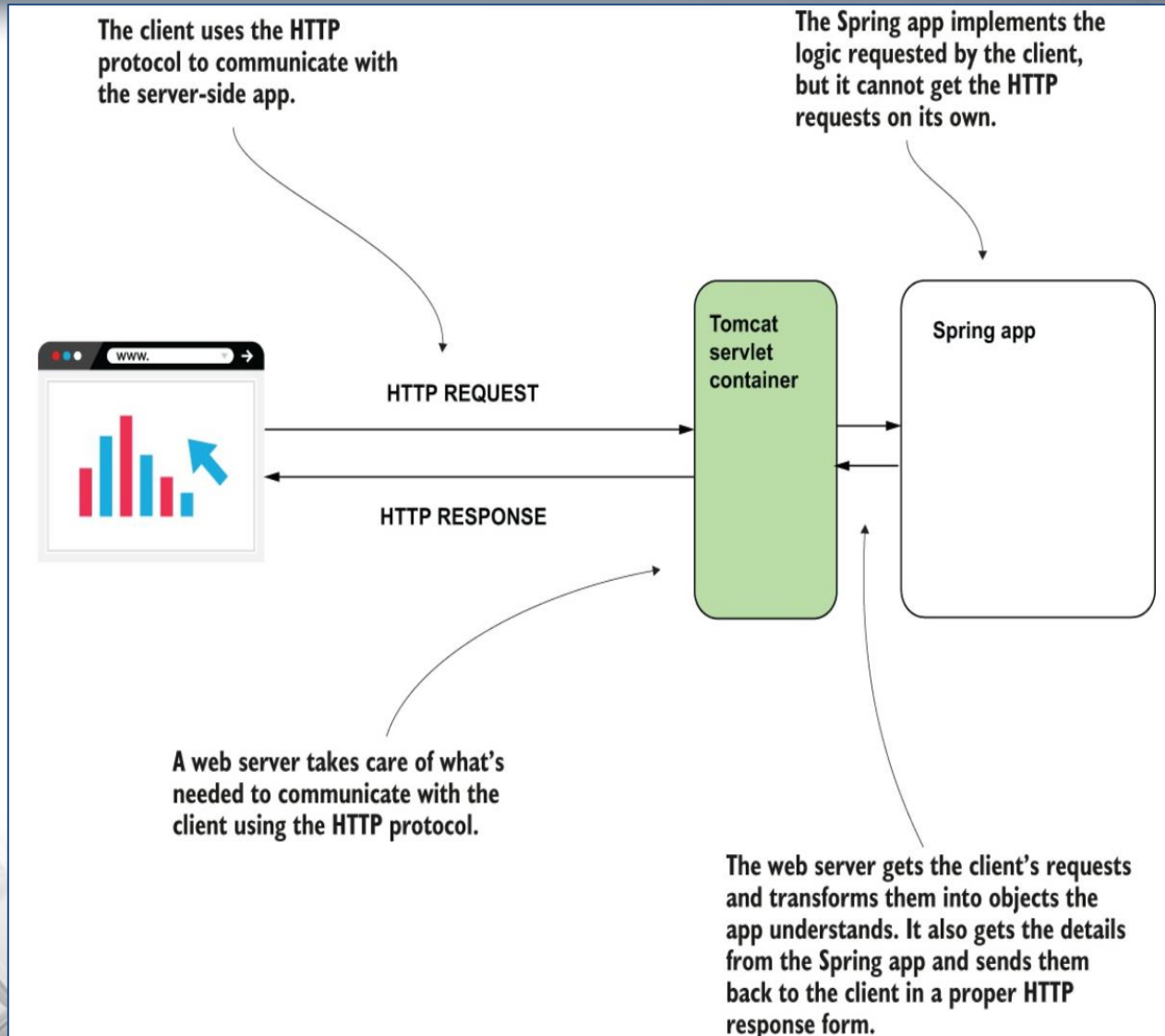
Using a servlet container in web app development

- One of the most important things to consider is the communication between the client and the server.
- A web browser uses a protocol named Hypertext Transfer Protocol (HTTP) to communicate with the server over the network.
- This protocol accurately describes how the client and the server exchange data over the network.
- But unless you are passionate about networking, you don't need to understand how HTTP works in detail to write web apps.
- As a software developer, you're expected to know that the web app components use this protocol to exchange data in a request-response fashion.
- The client sends a request to the server, and the server responds.
- The client waits for the response after every request it sends.

Servlet container - Tomcat

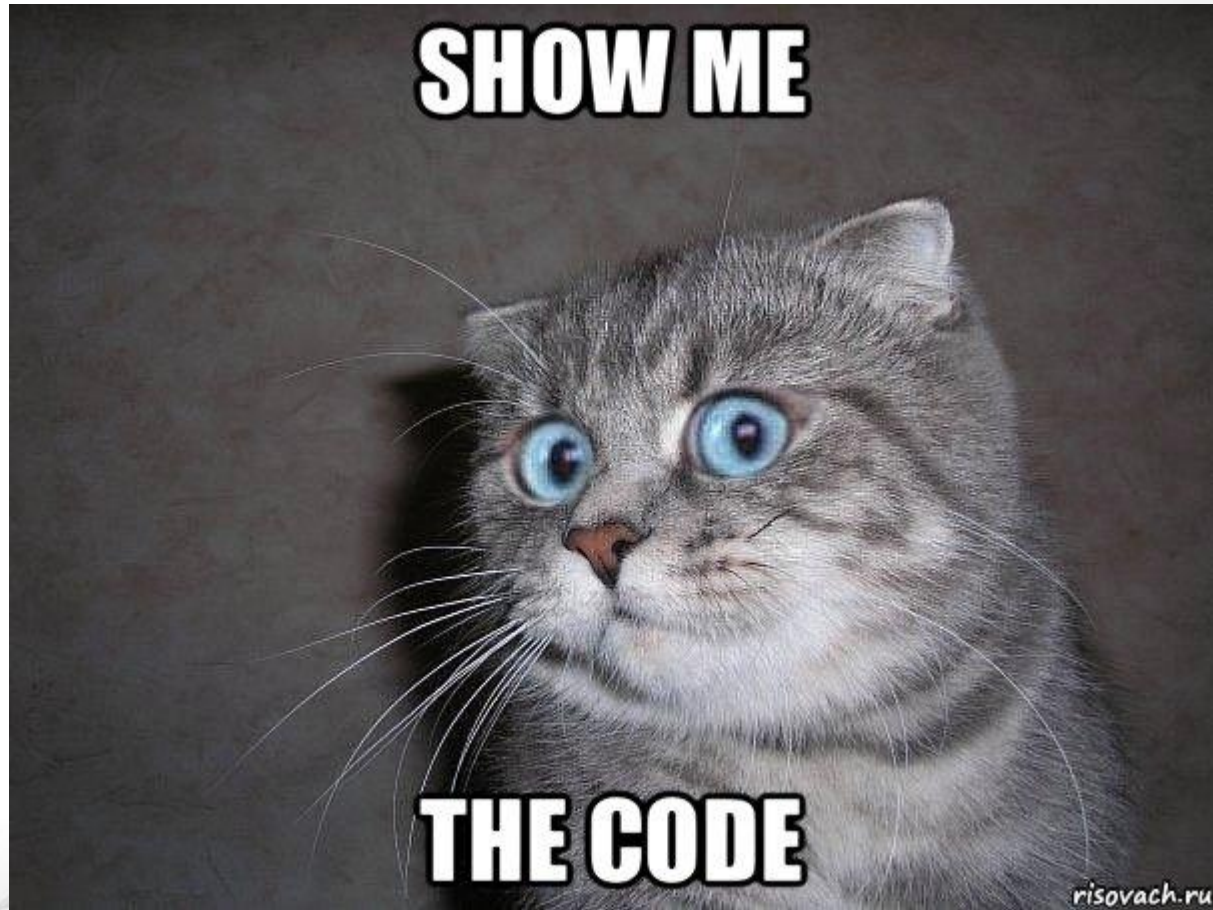
- In fact, what you need is not only something that understands HTTP, but something that can translate the HTTP request and response to a Java app.
- This something is a *servlet container* (sometimes referred to as a web server): a translator of the HTTP messages for your Java app.
- This way, your Java app doesn't need to take care of implementing the communication layer.
- One of the most appreciated servlet container implementations is **Tomcat**, which is also the dependency we'll use for the examples

Servlet container - Tomcat





Example



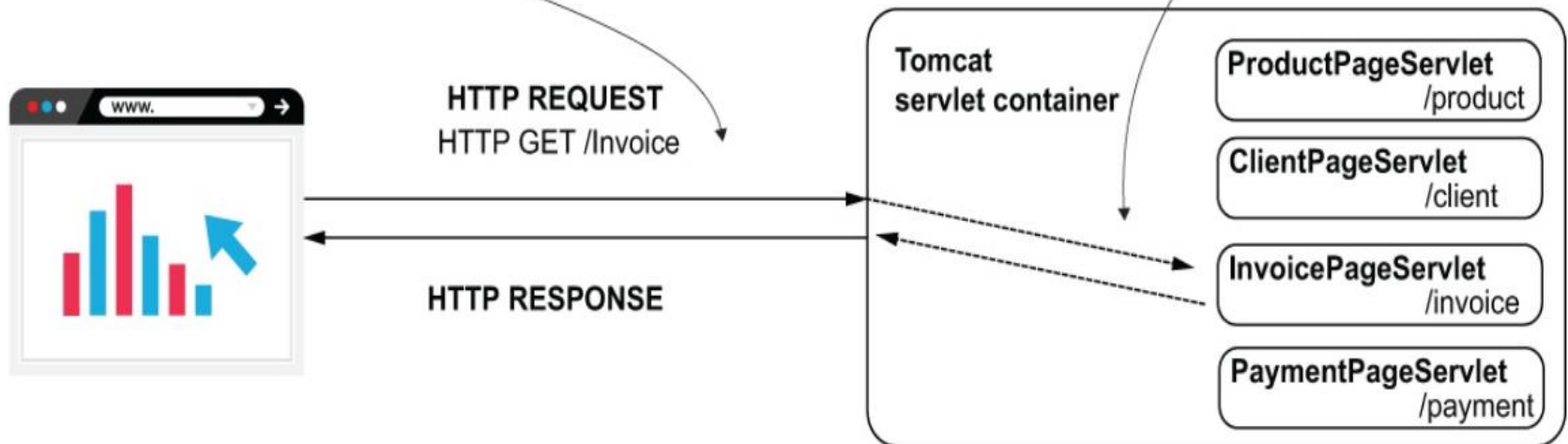
Servlet

- A servlet is a Java object that directly interacts with the servlet container.
- When the servlet container gets an HTTP request, it calls a servlet object's method and provides the request as a parameter.
- The same method also gets a parameter representing the HTTP response used by the servlet to set the response sent back to the client that made the request.
- Some time ago, the servlet was the most critical component of a backend web app from the developer's point of view.
- Suppose a developer had to implement a new page accessible at a specific path in the URL (e.g., /home/profile/edit, etc.) for a web app.
- The developer needed to create a new servlet instance, configure it in the servlet container, and assign it to a specific path

Servlets

When the client sends a request to a specific path, the servlet container calls the servlet registered at that specific path.

The servlet container translates the HTTP request and response to Java objects when calling the servlet associated with the path.



The servlet container registers multiple servlet instances. Each servlet is a Java object registered for a specific path.

Servlet container - Tomcat

- The servlet contained the logic associated with the user's request and the ability to prepare a response, including info for the browser on how to display the response.
- For any path the web client could call, the developer needed to add the instance in the servlet container and configure it.
- Because such a component manages servlet instances you add into its context, we name it a servlet container.
- It basically has a context of servlet instances it controls, just as Spring does with its beans.
- For this reason, we call a component such as Tomcat a servlet container.

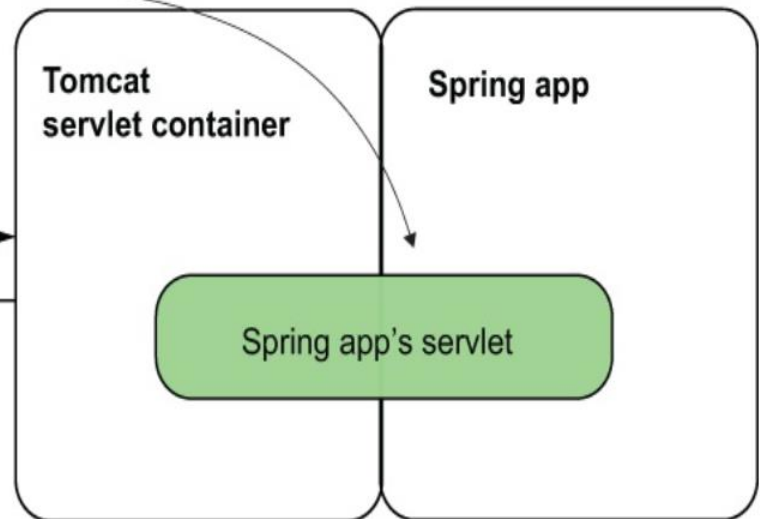
Servlet in Spring application

The Spring web app defines a servlet object. We register this object, so Tomcat calls it for any path of the client's request. This servlet becomes the entry point to our app's logic.



HTTP REQUEST

HTTP RESPONSE



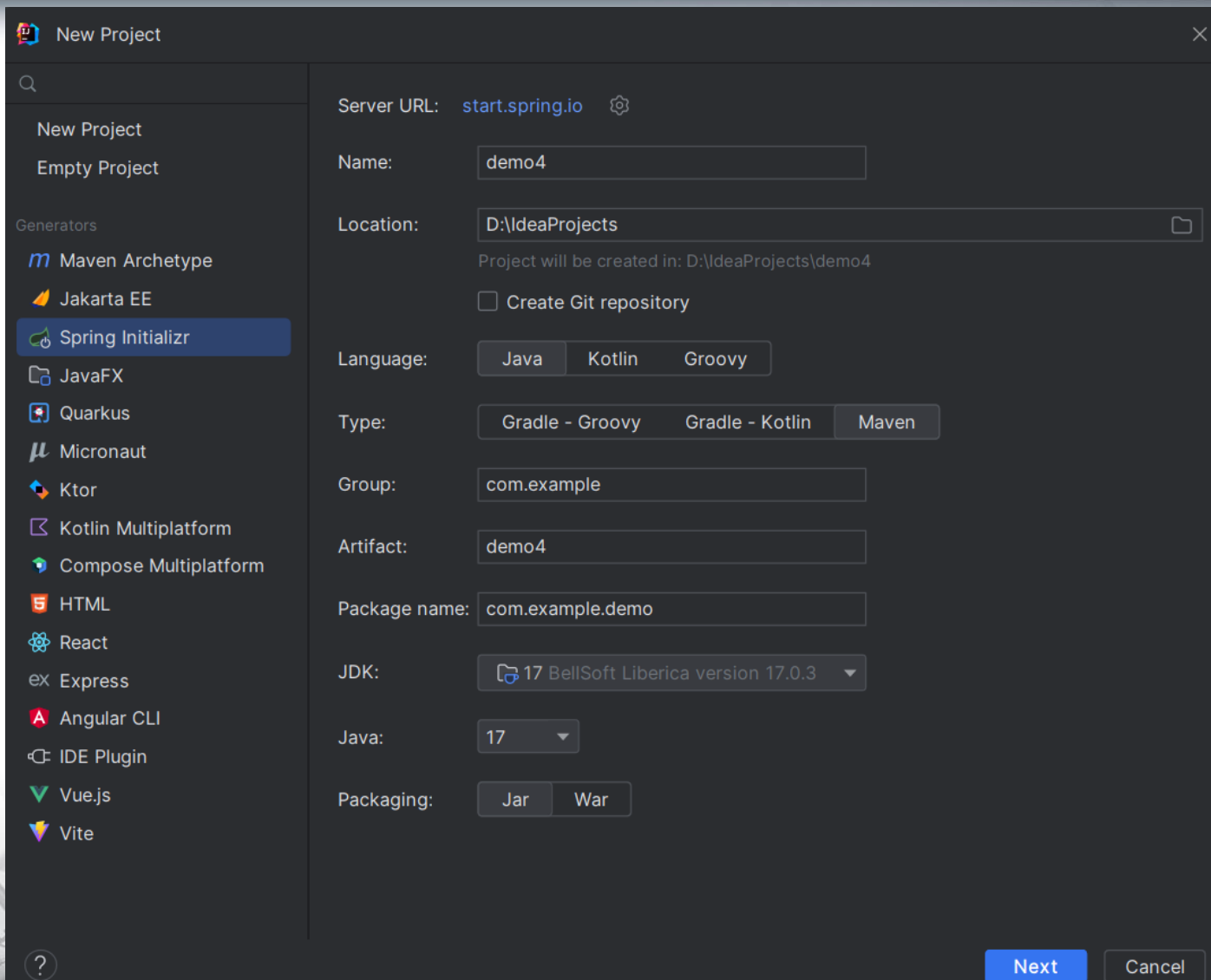
The magic of Spring Boot

- To create a Spring web app, we need to configure a servlet container, create a servlet instance, and then make sure we correctly configure this servlet instance such that Tomcat calls it for any client request.
- What a headache to write so many configurations!
- Many years ago, when I was learning/teaching Spring 3 (the latest Spring version at that time) and we configured web apps, this was the part both the students and I hated the most.
- Spring Boot is now one of the most appreciated projects in the Spring ecosystem. It helps you create Spring apps more efficiently and focus on the business code you write by eliminating a huge part of the code you used to write for configurations.
- Especially in a world of service-oriented architectures (SOA) and microservices, where you create apps more often, avoiding the pain of writing configurations is helpful.

Spring Boot features

- ***Simplified project creation*** — you can use a project initialization service to get an empty but configured skeleton app.
- ***Dependency starters*** — Spring Boot groups certain dependencies used for a specific purpose with dependency starters. You don't need to figure out all the must-have dependencies you need to add to your project for one particular purpose nor which versions you should use for compatibility.
- ***Autoconfiguration based on dependencies*** — based on the dependencies you added to your project, Spring Boot defines some default configurations. Instead of writing all the configurations yourself, you only need to change the ones provided by Spring Boot that don't match what you need. Changing the configs likely requires less code (if any).

Using a project initialization service to create a Spring Boot project



The screenshot shows the 'New Project' dialog in IntelliJ IDEA. The 'Generators' list on the left includes 'Spring Initializr', which is highlighted. The main configuration area on the right is set up for a new project:

- Server URL:** start.spring.io
- Name:** demo4
- Location:** D:\IdeaProjects
Project will be created in: D:\IdeaProjects\demo4
- ☐ Create Git repository
- Language:** Java (selected), Kotlin, Groovy
- Type:** Gradle - Groovy (selected), Gradle - Kotlin, Maven
- Group:** com.example
- Artifact:** demo4
- Package name:** com.example.demo
- JDK:** 17 BellSoft Liberica version 17.0.3
- Java:** 17
- Packaging:** Jar (selected), War

At the bottom right, there are 'Next' and 'Cancel' buttons.

Using a project initialization service to create a Spring Boot project

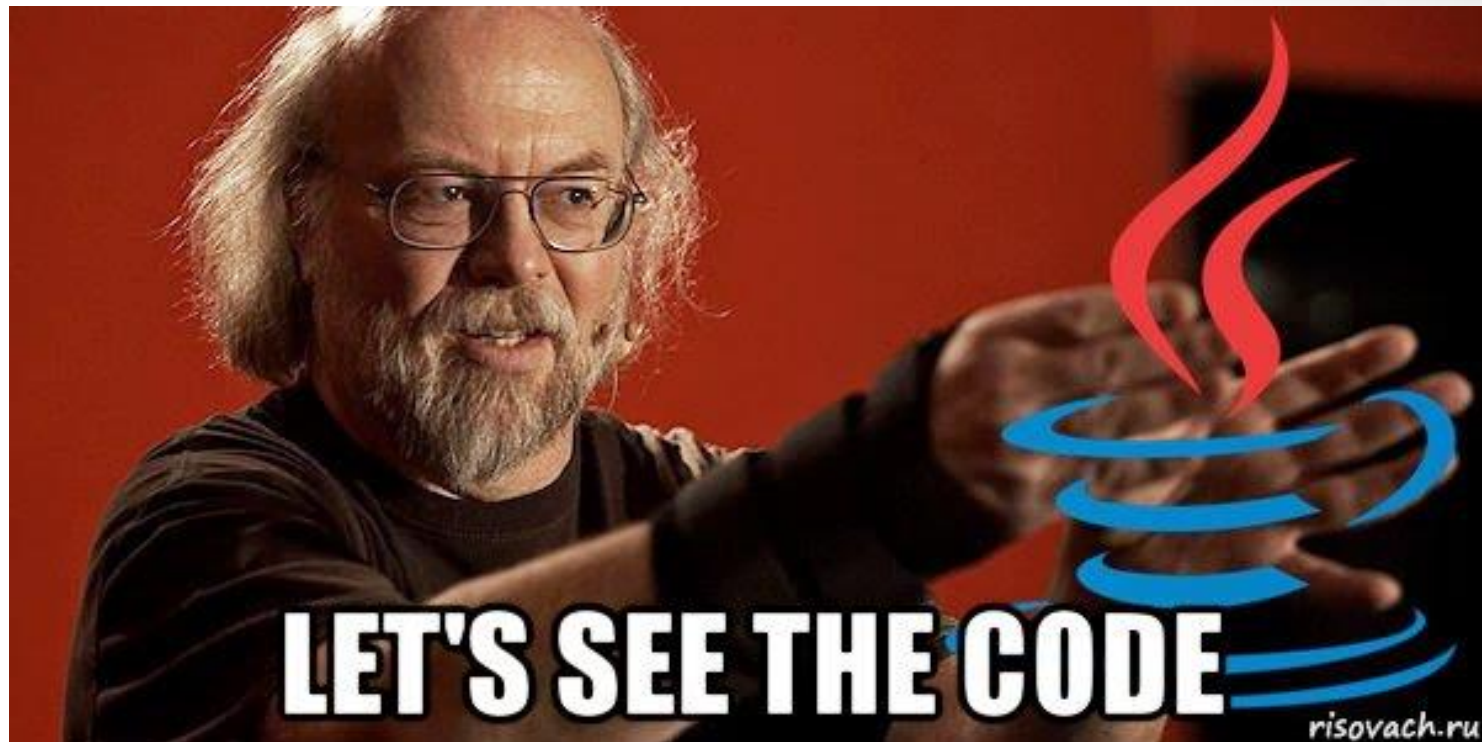
- If your IDE supports this feature, you'll probably find it named Spring Initializr in your project creation menu.
- But if your IDE doesn't support direct integration with a Spring Boot project initialization service, you can use this feature by accessing <http://start.spring.io> directly in your browser.
- This service will help you create a project you can import into any IDE.
- Let's use this approach to create our first project.
- The next slide summarizes the steps we'll take to create the Spring Boot project using start.spring.io



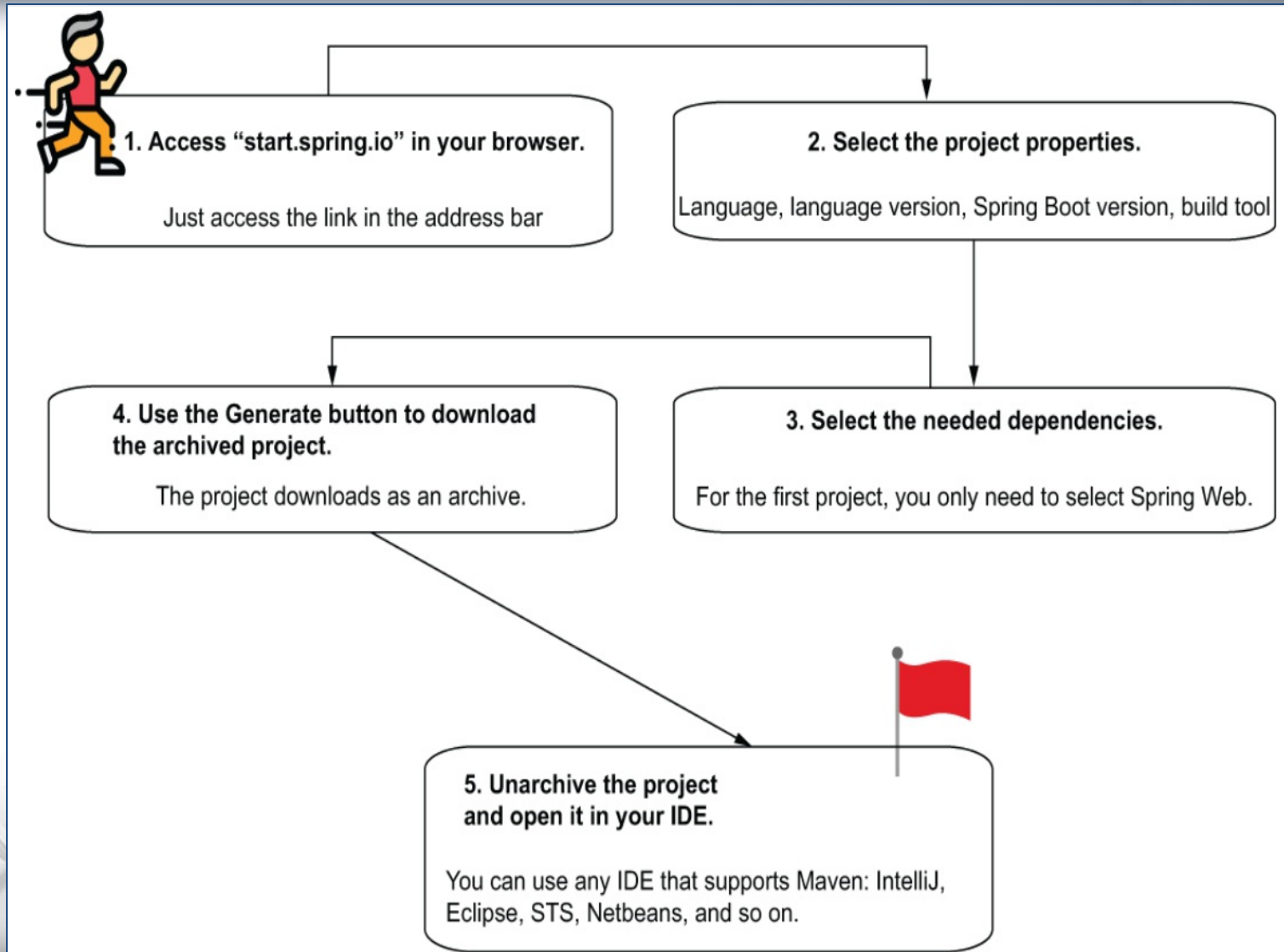


НАЦІОНАЛЬНИЙ
УНІВЕРСИТЕТ
КОРАБЛЕБУДУВАННЯ
ІМЕНІ АДМІРАЛА МАКАРОВА

Example



Using a project initialization service to create a Spring Boot project



Using a project initialization service to create a Spring Boot project



Here you will search and add dependencies to your project.

Project

- ☐ Gradle - Groovy
- ☐ Gradle - Kotlin
- ☒ Maven

Language

- ☒ Java
- ☐ Kotlin
- ☐ Groovy

Spring Boot

- ☐ 3.1.0 (SNAPSHOT)
- ☐ 3.1.0 (M1)
- ☐ 3.0.5 (SNAPSHOT)
- ☒ 3.0.4
- ☐ 2.7.10 (SNAPSHOT)
- ☐ 2.7.9

Project Metadata

Group

Artifact

Name

Description

Dependencies

ADD DEPENDENCIES... CTRL + B

No dependency selected

Here you select certain properties of your project, like the Java version, which build tool you prefer, and even the language you'd like to use. If you prefer to use a different syntax than Java, you can also develop your Spring Boot project with Kotlin and Groovy.

GENERATE CTRL + G

EXPLORE CTRL + SPACE

SHARE...

After configuring your project and selecting the needed dependencies, click this button to download the project in a zip archive.

Using a project initialization service to create a Spring Boot project



Project

- ☐ Gradle - Groovy
☐ Gradle - Kotlin
☒ Maven

Language

- ☒ Java ☐ Kotlin
☐ Groovy

Spring Boot

- ☐ 3.1.0 (SNAPSHOT) ☐ 3.1.0 (M1) ☐ 3.0.5 (SNAPSHOT)
☒ 3.0.4 ☐ 2.7.10 (SNAPSHOT) ☐ 2.7.9

Project Metadata

Group
Artifact
Name
Description

Give your project a name here.

Click ADD DEPENDENCIES and select Spring Web. Spring Web is the only dependency you need for this example.

Dependencies

ADD DEPENDENCIES... CTRL + B

Spring Web

WEB

Build web, including RESTful, applications using Spring MVC.
Uses Apache Tomcat as the default embedded container.

After adding the Spring Web dependency, you see it here.

When you click on the GENERATE button, your browser downloads a zip archive with the Maven project.

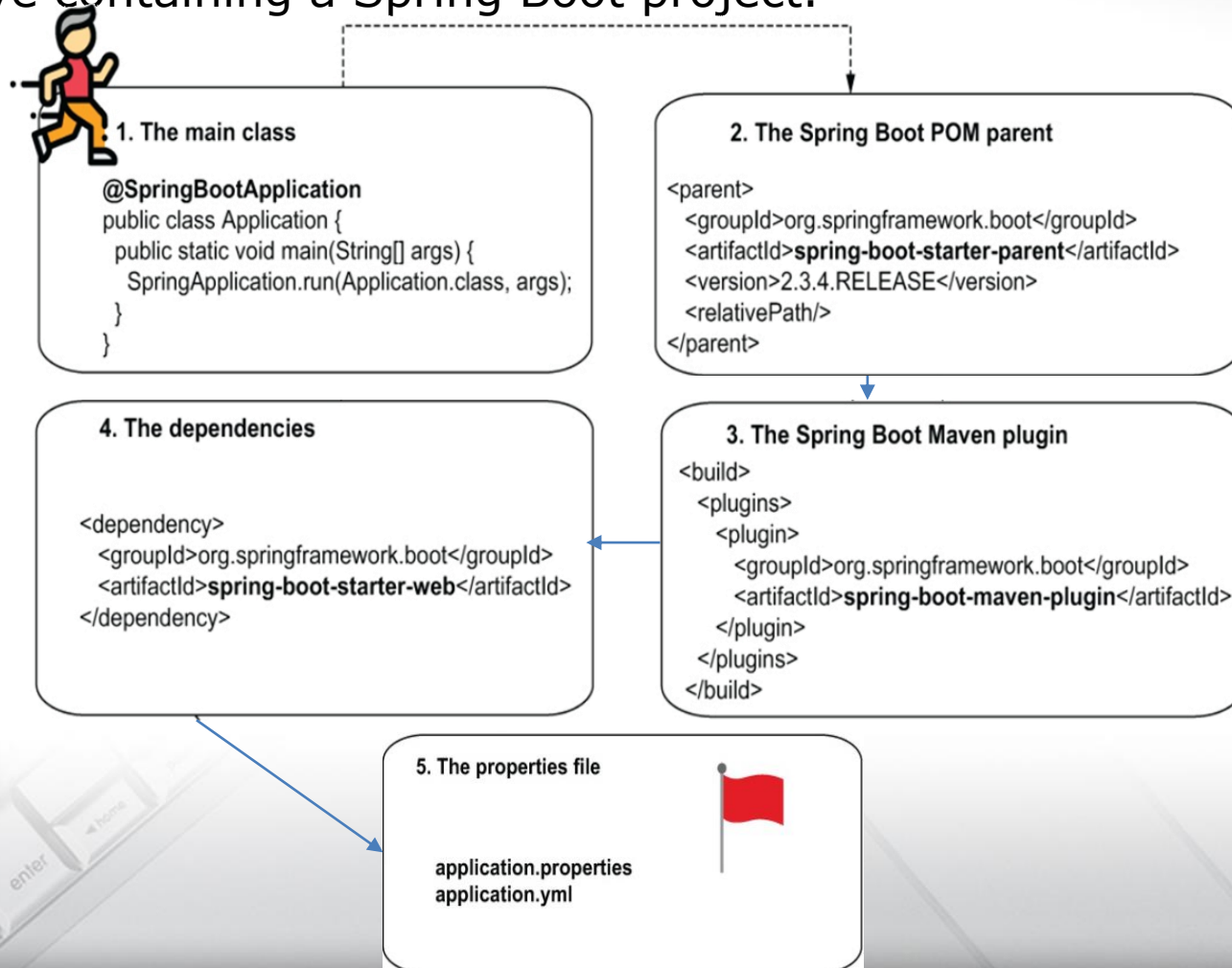
GENERATE CTRL + G

EXPLORE CTRL + SPACE

SHARE...

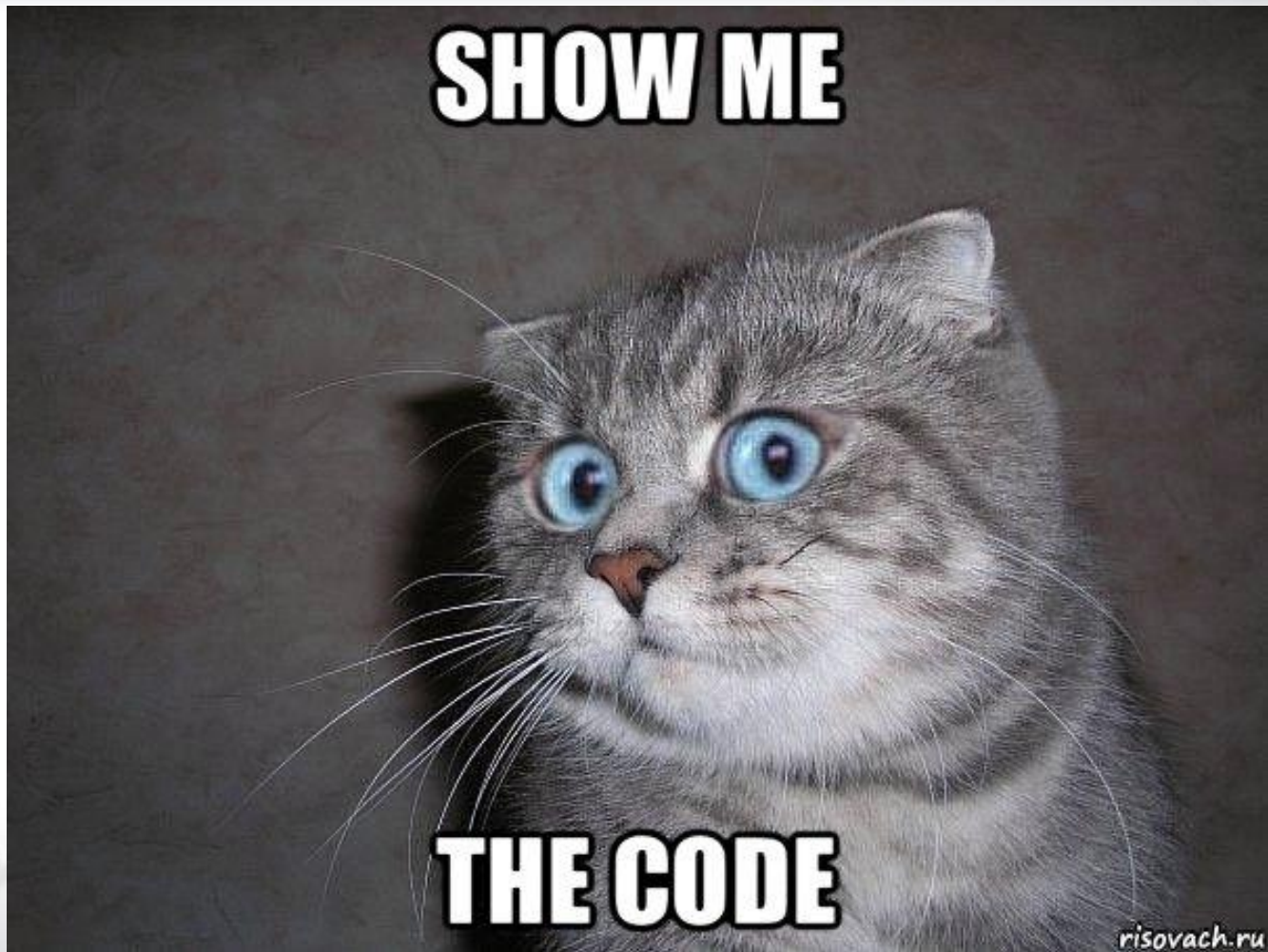
Using a project initialization service to create a Spring Boot project

- When you click the Generate button, the browser downloads a zip archive containing a Spring Boot project.





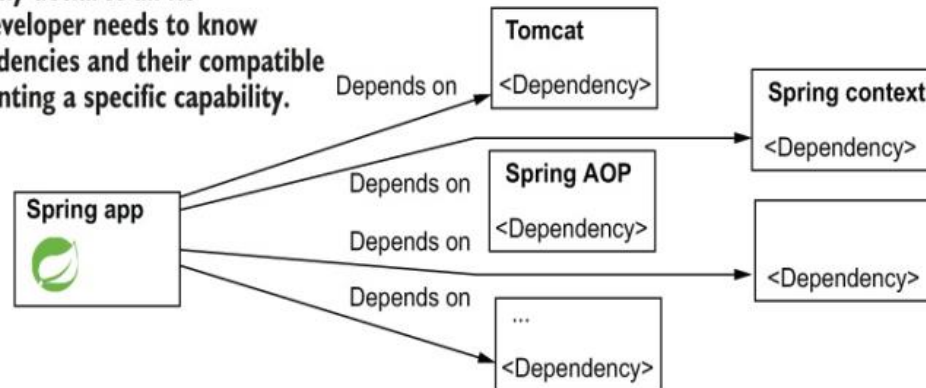
Demo



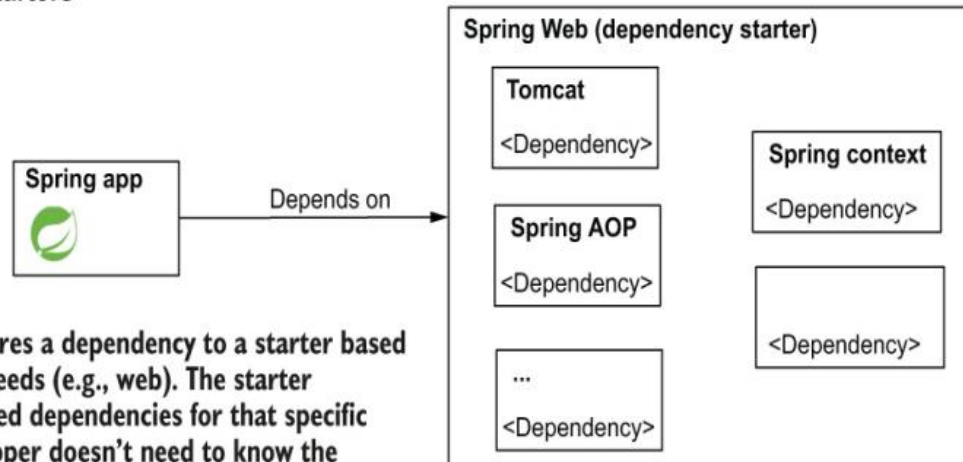
Dependency starters

Without dependency starters

The application directly declares all its dependencies. The developer needs to know all the needed dependencies and their compatible versions for implementing a specific capability.



With dependency starters



The application declares a dependency to a starter based on the capability it needs (e.g., web). The starter contains all the needed dependencies for that specific capability. The developer doesn't need to know the exact dependencies it needs nor their compatible versions.

Implementing a web app with Spring MVC

To add a web page to your app, you follow two steps:

1. Write an HTML document with the content you want to be displayed by the browser.
2. Write a controller with an action for the web page created at point 1



1. Write the HTML document.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Home Page</title>
</head>
<body>
  <h1>Welcome!</h1>
</body>
</html>
```

2. Write a controller with an action for the HTML document to display.

```
@Controller
public class MainController {

  @RequestMapping("/home")
  public String home() {
    return "home.html";
  }
}
```

Example

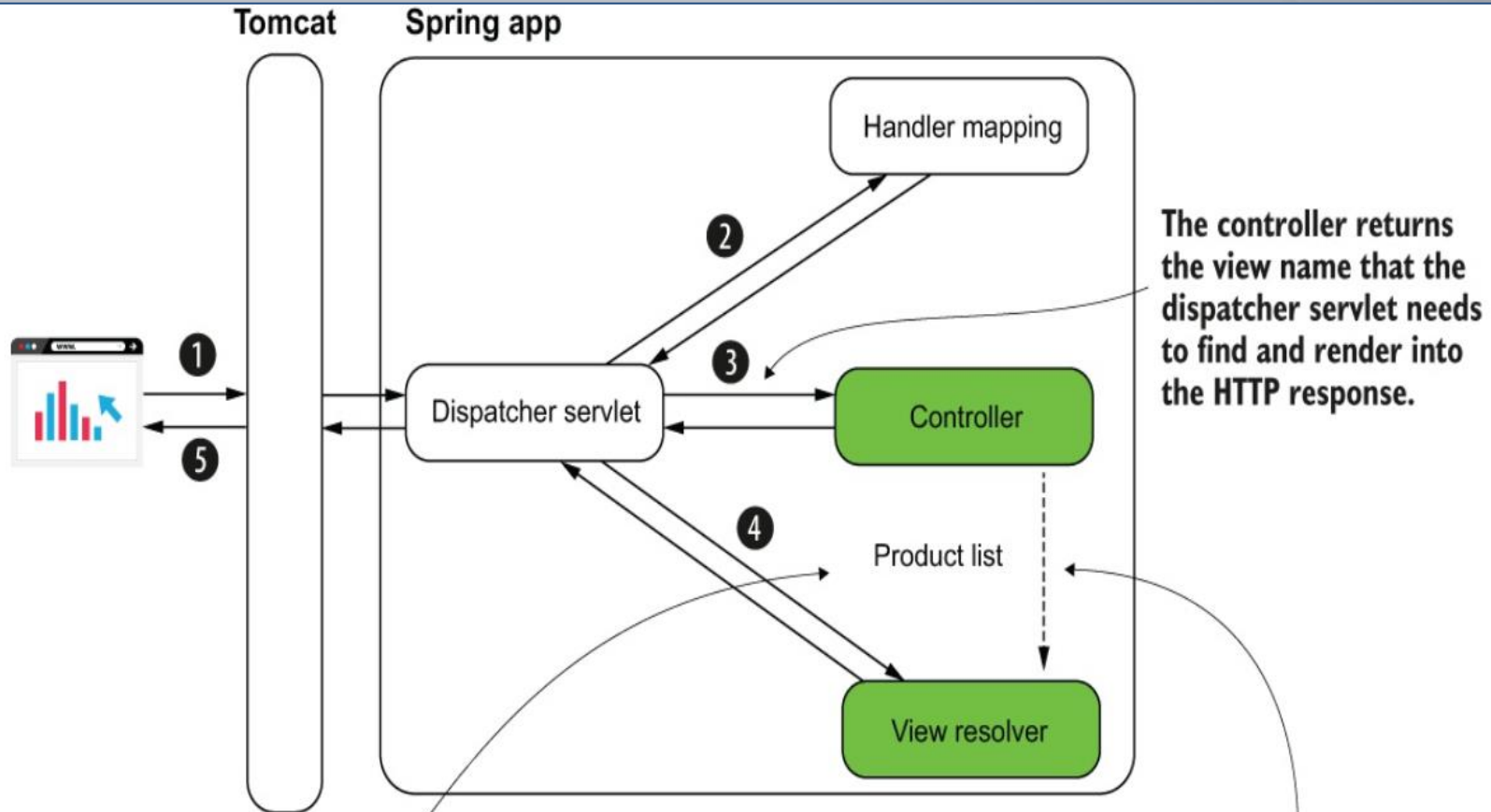


Implementing web apps with a dynamic view

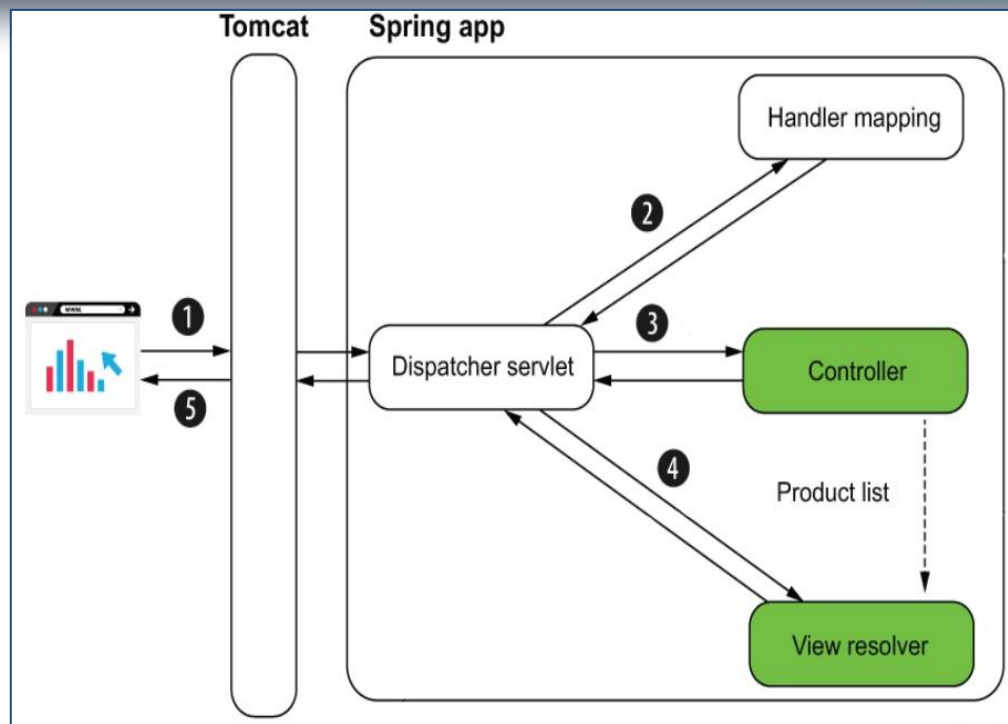
```
@Controller 1  
public class MainController {  
  
    @RequestMapping("/home") 2  
    public String home(Model model) { 3  
        model.addAttribute("username", "Student"); 4  
        model.addAttribute("color", "red"); 4  
        return "home"; 5  
    }  
}
```

- 1 The @Controller stereotype annotation marks this class as Spring MVC controller and adds a bean of this type to the Spring context.
- 2 We assign the controller's action to an HTTP request path.
- 3 The action method defines a parameter of type Model that stores the data the controller sends to the view.
- 4 We add the data we want the controller to send to the view.
- 5 The controller's action returns the view to be rendered into the HTTP response

Implementing web apps with a dynamic view



Implementing web apps with a dynamic view



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.

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

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

❶ Defines the Thymeleaf “th” prefix

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

Getting data on the HTTP request

In most cases, to send data through the HTTP request you use one of the following ways:

- An HTTP request parameter represents a simple way to send values from client to server in a key-value(s) pair format. To send HTTP request parameters, you append them to the URI in a request query expression. They are also called query parameters. You should use this approach only for sending a small quantity of data.
- An HTTP request header is similar to the request parameters in that the request headers are sent through the HTTP header. The big difference is that they don't appear in the URI, but you still cannot send large quantities of data using HTTP headers.
- A path variable sends data through the request path itself. It is the same as for the request parameter approach: you use a path variable to send a small quantity of data. But we should use path variables when the value you send is mandatory.
- The HTTP request body is mainly used to send a larger quantity of data (formatted as a string, but sometimes even binary data such as a file).

Using request parameters to send data from client to server

You use request parameters in the following scenarios:

- The quantity of data you send is not large. You set the request parameters using query variables (as shown in next example). This approach limits you to about 2,000 characters.
- You need to send optional data. A request parameter is a clean way to deal with a value the client might not send. The server can expect to not get a value for specific request parameters



Getting a value through a request parameter

```
@Controller
public class MainController {

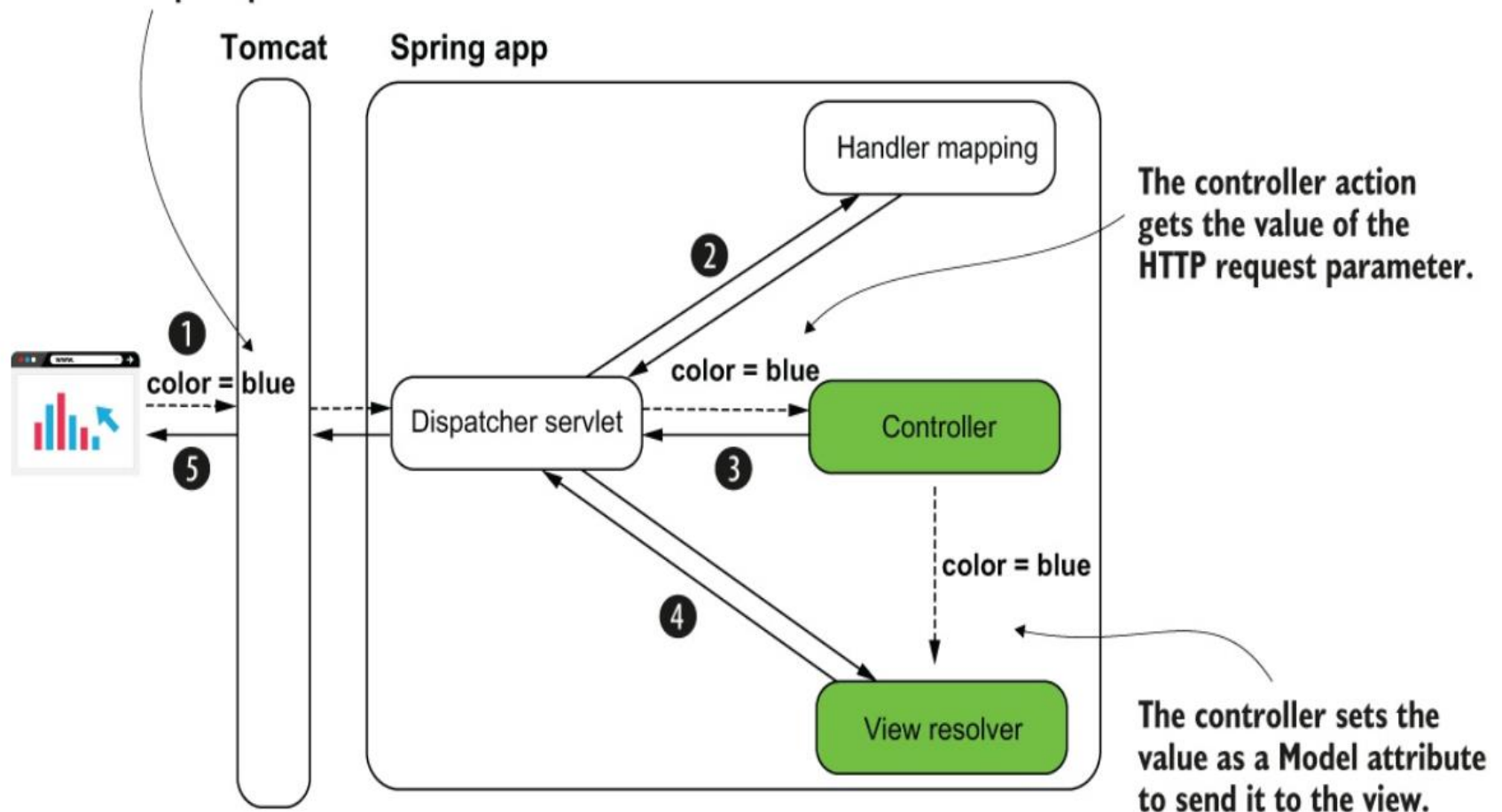
    @RequestMapping("/home")
    public String home(
        @RequestParam String color, ❶
        Model model) { ❷

        model.addAttribute("username", "Student");
        model.addAttribute("color", color); ❸
        return "home";
    }
}
```

- ❶ We define a new parameter for the controller's action method and annotate it with `@RequestParam`.
- ❷ We also add the `Model` parameter that we use to send data from the controller to the view.
- ❸ The controller passes the `color` sent by the client to the view

Getting a value through a request parameter

The client sends the color through an HTTP request parameter.



Getting a value through a request parameter

Run the application and access the `/home` path.

To set the request parameter's value, you need to use the next snippet's syntax:

```
http://localhost:8080/home?color=blue
```

When setting HTTP request parameters, you extend the path with a `?` symbol followed by pairs of `key=value` parameters separated by the `&` symbol.

For example, to send also the name as a request parameter:

```
http://localhost:8080/home?color=blue&name=Jane
```

Getting a value through a request parameter

```
@Controller
public class MainController {

    @RequestMapping("/home")
    public String home(
        @RequestParam String name,
        @RequestParam String color,
        Model model) {
        model.addAttribute("username", name);
        model.addAttribute("color", color);
        return "home";
    }
}
```

1

2

- 1 Gets the new request parameter “name”
- 2 Sends the “name” parameter’s value to the view

The request parameter query starts here.

Each parameter is given as a key-value pair.

`http://localhost:8080/home?color=blue&username=Student`

The key-value pairs are separated by an "&" symbol.

Using path variables to send data from client to server

- Using path variables is also a way of sending data from client to server.
- But instead of using the HTTP request parameters, you directly set variable values in the path, as presented in the next snippets

Using request parameters:

```
http://localhost:8080/home?color=blue
```

Using path variables:

```
http://localhost:8080/home/blue
```

- *You don't identify the value with a key anymore.*
- *You just take that value from a precise position in the path.*
- *On the server side, you extract that value from the path from the specific position.*
- *You may have more than one value provided as a path variable, but it's generally better to avoid using more than a couple.*

A quick comparison of the request parameters and path variables

Request parameters	Path variables
<ol style="list-style-type: none">1. Can 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.3. Some developers consider the query expression more difficult to read than the path expression.	<ol style="list-style-type: none">1. Should not be used with optional values.2. Always avoid sending more than three path variables. It's even better if you keep a maximum of two.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 path variables to send data from client to server

```
@Controller
public class MainController {

    @RequestMapping("/home/{color}")
    public String home(
        @PathVariable String color,
        Model model) {
        model.addAttribute("username", "Student");
        model.addAttribute("color", color);
        return "home";
    }
}
```

1

2

- 1 To define a path variable, you assign it a name and put it in the path between curly braces.
- 2 You mark the parameter where you 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 send data from client to server

Run the app and access the page in your browser with different values for the color.

```
http://localhost:8080/home/blue  
http://localhost:8080/home/red  
http://localhost:8080/home/green
```

Using the GET and POST HTTP methods

```
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Student {
    private String name;
    private int age;
}
```

```
@Service
public class StudentService {
    private List<Student> students = new ArrayList<>();
    public void addStudent(Student student) {
        students.add(student);
    }
    public List<Student> findAll() {
        return students;
    }
}
```

Using the GET and POST HTTP methods

Observe the two use cases described by the scenario.

The user needs to do the following:

- View all students in the list; here, we'll use HTTP GET.
- Add student to the list; here, we'll use HTTP POST

Using the GET and POST HTTP methods

```
@Controller
@AllArgsConstructor
public class StudentController {

    private final StudentService studentService;

    @GetMapping("/students")
    public String viewStudents(Model model) {
        var students = studentService.findAll();
        model.addAttribute("students", students);
        return "students";
    }

    @PostMapping("/add_student")
    public String addStudent(
        @RequestParam String name,
        @RequestParam int age) {
        studentService.addStudent(new Student(name, age));
        return "redirect:/students";
    }
}
```



НАЦІОНАЛЬНИЙ
УНІВЕРСИТЕТ
КОРАБЛЕБУДУВАННЯ
ІМЕНІ АДМІРАЛА МАКАРОВА

Example

