Spring Introduction MVC

Spring Fundamentals

SoftUni Team

Technical Trainers

Software University

https://softuni.bg

Questions

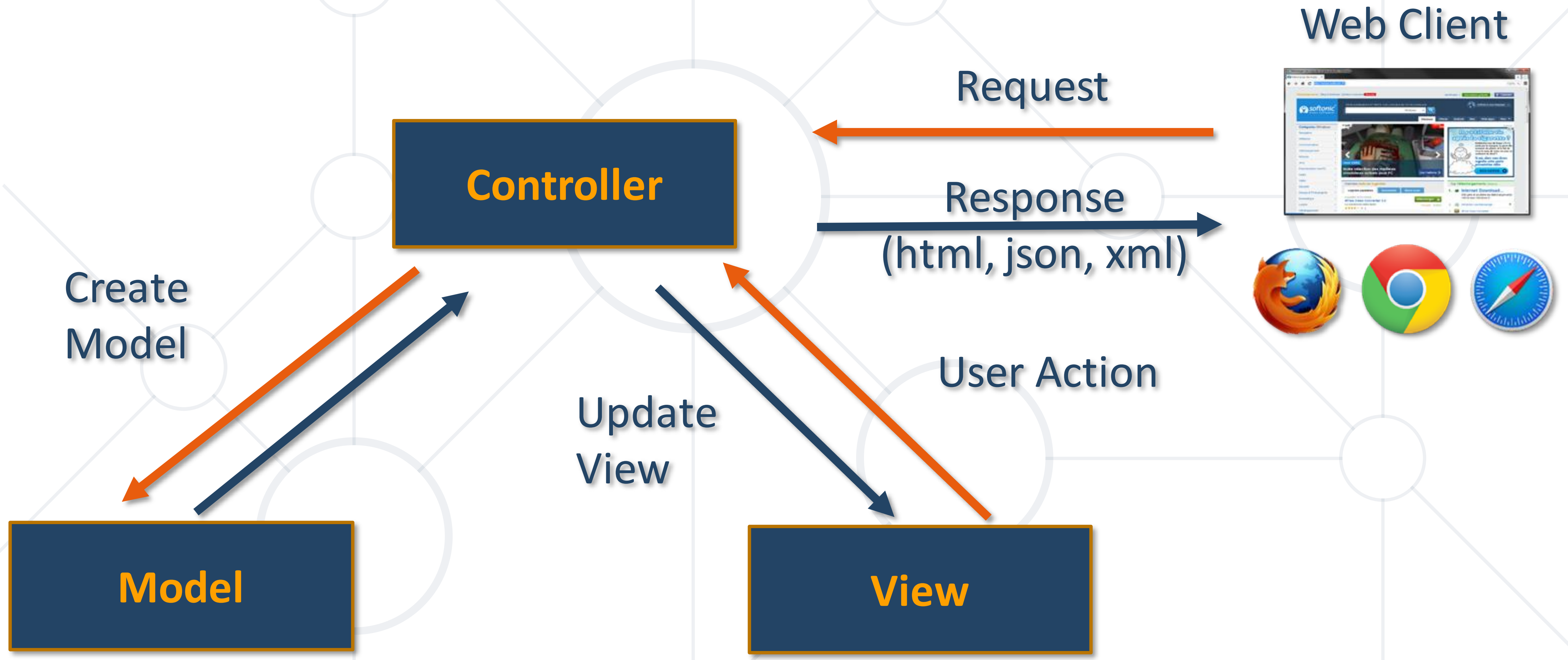
sli.do

#java-web

Table of Content

MVC Introduction

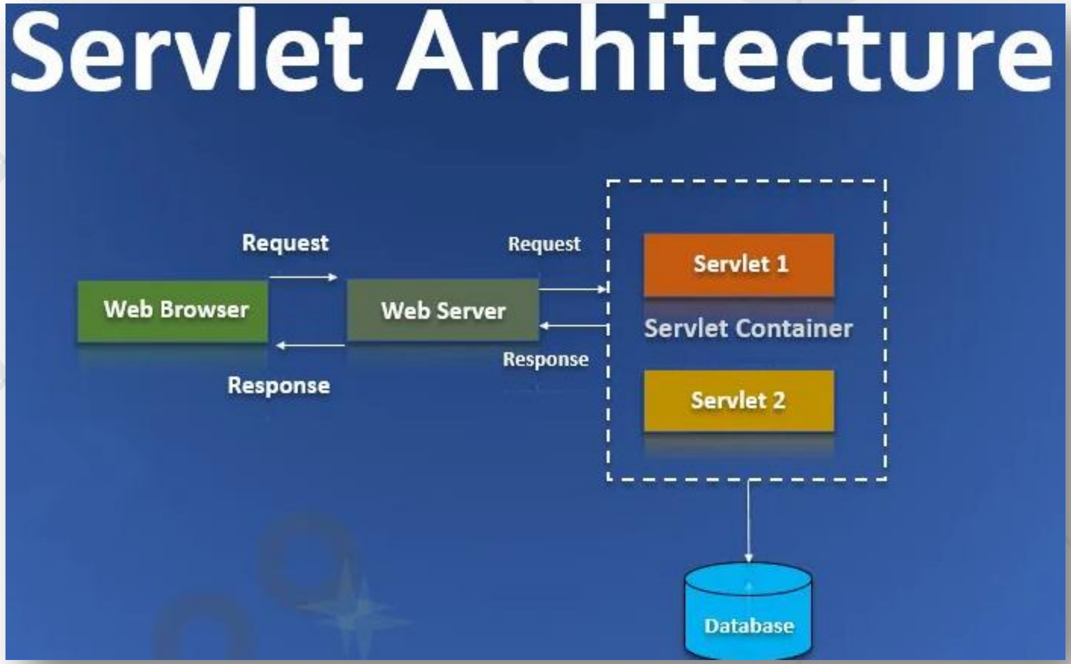
MVC – Control Flow



Servlets

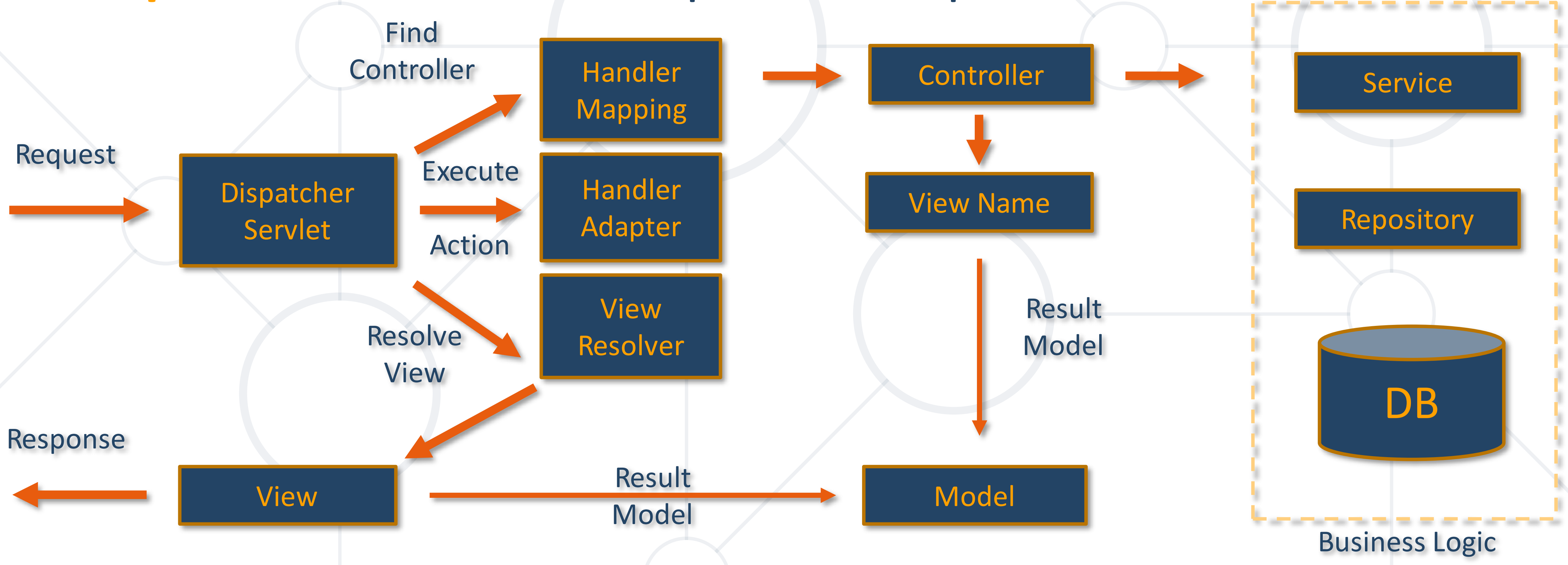
Servlets

- **What is Servlet**
  - A **Servlet** is a Java **class** that **extends** the capabilities of a server, allowing it to handle requests and generate responses for web applications
  - Servlets operate on the **server-side** to process client requests, interact with **databases**, and generate dynamic content



Spring MVC and DispatcherServlet

- **Model-view-controller (MVC)** framework is designed around a **DispatcherServlet** that dispatches requests to handlers

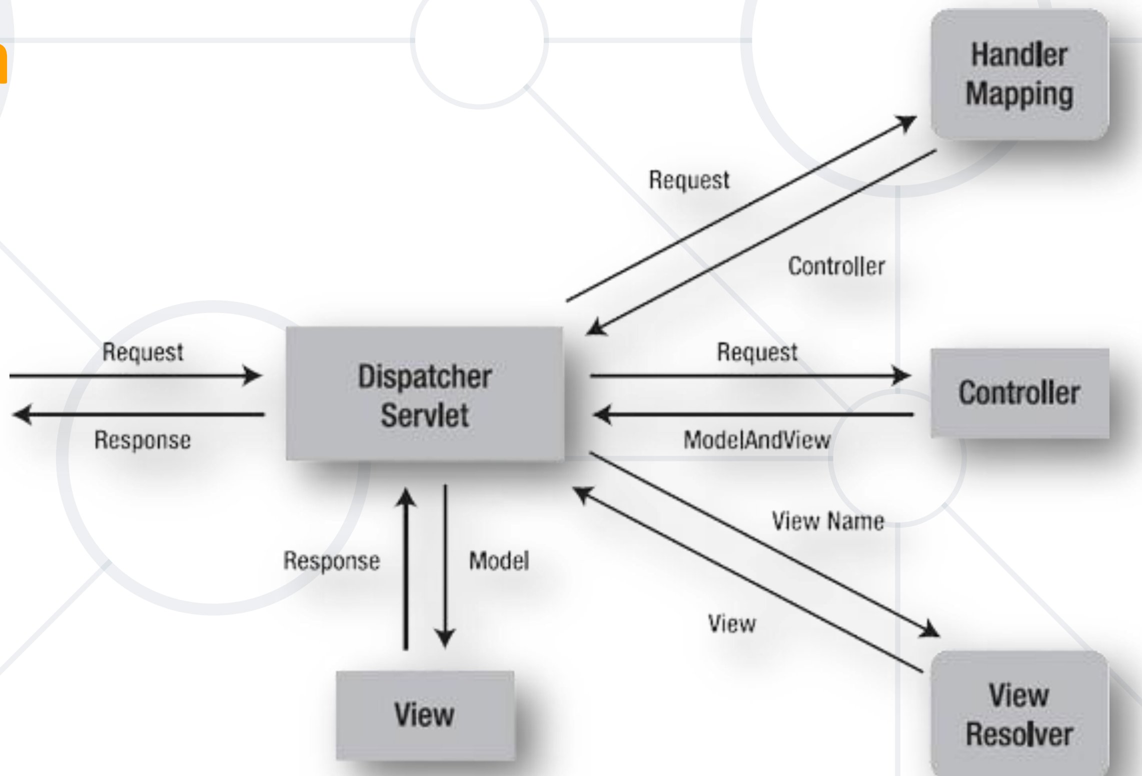


DispatcherServlet

- **DispatcherServlet**
  - The DispatcherServlet is a **Servlet** provided by Spring MVC that acts as the front **controller** for handling **web** requests
  - The **DispatcherServlet** coordinates the entire **request processing lifecycle**, including request parsing, dispatching, executing controller logic, rendering views, and sending responses back to the client

DispatcherServlet and Controllers

- **Controllers** receive requests from the **DispatcherServlet** and perform **business logic** to process them
- In Spring MVC, controllers are **Java classes** responsible for processing incoming requests and generating responses
- Controllers often interact with other components such as **services**, **repositories** or **models** to fulfil the request



DispatcherServlet Explained

- **Request Handling:** DispatcherServlet **receives** all incoming requests from clients and delegates them to the appropriate handlers for processing
- **Mapping and Dispatching:** The DispatcherServlet **maps** incoming requests to specific controller methods based on the request URL and other criteria defined in the application's request mappings

DispatcherServlet Explained

- **Handler Execution:** Once a request is mapped to a controller method, the **DispatcherServlet** executes the corresponding handler method
  - This method typically performs **business logic**, accesses data, and prepares the model and view for rendering the response
- **View Resolution:** After the handler method has processed the request and populated the model, the DispatcherServlet **resolves** the **view name** returned by the handler to an actual view implementation

DispatcherServlet Explained

- **Rendering Response:** Once the view has been resolved, the DispatcherServlet invokes the **view's rendering process** to generate the response content
  - The rendered output is then **sent back** to the client as the HTTP response
- **Exception Handling:** The DispatcherServlet also handles **exceptions** thrown during request processing
  - It can route exceptions to custom **error handling** mechanisms, such as global exception handlers or specific error pages

Custom Servlets

- **Custom Servlets**
  - While Spring MVC primarily uses the **DispatcherServlet** for request handling, developers can still create **custom** Servlets when needed
  - Custom Servlets can be registered alongside Spring MVC's **DispatcherServlet** in a web application's deployment descriptor (web.xml) or through Spring's **Java-based** configuration
  - Custom Servlets can perform **specialized** tasks such as servlet-specific filtering, authentication, or handling specific types of requests

Spring Controllers

Spring Controllers

- Defined with the **@Controller** annotation

```
@Controller
public class HomeController {
   ...
}
```

- Controllers can contain multiple actions on different routes

Request Mapping Method-level

- Annotated with **@RequestMapping(…)**

```
@RequestMapping("/home")
public String home(Model model) {
    model.addAttribute("message", “Welcome!");
    return "home-view";
}
```

- Or

```
@RequestMapping("/home")
public ModelAndView home(ModelAndView mav) {
    mav.addObject("message", “Welcome!");
    mav.setViewName("home-view");
    return mav;
}
```

Request Mapping Class-level

- Annotated with **@RequestMapping(…)**

```
@RequestMapping("/home")
public class HomeController {
…
}
```

- Combined

```
@RequestMapping("/home")
public class HomeController {

   @RequestMapping("/menu")
   public String getMenu() {
        model.addAttribute("message", "Welcome to menu!");
    return "home-view";
}
```

Request Mapping

- Problem when using **@RequestMapping** is that it accepts all types of request methods (get, post, put, delete, head, patch)

```
@RequestMapping(value="/home", method=RequestMethod.GET)
public String home() {
   return "home-view";
}
```

Get Mapping

- Easier way to create route for a GET request

```
@GetMapping("/home")
public String home() {
   return "home-view";
}
```

- This is alias for **RequestMapping** with method GET

Actions – Get Requests

CatController.java

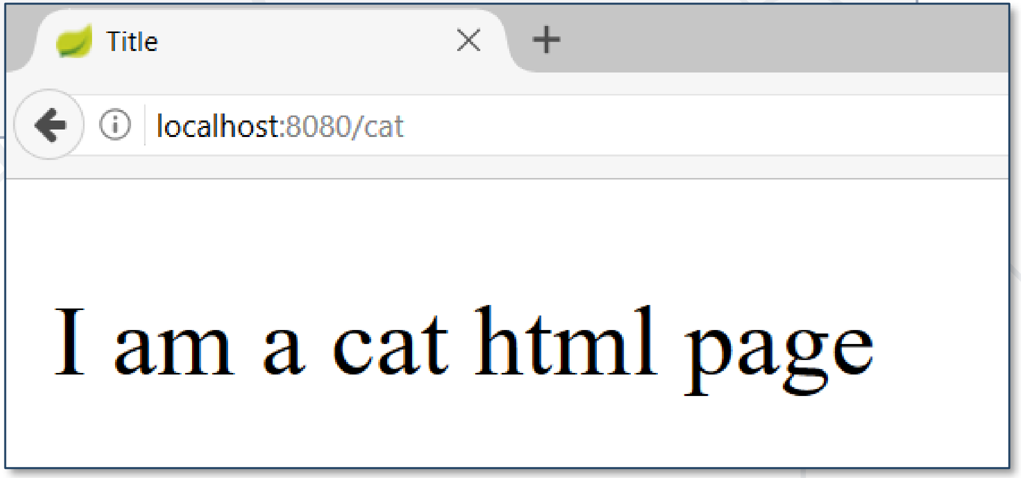
```
@Controller
public class CatController {

    @GetMapping("/cat")
    public String getHomeCatPage(){
        return "cat-page.html";
    }
}
```

Controller

Request Mapping

Action

View



I am a cat html page

Controllers

DogController.java

```
@Controller
public class DogController {

    @GetMapping("/dog")
    @ResponseBody
    public Dog getDogHomePage(){
      Dog dog = dogService.getBestDog();
      return dog;
    }
}
```

Controller

Request Mapping

Action

Post Mapping

- Similar to the **GetMapping** there is also an alias for **RequestMapping** with method POST

```
@PostMapping("/register")
public String register(UserDTO userDto) {
    …
}
```

- If we use **@RequestBody** Spring Boot will expect the incoming data to be in a JSON or XML format, and it will automatically deserialize the request body into the UserDTO object:

```
@PostMapping("/register")
public String register(@RequestBody UserDTO userDto) {
    …
}
```

- Similar annotations exist for all other types of request methods

What is a HTML Form?

- **HTML Form**
  - A **HTML Form** is a crucial component of web development used to collect user **input** and submit it to a server for processing
  - It's created using the **<form>** element in HTML and typically consists of various **input fields**, such as text fields, checkboxes, radio buttons, dropdown lists, and buttons, allowing users to input data

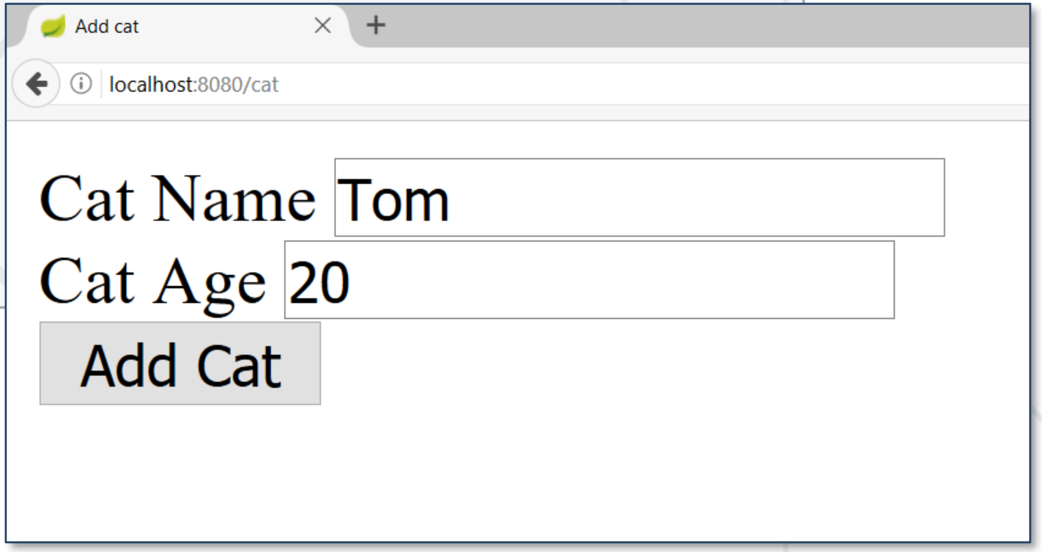
<form>
Name:
Email:
Phone:
Submit

Actions – Post Requests

CatController.java

```
@Controller
@RequestMapping("/cat")
public class CatController {

    @PostMapping("/new")
    public String addCat(){
        return "new-cat.html";
    }
}
```

Starting route



Cat Name Tom
Cat Age 20

Actions – Post Requests

CatController.java

```
@Controller
@RequestMapping("/cat")
public class CatController {

    @PostMapping
    public String addCatConfirm(@RequestParam String catName,
@RequestParam int catAge){
        System.out.println(String.format(
            "Cat Name: %s, Cat Age: %d", catName, catAge));
        return "redirect:/cat";
    }
}
```

Request param

Redirect

```
Cat Name: Tom, Cat Age: 20
```

Passing Attributes to View

- Passing a **String** to the view

```java
@GetMapping("/")
public String welcome(Model model) {
  model.addAttribute("name", "Peter");
  return "index";
}
```

- The **Model** object will be automatically passed to the view as context variables
- Attributes can be accessed from Thymeleaf

Passing Attributes to View

- Passing a **ModelMap** object to the view

```
@GetMapping("/")
public String welcome(ModelMap modelMap) {
  modelMap.addAttribute("name", "Peter");
  return "index";
}
```

- The **ModelMap** object will be automatically passed to the view as context variables
- Attributes can be accessed from Thymeleaf

Passing Attributes to View

- Passing a **ModelAndView** object to the view

```
@GetMapping("/")
public ModelAndView welcome(ModelAndView model) {
  model.addObject("name", "Peter");
  model.setViewName("index")
  return model;
}
```

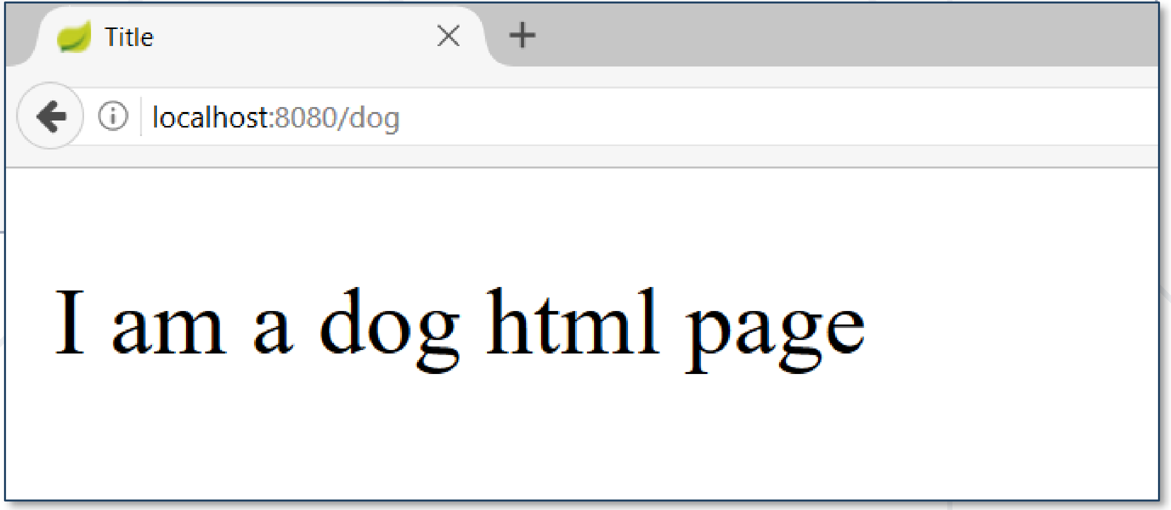
- The **ModelAndView** object will be automatically passed to the view as context variables
- Attributes can be accessed from Thymeleaf

Models and Views

DogController.java

```
@Controller
public class DogController {

    @GetMapping("/dog")
    public ModelAndView getDogHomePage(ModelAndView modelAndView){
        modelAndView.setViewName("dog-page.html");
        return modelAndView;
    }
}
```

Model and View



I am a dog html page

Request Parameters

- Getting a parameter from the query string

```
@GetMapping("/details")
public String details(@RequestParam("id") Long id) {
  ...
}
```

- **@RequestParam** can also be used to get POST parameters

```
@PostMapping("/register")
public String register(@RequestParam("name") String name) {
  ...
}
```

Request Parameters with Default Value

- Getting a parameter from the query string

```
@GetMapping("/comment")
public String comment(@RequestParam(name="author",
defaultValue = "Annonymous") String author) {
  ...
}
```

- Making parameter optional

```
@GetMapping("/search")
public String search(@RequestParam(name="sort",
required = false) String sort) {
  ...
}
```

Path Variable

- Getting a parameter from the path variable:

```
@GetMapping("/details/{id}")
public String details(@PathVariable("id") Long id) {
  ...
}
```

Form Objects

- Spring will **automatically** try to fill objects with a form data

```
@PostMapping("/register")
public String register(UserDTO userDto) {
  ...
}
```

- The input field names **must** be the same as the **object** field names

Redirecting

- Redirecting after **POST** request

```
@PostMapping("/register")
public String register(UserDTO userDto) {
  ...
  return "redirect:/login";
}
```

Redirecting with Parameters

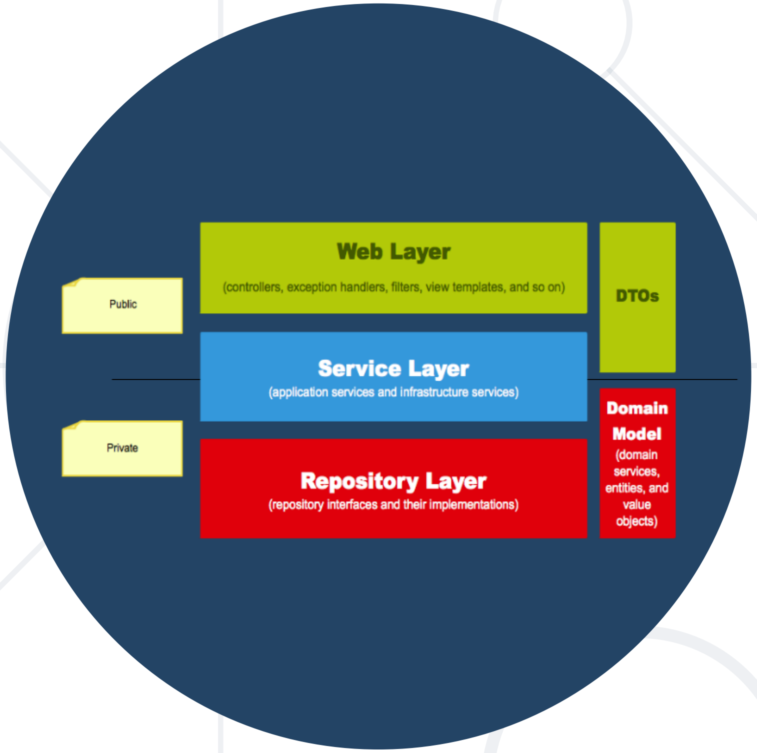
- Redirecting with **query** string parameters

```
@PostMapping("/register")
public String register(UserDTO userDto,
RedirectAttributes redirectAttributes) {

  redirectAttributes.addAttribute("errorId", 3);
  return "redirect:/login";
}
```

Redirecting with Attributes

- Keeping objects after **redirect**

```
@PostMapping("/register")
public String register(@ModelAttribute UserDTO userDto,
RedirectAttributes redirectAttributes) {
  ...
  redirectAttributes.addFlashAttribute("userDto", userDto);
  return "redirect:/register";
}
```

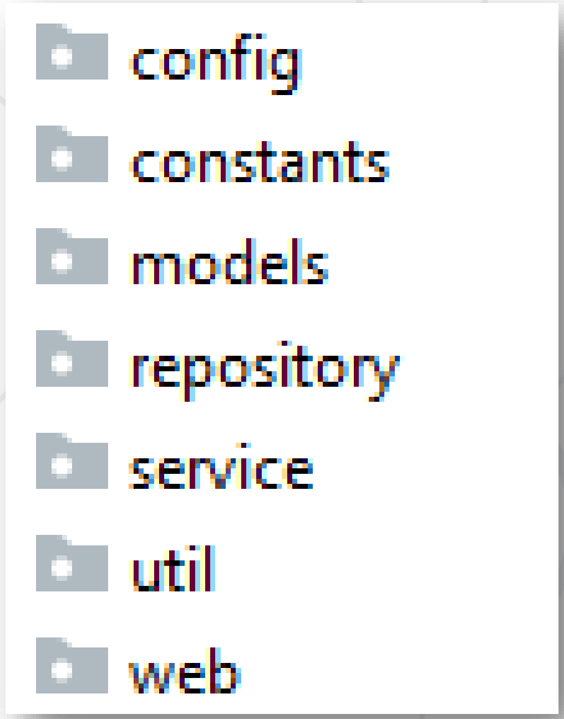

Public
Private
Web Layer
(controllers, exception handlers, filters, view templates, and so on)
DTOs
Service Layer
(application services and infrastructure services)
Domain Model
(domain services, entities, and value objects)
Repository Layer
(repository interfaces and their implementations)

Layers

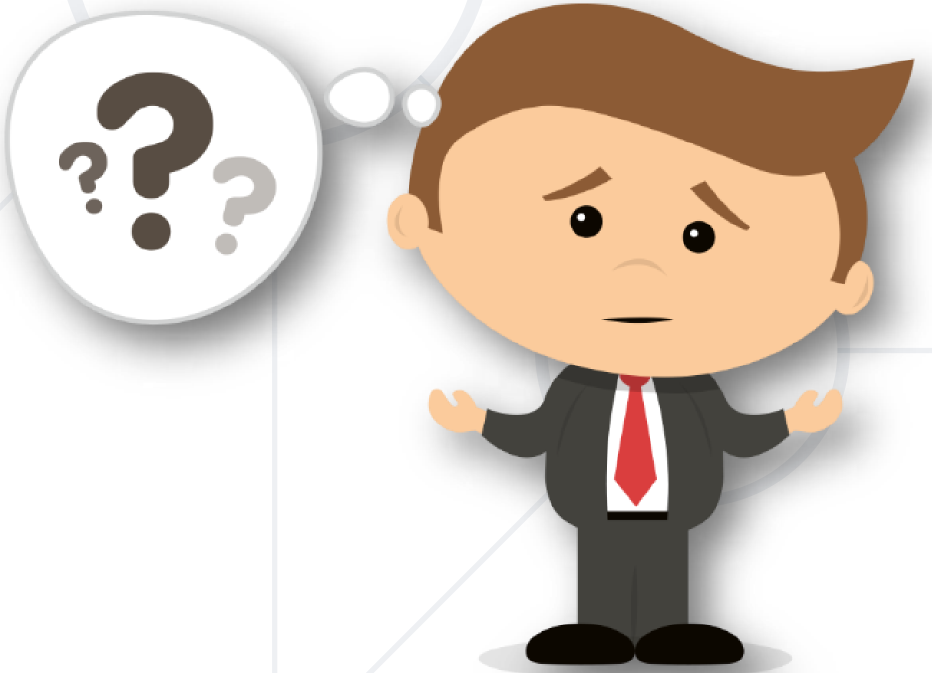
The Correct Project Structure

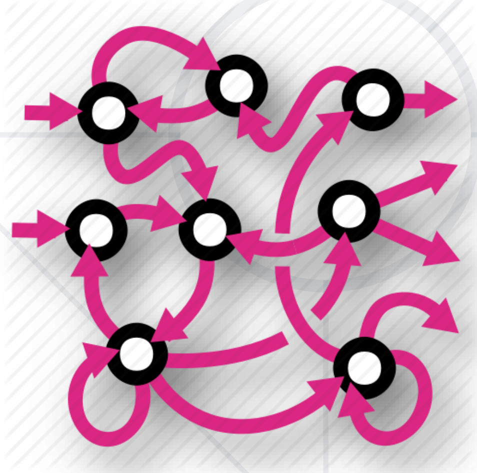
Layers

- We are used to **splitting** our code based **on its functionality**:
- It gets **hard to navigate** in bigger applications



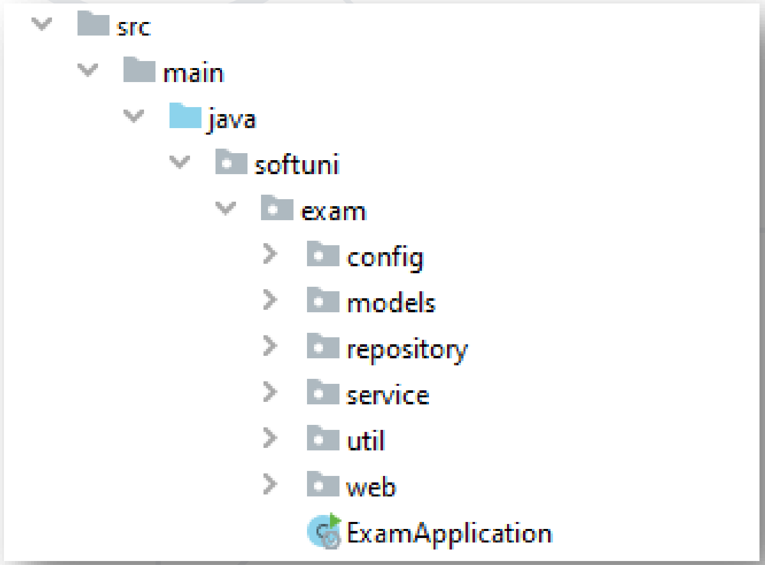
- config
- constants
- models
- repository
- service
- util
- web





Layers

- **Splitting** the project into **different modules**
  - Each module corresponding to the application layer
  - Makes it easier to navigate



```
src
  main
    java
      softuni
        exam
          config
          models
          repository
          service
          util
          web
          ExamApplication
```

Layers

- **Presentation Layer (Controller)**
  - The presentation layer, also known as the **controller** layer, is responsible for handling incoming **HTTP** requests, processing them, and generating appropriate responses
  - Controllers receive requests from clients, execute **business** logic (or delegate it to service layer components) and return responses
  - Controllers are annotated with **@Controller** and handle requests based on mappings defined with annotations like **@RequestMapping**

Layers

- **Service Layer**
  - The service layer contains business logic and acts as an **intermediary** between the **presentation** layer (controllers) and the **data access** layer (repositories)
  - Services **encapsulate** reusable business logic, such as processing data performing calculations, or coordinating interactions between multiple components
  - Services are typically annotated with **@Service** to be identified as Spring-managed components and are injected into controllers or other services

Layers

- **Data Access Layer (Repository)**
  - The data access layer is responsible for **interacting** with databases or other external data sources to perform **CRUD** (Create, Read, Update, Delete) operations
  - Repositories, also known as **data access objects** (DAOs), abstract the underlying data access mechanisms and provide a consistent interface for data manipulation
  - Repositories are annotated with **@Repository** to indicate that they handle data access operations and are managed by the Spring container

Layers

- **Model Layer**
  - The model layer represents the domain model or business objects that **encapsulate** data and **behavior** relevant to the application's domain
  - Models may also include **DTOs** (Data Transfer Objects) to **transfer** data between layers or to external systems

Layers

- **View Layer**
  - The view layer is responsible for **rendering** the user interface and **presenting** data to the client
  - Views are typically **HTML** templates, **JSP** files, **Thymeleaf** templates or other view technologies supported by Spring MVC
  - Views receive model data from **controllers** and generate HTML content that is sent back to the client's browser for display

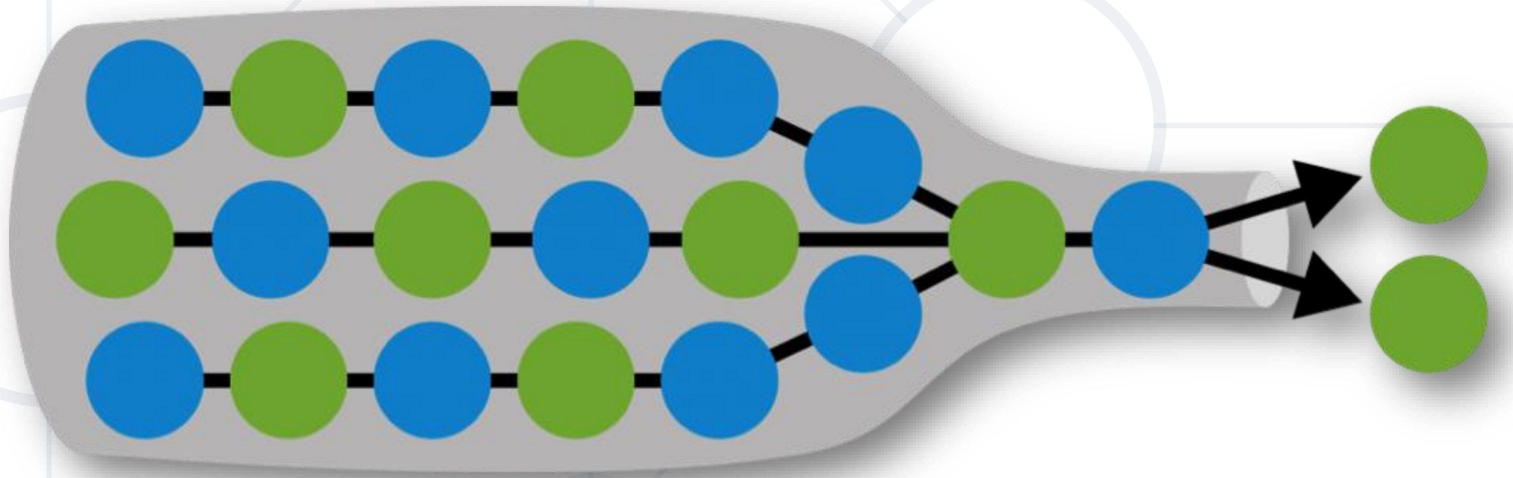
Layers

- **Integration Layer (Optional)**
  - In some cases, an integration layer may be introduced to interact with **external** services, **APIs**, or **messaging** systems
  - Integration components handle communication with external systems and manage **data exchange** using protocols such as RESTful APIs, SOAP, or messaging queues
  - Integration components can be implemented as Spring-managed **beans** and injected into **services** or **controllers** as needed

Spring MVC vs. Reactive Programming

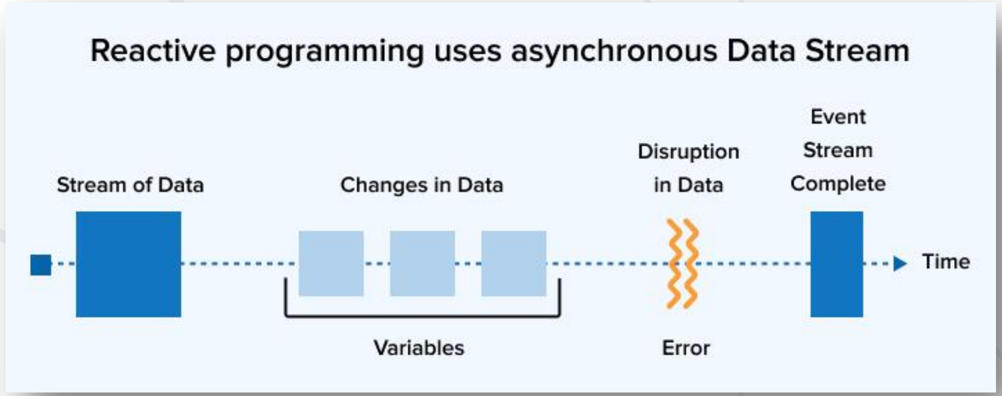
What is a "Bottleneck"?

- **"Bottleneck"** refers to a point in a system where the **flow** of data or processing is constrained or limited, thereby reducing the overall **performance** or **throughput** of the system



The Solution To The "Bottleneck" Problem

- To solving the **Bottleneck** issue in MVC applications, you can use **reactive** programming to handle requests in a **non-blocking** manner
  - **Reactive programming** is a programming paradigm that focuses on asynchronous and non-blocking operations
  - **Asynchronous** programming is often useful to make systems quick and responsive when dealing with **large** amounts of data or **multiple** users



"Bottleneck" in Spring MVC

- In **traditional** MVC architecture, each request is typically handled by a **single thread**
- If a request involves **blocking** operations, such as waiting for a database query to complete, the thread becomes **blocked**, and it **cannot** handle other requests until the blocking operation finishes
- This can lead to **"Bottlenecks"**, especially in scenarios where **multiple** parallel requests are made to the database

"Bottleneck" in Spring MVC

- If the number of concurrent requests **exceeds** the available **threads** in the server's thread pool, requests will start queuing up, leading to a **Bottleneck**:

```
@RestController
public class UserController {

    @Autowired
    private UserRepository userRepository;

    @GetMapping("/users")
    public List<User> getUsers() {
        // Perform a blocking database query to fetch all users
        return userRepository.findAll();
    }
}
```

"Bottleneck" in Spring MVC

- The **getUsers** method returns a **CompletableFuture<List<User>>** indicating that it will provide the result **asynchronously**

```
@RestController
public class UserController {

    @Autowired
    private UserRepository userRepository;

    @GetMapping("/users")
    public CompletableFuture<List<User>> getUsers() {
        return getUsersAsync();
    }
. . .
```

"Bottleneck" in Spring MVC

- The **getUsersAsync** is annotated with **@Async**, indicating that it should be executed **asynchronously** in a **separate** thread from the thread **pool** managed by Spring
- When the database query completes, we complete the **CompletableFuture** with the list of users and return it

```java
. . .
    @Async
    public CompletableFuture<List<User>> getUsersAsync() {
        // Perform the database query asynchronously
        return CompletableFuture.supplyAsync(() -> userRepository.findAll());
    }
}
```

What Is Project Reactor

- **Project Reactor:** a reactive programming library for building **asynchronous** and **event-driven** applications in Java. It provides two fundamental types (classes) – **Flux** and **Mono**

Flux Class	Mono Class
a reactive stream that can emit **zero** to **many** items asynchronously	a reactive stream can emit **zero** or **one** item asynchronously
produces **multiple** values over time	representing a **single** result or value that m ay be available at some point in the future
handles sequences of events or data streams that may have **multiple** elements	handling asynchronous computations or operations that **produce a single value**

Using R2DBC

- **R2DBC** stands for **Reactive Relational Database Connectivity**, a specification to integrate **SQL** databases using **reactive** drivers
- Spring Data R2DBC applies familiar Spring abstractions and **repository** support for R2DBC

```java
import org.springframework.data.r2dbc.repository.R2dbcRepository;
import reactor.core.publisher.Flux;

public interface AgentRepository extends R2dbcRepository<AgentRow, Integer> {

    Flux<AgentRow> findAllByCorporationId(int corpId);
    Flux<AgentRow> findAllByLocationId(int locationId);
}
```

Using R2DBC

```java
@Service
public class AgentService {

    @Autowired
    private AgentRepository repo;

    public Flux<AgentRow> getAll() {
        return repo.findAll();
    }

    public Flux<AgentRow> getForCorp(int corpId) {
        return repo.findAllByCorporationId(corpId);
    }
    . . .
```

Using R2DBC

```
. . .
public Flux<AgentRow> getForLocation(int locationId) {
        return repo.findAllByLocationId(locationId);
    }

    public Mono<AgentRow> getAgent(int agentId) {
        return repo.findById(agentId);
    }
}
```

Spring MVC vs. Reactive Programming

- **Programming Model**
  - **Spring MVC:** Follows a traditional synchronous, **blocking** programming model where each HTTP request is handled by a dedicated thread from a thread pool.
  - **Reactive Programming:** Follows a non-blocking, asynchronous programming model where requests are handled by a small number of event-loop threads. Instead of blocking threads, Reactive applications use asynchronous and **non-blocking I/O** operations to handle requests

Spring MVC vs. Reactive Programming

- **Concurrency Model**
  - **Spring MVC:** Relies on **multi-threading** to handle concurrent requests. Each request is typically processed by a dedicated thread, and **blocking** operations can lead to thread contention and resource wastage
  - **Reactive Programming:** Relies on a **single-threaded** event-loop model, where a small number of threads can handle a large number of concurrent connections efficiently. Reactive applications leverage **asynchronous** I/O operations and reactive programming constructs to handle concurrency without blocking threads

Summary

- What is Spring MVC
- Servlets
- Spring Controlers
- Layers
  - Project Structure
- Spring MVC vs. Reactive Programming

Software University

Questions?

SoftUni

Software University

SoftUni Creative

SoftUni Digital

SoftUni Foundation

SoftUni Kids

СофтУни БУДИТЕЛ

Finance Academy

SoftUni Diamond Partners

Software University

SmartIT

Coca-Cola HBC Bulgaria

SUPER HOSTING .BG

createX

Postbank
Решения за твоето утре

DRAFT KINGS

VIVACOM

AMBITIONED

INDEAVR
Serving the high achievers

SOFTWARE GROUP

PHAR VISION

Trainings @ Software University (SoftUni)

- Software University – High-Quality Education, Profession and Job for Software Developers
  - softuni.bg, about.softuni.bg
- Software University Foundation
  - softuni.foundation
- Software University @ Facebook
  - facebook.com/SoftwareUniversity

License