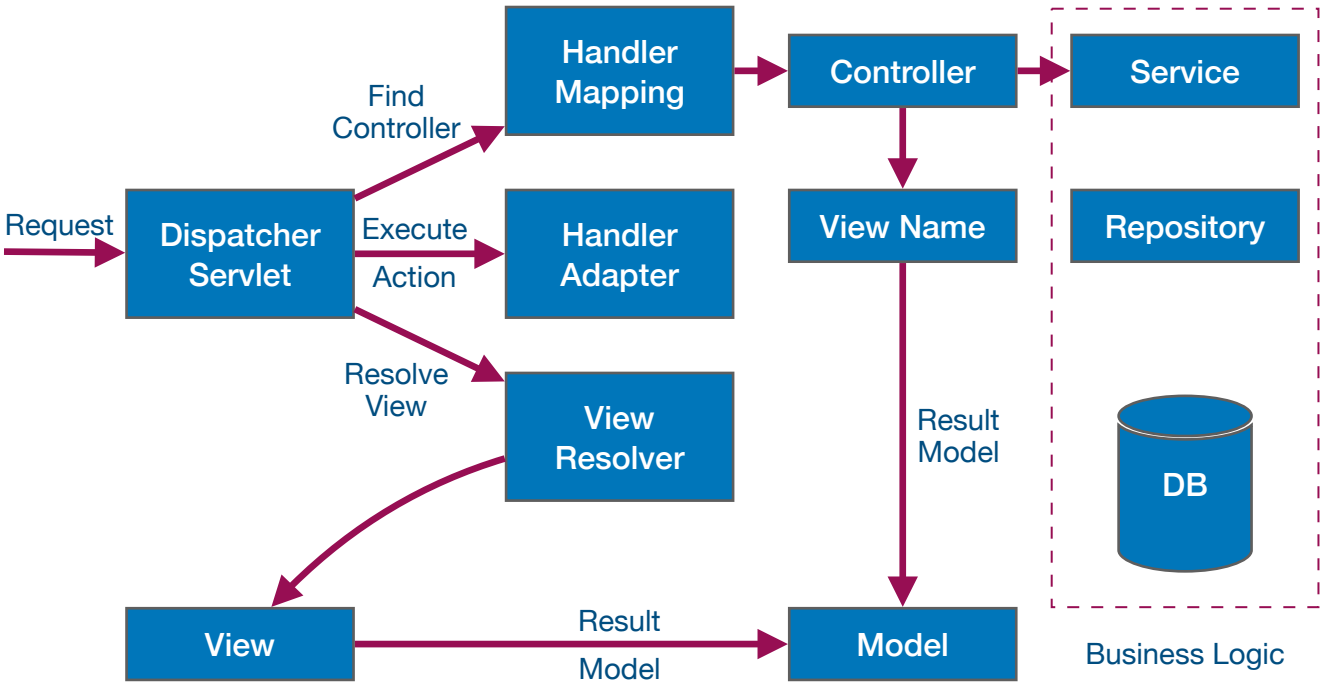
Spring Introduction MVC
Spring Fundamentals

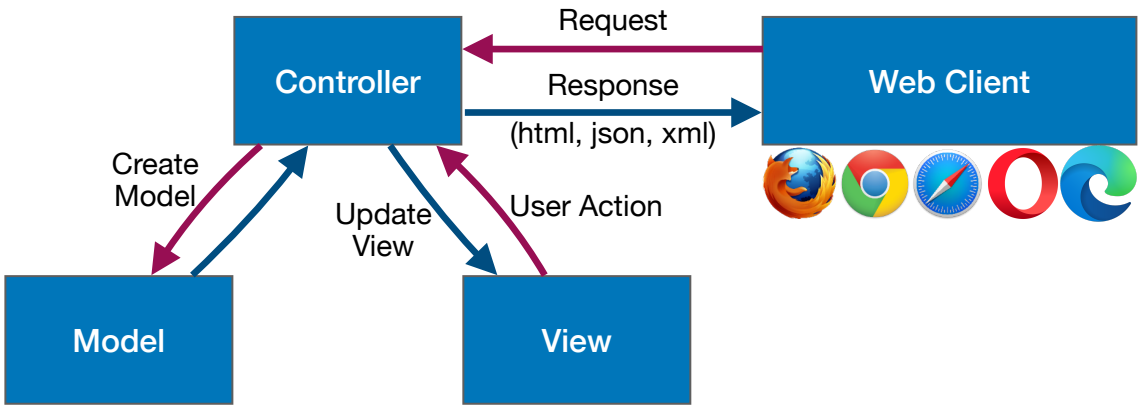
Table of Content

What is Spring MVC?

- **Model-View-Controller (MVC)** framework is designed around a **DispatcherServlet** that dispatches requests to handlers.



MVC – Control Flow



Spring Controllers
Annotations, IoC Container

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

- Combined

```
@RequestMapping("/home")
@Controller
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, PATCH, DELETE, HEAD, OPTIONS)
- Execute only on GET requests

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

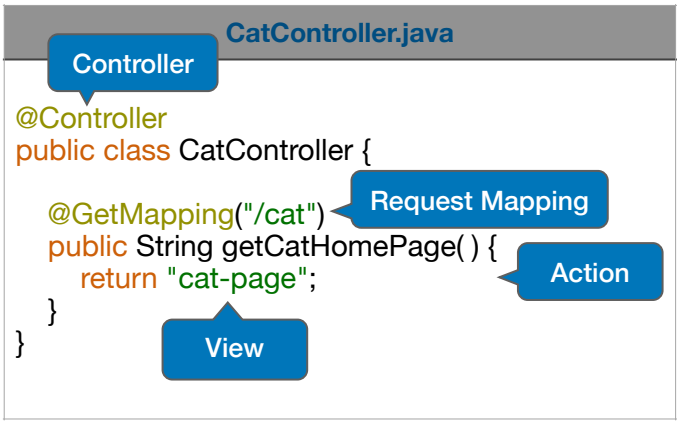
Get Mapping

- 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



Controllers

DogController.java

```
@Controller
public class DogController {

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

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

Actions – Post Requests

CatController.java

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

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

Starting route

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

Passing Attributes to View

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

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

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

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

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

Model and View

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

- The **Model**, **ModelMap** and **ModelAndView** objects will be automatically passed to the view as context variables
- Attributes can be accessed from Thymeleaf

Models and Views

DogController.java

Model and View

```java
@Controller
public class DogController {

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

Request Parameters

- Getting a parameter from the query string

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

- @RequestParam can also be used to get POST parameters

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

Request Parameters with Default Value

- Getting a parameter from the query string

```java
@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) {
    ...
}
```

From Objects

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

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

Redirecting with Parameters

- Redirecting with query string parameters

```
@PostMapping("/register")
public String register(UserDTO userDto, RedirectAttributes redirectAttributes) {
    redirectAttributes.addAttribute("errorOd", 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";
}
```

Inversion of Control
Constructor vs Field vs Setter Injection

Field Injection

- Easy to write
- Easy to add new dependencies
- It hides potential architectural problems!

```
@Autowired
private ServiceA serviceA
@Autowired
private ServiceB serviceB
@Autowired
private ServiceC serviceC
```

Constructor Injection

- Time Consuming
- Harder to add dependencies
- It shows potential architectural problems!

```
@Autowired
public ControllerA(ServiceA serviceA, ServiceB serviceB, ServiceC serviceC) {
    this.serviceA = serviceA;
    this.serviceB = serviceB;
    this.serviceC = serviceC;
}
```

Setter Injection

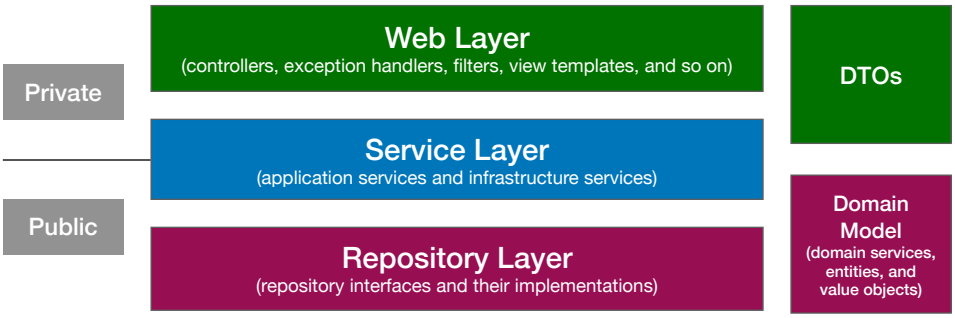
- Create setters for dependencies
- Can be combined easily with constructor injection
- Flexibility in dependency resolution or object reconfiguration

```
@Service
public class HomeController( ) {
    // ...
    @Autowired
    public void setServiceA(ServiceA serviceA) {
        this.serviceA = serviceA;
    }
}
```

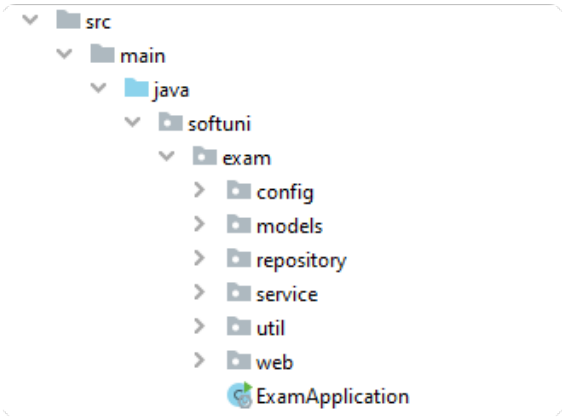
Layers
The Correct Project Structure

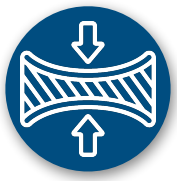
Layers

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



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





Thin Controllers
Creating Simple Components

Thin Controllers

- Controllers should follow well known principles such as DRY and KISS
- Should delegate functionality to the service layer
- The service layer consists of application logic, e.g. services, executors, strategies, mappers, DTOs, entities, etc.

Summary

- Spring MVC - MVC framework that has three main components:
  - Controller - controls the application flow
  - View - presentation layer
  - Model - data component with the main logic
- Constructor injection - the best way for DI
- Splitting your application code by layers
- Every component should be as "thin" as possible