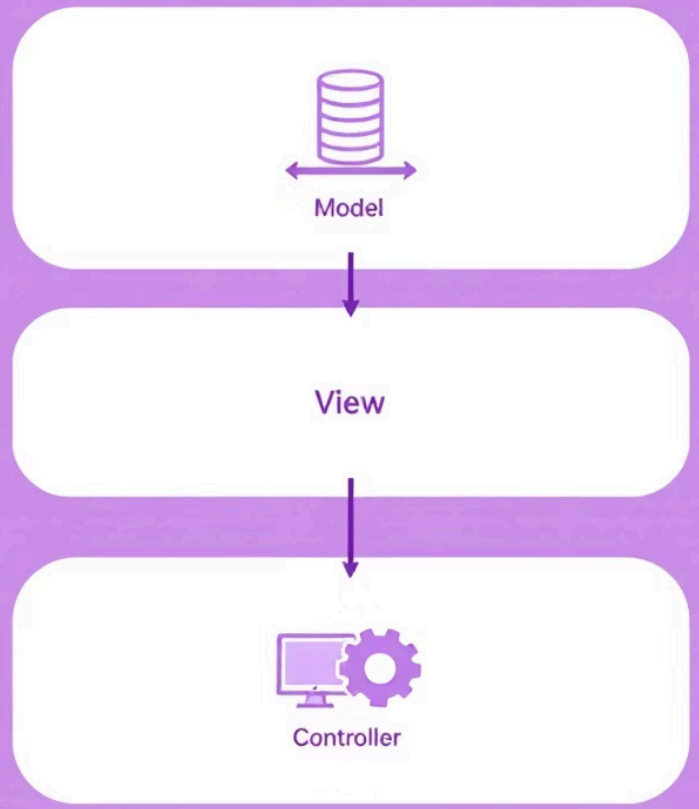
Spring MVC: Building Modern Web Applications

Welcome to this comprehensive guide on Spring MVC, a powerful framework for building web applications in Java.

We'll explore how Spring MVC simplifies development while providing robust architecture for your projects.

S by Saratha Natarajan



The MVC Architectural Pattern

Model

Contains application data and business logic.

Maintains the state and rules of the application.

View

Renders the model data to the user.

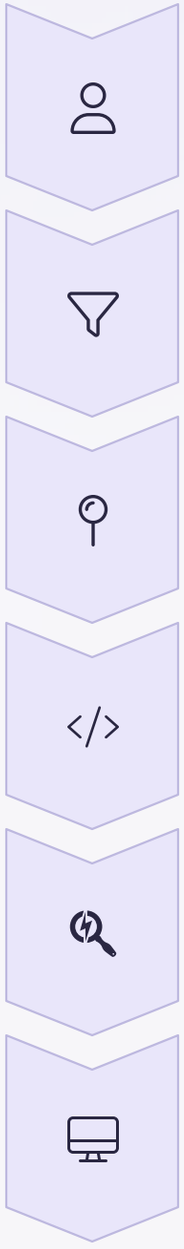
Implemented using JSP, Thymeleaf, or other template engines.

Controller

Processes incoming requests.

Updates the model and selects the appropriate view.

Spring MVC Workflow



Client Request

User sends HTTP request to the server

DispatcherServlet

Central servlet receives all requests

Handler Mapping

Identifies the appropriate controller

Controller Processing

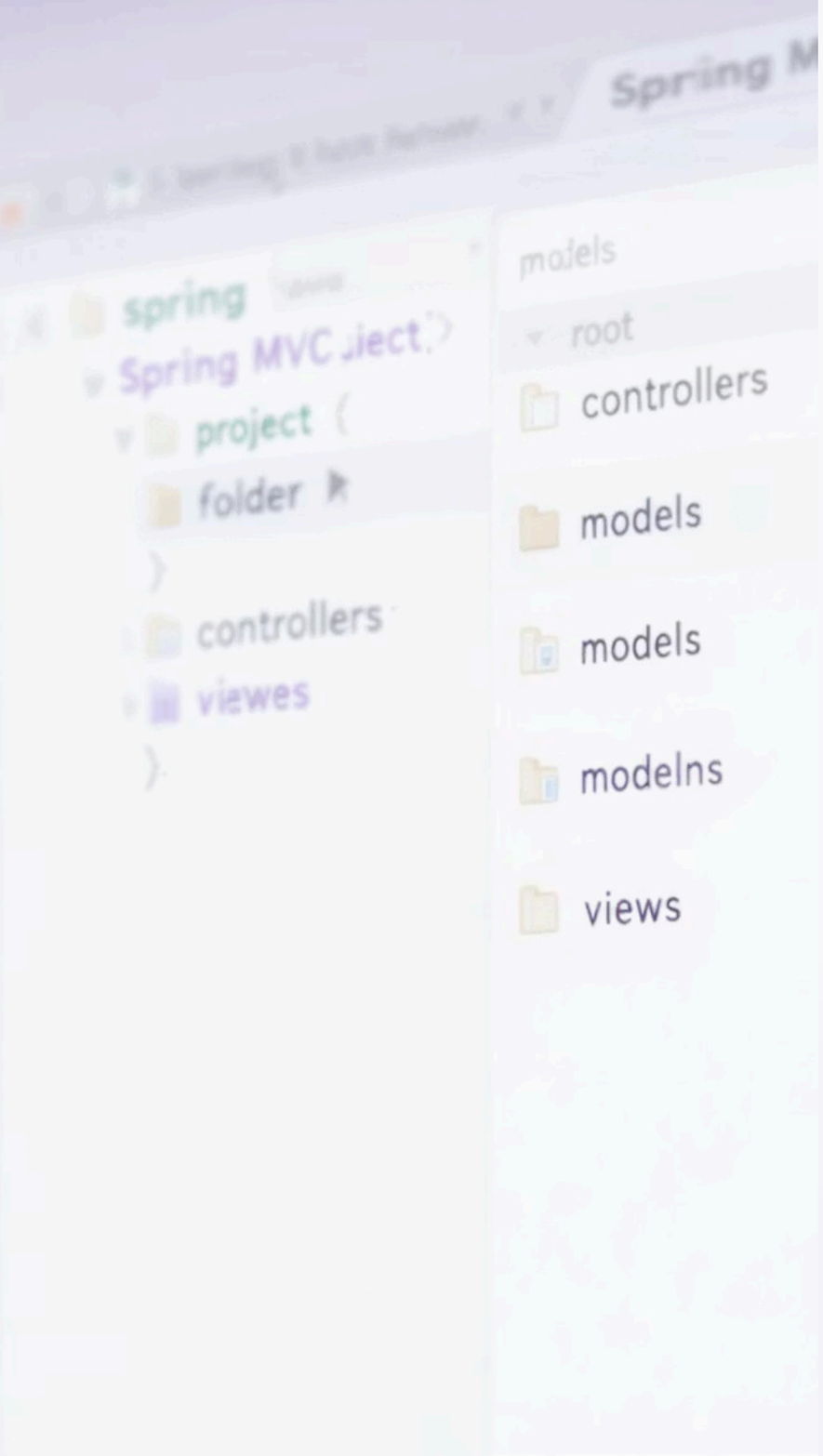
Updates model and returns view name

View Resolution

ViewResolver finds the correct view template

Response Rendering

View renders model data as HTML



Setting Up a Spring MVC Project

Add Dependencies

Include Spring MVC in your Maven/Gradle project.

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-webmvc</artifactId>
  <version>5.3.20</version>
</dependency>
```

Configure DispatcherServlet

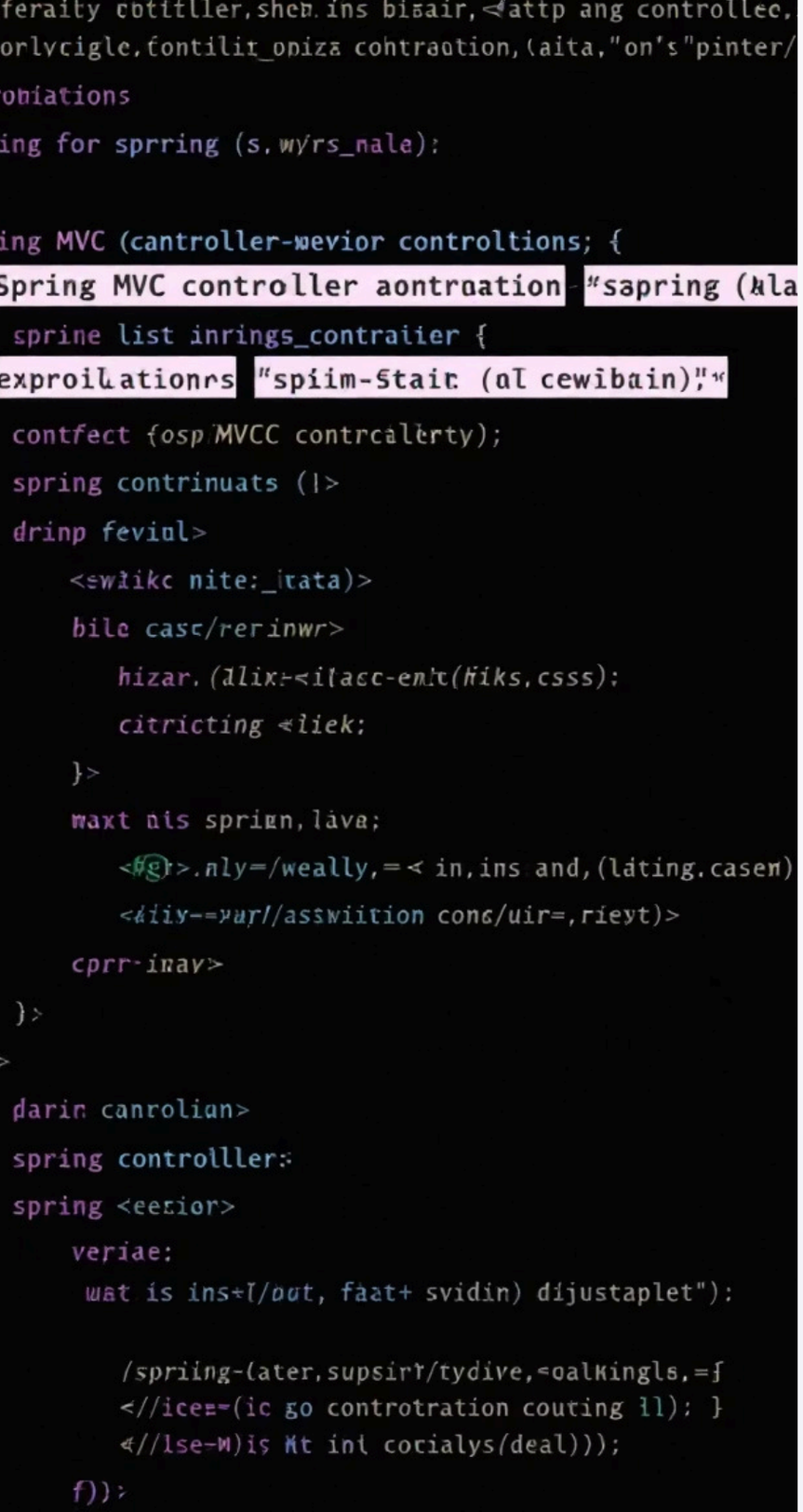
Set up web.xml or create WebApplicationInitializer class.

Create Project Structure

Organize controllers, models, views, and config files.

Write "Hello World"

Create a simple controller to test your setup.



Defining Controllers

@Controller Annotation

Marks a class as a web controller, capable of handling requests.

```
@Controller
public class UserController
{
    // Methods here
}
```

@RequestMapping

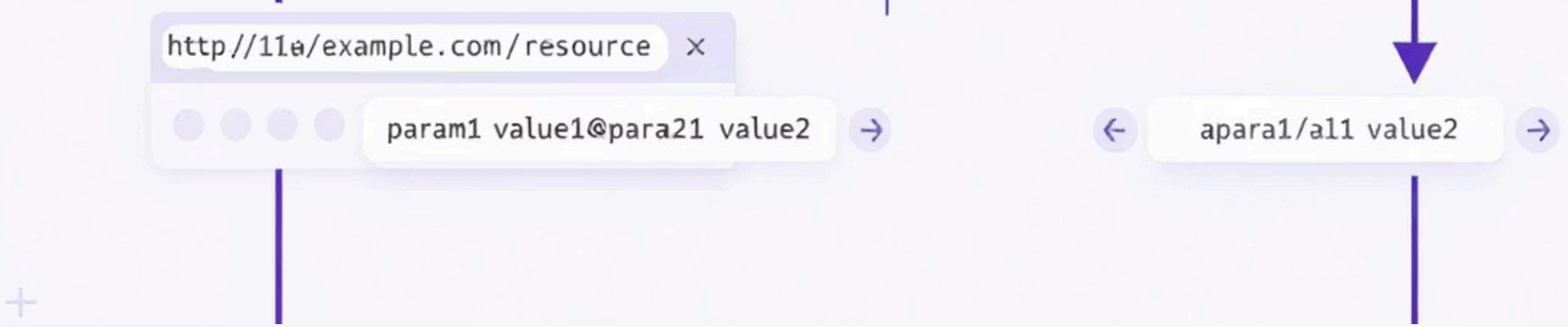
Maps URLs to specific controller methods.

```
@RequestMapping("/users"
)
public class UserController
{
    // Handles /users
requests
}
```

HTTP Method Handling

Use @GetMapping, @PostMapping for specific HTTP methods.

```
@GetMapping("/login")
public String loginForm() {
    return "loginView";
}
```



Handling Requests and Parameters

1

@RequestParam

Extracts query parameters or form fields.

```
@GetMapping("/search")
public String search(@RequestParam String
query) {
// Use query parameter
}
```

2

@RequestBody

Binds HTTP request body to an object.

```
@PostMapping("/api/users")
public User addUser(@RequestBody User user) {
   // Process user object
}
```

3

@PathVariable

Extracts values from the URL path.

```
@GetMapping("/users/{id}")
public User getUser(@PathVariable Long id) {
// Find user by id
}
```

Working with Models

Adding to Model

Use Model parameter to add attributes.

```
public String
showProducts(Model
model) {

model.addAttribute("pro
ducts",
productService.getAll());
   return "productList";
}
```

@ModelAttribute

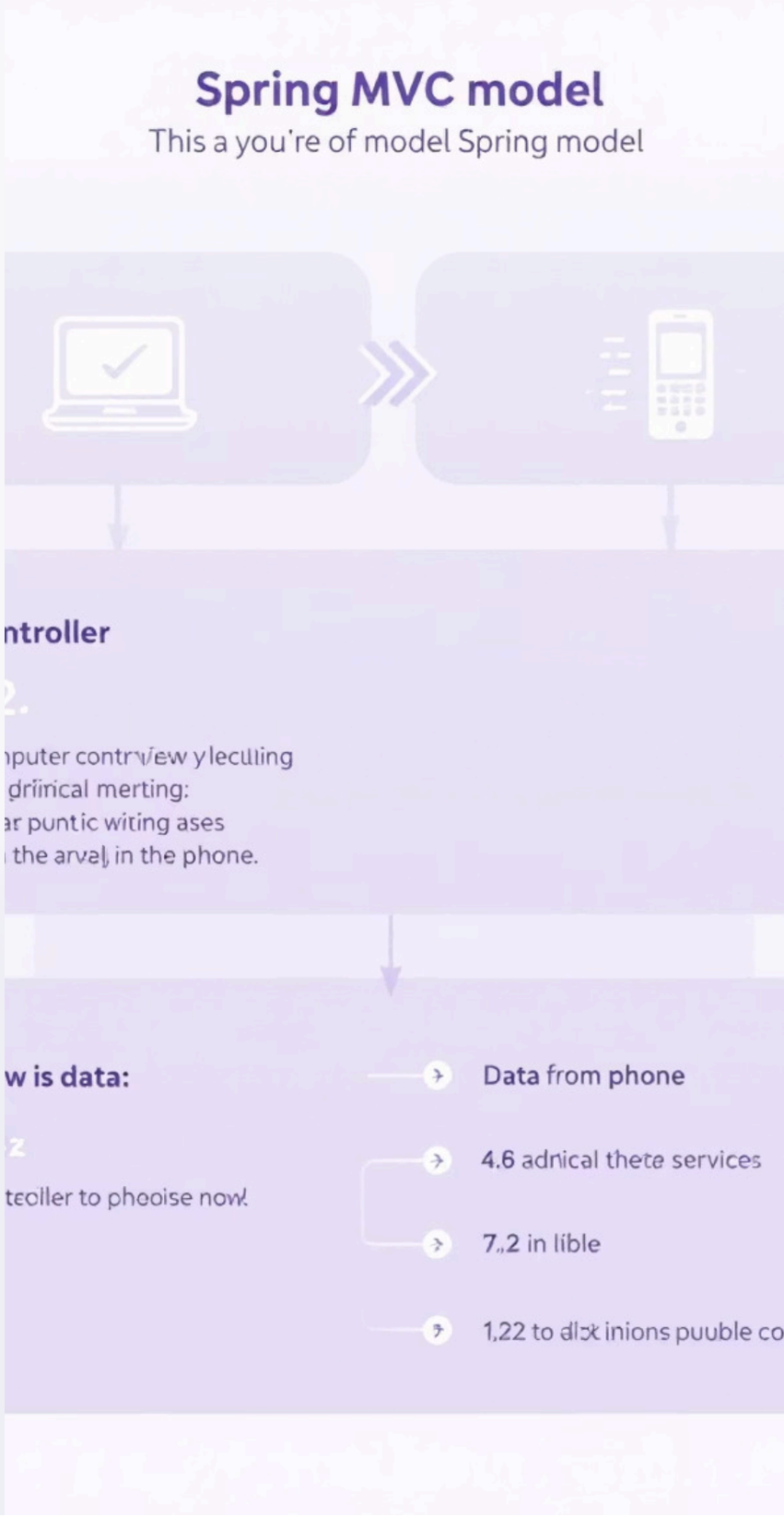
Pre-populates model before request processing.

```
@ModelAttribute("categ
ories")
public List
getCategories() {
   return
categoryService.getAll();
}
```

Data Transfer

Model attributes are accessible in view templates.

```
<!-- In Thymeleaf template -->
<div th:each="product : ${products}">
   <span th:text="${product.name}"></span>
</div>
```



View Technologies

JSP

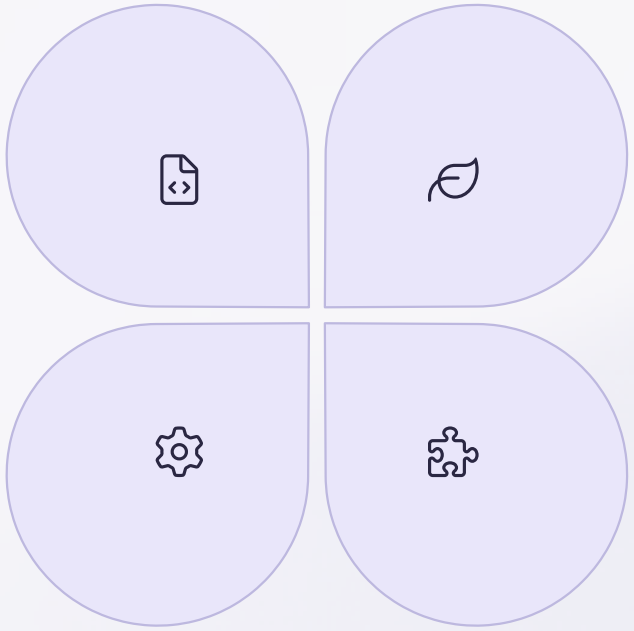
Traditional Java Server Pages.

- Familiar syntax
- JSTL tag library support

Configuration

ViewResolvers map logical view names to templates.

- InternalResourceViewResolver
- ThymeleafViewResolver



Thymeleaf

Modern server-side template engine.

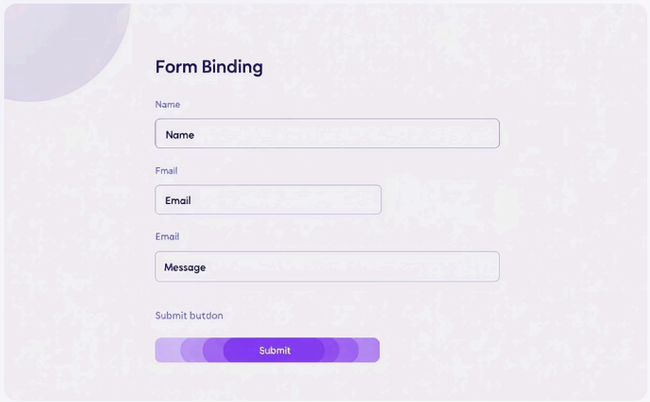
- Natural templating
- Great Spring integration

FreeMarker

Template engine focusing on MVC patterns.

- No servlet dependency
- Powerful macro capabilities

Form Handling



Form Creation

Spring provides form tags to simplify binding.

```
<form:form modelAttribute="user">
    <form:input path="username"/>
    <form:password
path="password"/>
    <input type="submit"
value="Submit"/>
</form:form>
```

Data Binding

Form fields automatically bind to model attributes.

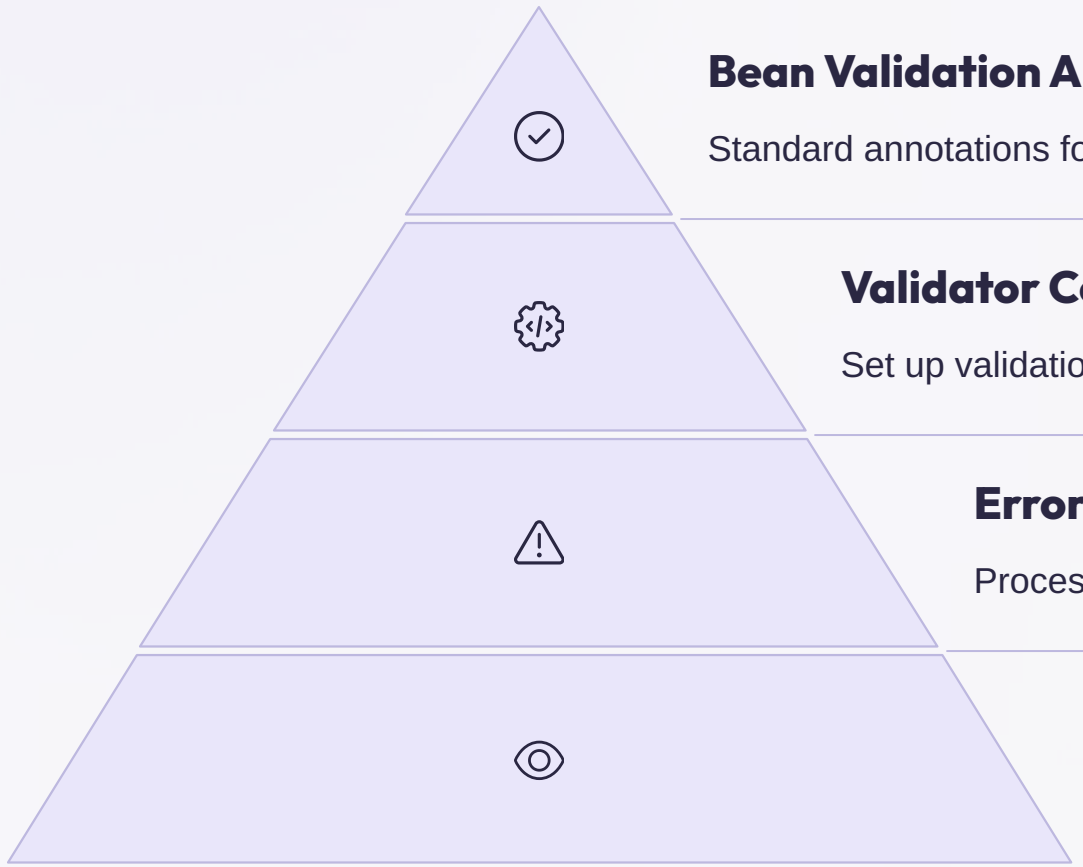
```
@PostMapping("/register")
public String processRegistration(
    @ModelAttribute("user") User
user) {
    // Process user registration
    return "redirect:/success";
}
```

Validation

Enforce data constraints and show error messages.

```
@PostMapping("/register")
public String processRegistration(
    @Valid @ModelAttribute("user")
User user,
    BindingResult result) {
    if (result.hasErrors()) {
        return "registrationForm";
    }
    // Process valid registration
}
```

Validation



Bean Validation API

Standard annotations for constraints

Validator Configuration

Set up validation in Spring context

Error Handling

Process validation errors

Error Display

Show validation messages in view

Validation ensures data integrity. Apply constraints to entity classes with annotations like @NotNull, @Size, and @Email.

Example: A User class with validation constraints:

```
public class User {
    @NotBlank(message = "Username is required")
    private String username;

    @Size(min = 8, message = "Password must be at least 8 characters")
    private String password;

    @Email(message = "Invalid email format")
    private String email;
}
```

Interceptors

Pre-Processing

Executes before controller method.

Can modify request or reject it entirely.

Post-Processing

Executes after controller method.

Can modify model or view returned.

Completion

Executes after response is rendered.

Useful for cleanup operations.

Use Cases

Authentication, logging, performance monitoring.

Example: A logging interceptor to track request execution time.

```
public class LoggingInterceptor implements HandlerInterceptor {
    private static final Logger logger = LoggerFactory.getLogger(LoggingInterceptor.class);

    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) {
        request.setAttribute("startTime", System.currentTimeMillis());
        return true;
    }

    @Override
    public void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex) {
        long startTime = (Long) request.getAttribute("startTime");
        long endTime = System.currentTimeMillis();
        logger.info("Request URL: {} - Time Taken: {}ms", request.getRequestURL(), (endTime - startTime));
    }
}
```

Exception Handling

@ExceptionHandler

Method-level exception handling

@ControllerAdvice

Global exception handling

Custom Error Pages

User-friendly error displays

Handle exceptions elegantly across your application. Use @ExceptionHandler for controller-specific handling or @ControllerAdvice for global handling.

```
@ControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(ResourceNotFoundException.class)
    public ModelAndView handleResourceNotFound(ResourceNotFoundException ex) {
        ModelAndView modelAndView = new ModelAndView("error/not-found");
        modelAndView.addObject("message", ex.getMessage());
        return modelAndView;
    }

    @ExceptionHandler(Exception.class)
    public ModelAndView handleGenericException(Exception ex) {
        ModelAndView modelAndView = new ModelAndView("error/generic");
        modelAndView.addObject("message", "An unexpected error occurred");
        return modelAndView;
    }
}
```

RESTful APIs with Spring MVC

Annotation	Purpose
@RestController	Combines @Controller and @ResponseBody
@RequestMapping	Maps URLs to controller methods
@PathVariable	Extracts values from URL path
@RequestBody	Binds JSON request to Java objects
ResponseEntity	Controls HTTP response status, headers

Example: A RESTful controller for product management:

```
@RestController
@RequestMapping("/api/products")
public class ProductRestController {

    @GetMapping
    public List getAllProducts() {
        return productService.findAll();
    }

    @GetMapping("/{id}")
    public ResponseEntity getProduct(@PathVariable Long id) {
        Product product = productService.findById(id);
        if (product == null) {
            return ResponseEntity.notFound().build();
        }
        return ResponseEntity.ok(product);
    }

    @PostMapping
    public ResponseEntity createProduct(@RequestBody Product product) {
        Product saved = productService.save(product);
        return ResponseEntity
            .created(URI.create("/api/products/" + saved.getId()))
            .body(saved);
    }
}
```

Testing Spring MVC Applications

1

Unit Tests

Test controller methods in isolation with MockMvc.

2

Integration Tests

Test full request-response cycle with Spring test context.

3

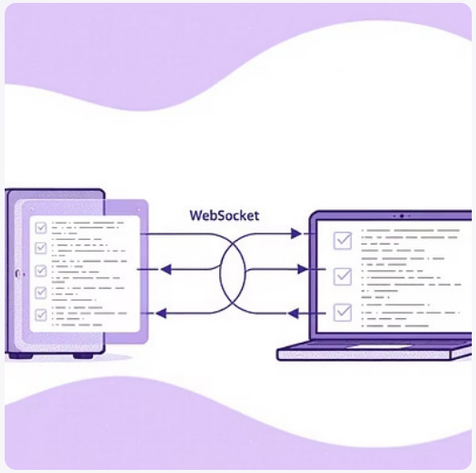
Test Coverage

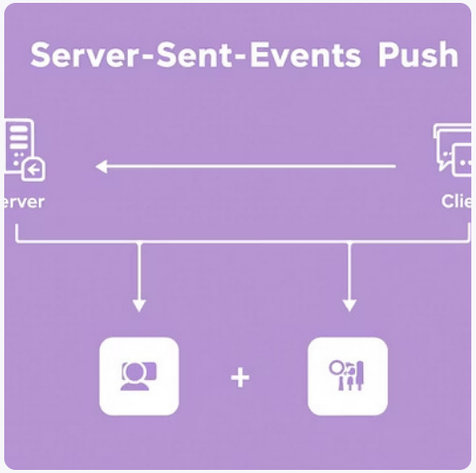
Aim for high coverage of controller and business logic.

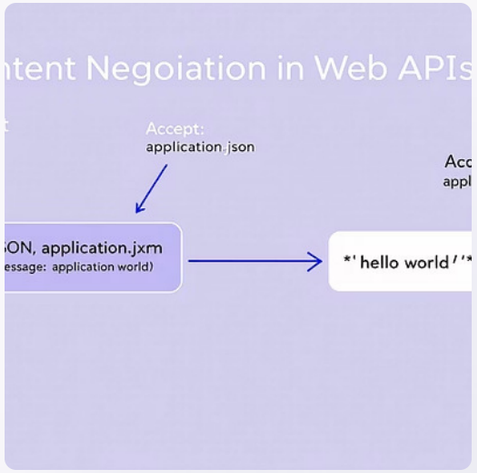
Example: Testing a controller with MockMvc

```
@WebMvcTest(UserController.class)
public class UserControllerTest {

    @Autowired
    private MockMvc mockMvc;

    @MockBean
    private UserService userService;

    @Test
    public void testGetUser() throws Exception {
        User user = new User(1L, "testuser", "test@example.com");
        when(userService.findById(1L)).thenReturn(user);

        mockMvc.perform(get("/users/1"))
            .andExpect(status().isOk())
            .andExpect(view().name("userDetails"))
            .andExpect(model().attributeExists("user"))
            .andExpect(model().attribute("user", hasProperty("username", is("testuser"))));
    }
}
```

Advanced Spring MVC Features







Asynchronous Processing

Handle long-running requests without blocking threads.

```
@GetMapping("/async")
public Callable processAsynchronously() {
    return () -> {
        // Long-running task
        return "asyncResult";
    };
}
```

WebSocket Support

Enable real-time bidirectional communication.

```
@MessageMapping("/chat")
@SendTo("/topic/messages")
public ChatMessage handleChat(ChatMessage
message) {
    return new ChatMessage(message.getFrom(),
message.getText(), new Date());
}
```

Server-Sent Events

Push updates from server to client.

Content Negotiation

Return different response formats (JSON, XML, etc.).

Best Practices and Conclusion

Layered Architecture

Separate concerns with clear layers

Follow Conventions

Use Spring best practices consistently

Thin Controllers

Keep controllers focused on request handling

Dependency Injection

Use Spring's DI for loose coupling

Proper Error Handling

Implement robust exception strategies

Spring MVC provides a powerful foundation for web applications. By following these best practices, you'll create maintainable, scalable systems.

Keep learning and exploring new features as Spring continues to evolve!