**Spring MVC (Model-View-Controller)** is a part of the Spring Framework used for building web applications. It follows the MVC design pattern, which helps in separating the application logic from the user interface.

1. **Model**: Represents the application's data. The model handles data processing and business logic.
2. **View**: Represents the UI (User Interface) of the application. It is responsible for rendering the model data.
3. **Controller**: Acts as an intermediary between the Model and View. It processes user requests, performs business logic, and returns the appropriate view.

**Developing Web Applications with Spring MVC**

1. **Setup and Configuration**
   * **Spring Initializer**: Create a Spring Boot project using Spring Initializer or manually set up dependencies in the pom.xml or build.gradle.
   * **DispatcherServlet**: Central servlet that handles all HTTP requests and responses. It is configured in web.xml or automatically by Spring Boot.
2. **Controller Development**
   * **Annotation-based Configuration**: Use @Controller to mark a class as a controller. Use @RequestMapping to map web requests to specific handler methods.
   * **Handler Methods**: Methods within the controller that handle HTTP requests. These methods can return ModelAndView objects or strings representing view names.

**Code:**

@Controller

public class HomeController {

@RequestMapping("/home")

public String home() {

return "home";

}

}

1. **Model and View**
   * **Model**: Use Model or ModelMap to pass data to the view.
   * **View**: Configure view resolvers to map view names to actual views (e.g., JSP, Thymeleaf).

**code**:

@Controller

public class HomeController {

@RequestMapping("/home")

public String home(Model model) {

model.addAttribute("message", "Welcome to Spring MVC");

return "home";

}

}

1. **Form Handling**
   * **Form Submission**: Use @ModelAttribute to bind form data to a model object.
   * **Form Validation**: Use @Valid and BindingResult to validate form input.

**Code:**

@Controller

public class UserController {

@RequestMapping(value = "/register", method = RequestMethod.GET)

public String showForm(Model model) {

model.addAttribute("user", new User());

return "register";

}

@RequestMapping(value = "/register", method = RequestMethod.POST)

public String submitForm(@Valid @ModelAttribute("user") User user, BindingResult result) {

if (result.hasErrors()) {

return "register";

}

return "success";

}

}

**Advanced Techniques**

1. **Interceptor**
   * Use HandlerInterceptor to intercept requests and perform pre-processing and post-processing logic.

**Code:**

public class MyInterceptor implements HandlerInterceptor {

@Override

public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) {

// Pre-processing logic

return true;

}

@Override

public void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler, ModelAndView modelAndView) {

// Post-processing logic

}

@Override

public void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex) {

// After request completion

}

}

1. **Exception Handling**
   * Use @ExceptionHandler to handle exceptions in controllers.
   * Use @ControllerAdvice to handle exceptions globally.

**Code:**

@Controller

public class HomeController {

@ExceptionHandler(Exception.class)

public String handleException() {

return "error";

}

}

@ControllerAdvice

public class GlobalExceptionHandler {

@ExceptionHandler(Exception.class)

public String handleGlobalException() {

return "error";

}

}

1. **Asynchronous Request Processing**
   * Use @EnableAsync and @Async to process requests asynchronously.

**Code:**

@Configuration

@EnableAsync

public class AppConfig {

}

@Service

public class MyService {

@Async

public void performTask() {

// Asynchronous processing logic

}

}

**Spring Controllers**

1. **Types of Controllers**
   * **Simple Controller**: Basic controller handling simple requests.
   * **Form Controller**: Handles form submission and validation.
   * **MultiAction Controller**: Handles multiple actions in a single controller.
2. **Controller Annotations**
   * @Controller: Marks the class as a Spring MVC controller.
   * @RequestMapping: Maps web requests to specific handler methods.
   * @RequestParam: Binds request parameters to method parameters.
   * @PathVariable: Binds URI template variables to method parameters.
3. **Returning Views**
   * Return view names as strings or use ModelAndView to pass both model data and view names.

**Code:**

@Controller

public class HomeController {

@RequestMapping("/home")

public ModelAndView home() {

ModelAndView mav = new ModelAndView("home");

mav.addObject("message", "Welcome to Spring MVC");

return mav;

}

}

**RESTful Web Services**

1. **Introduction to REST**
   * REST (Representational State Transfer) is an architectural style for designing networked applications. It uses standard HTTP methods (GET, POST, PUT, DELETE) to perform CRUD operations.
2. **Creating RESTful Services with Spring MVC**
   * Use @RestController to create RESTful web services.
   * Use @RequestMapping or @GetMapping, @PostMapping, @PutMapping, @DeleteMapping to map HTTP methods to handler methods.
   * Use @RequestBody to bind request body to method parameters.
   * Use ResponseEntity to manipulate HTTP responses.

**Code:**

@RestController

@RequestMapping("/api")

public class UserController {

@GetMapping("/users")

public List<User> getAllUsers() {

return userService.getAllUsers();

}

@PostMapping("/users")

public User createUser(@RequestBody User user) {

return userService.createUser(user);

}

@PutMapping("/users/{id}")

public User updateUser(@PathVariable Long id, @RequestBody User user) {

return userService.updateUser(id, user);

}

@DeleteMapping("/users/{id}")

public void deleteUser(@PathVariable Long id) {

userService.deleteUser(id);

}

}

1. **Exception Handling in REST**
   * Use @ExceptionHandler to handle exceptions in REST controllers.
   * Use ResponseEntityExceptionHandler for global exception handling.

**Code:**

@RestController

@RequestMapping("/api")

public class UserController {

@ExceptionHandler(UserNotFoundException.class)

public ResponseEntity<String> handleUserNotFoundException(UserNotFoundException ex) {

return new ResponseEntity<>(ex.getMessage(), HttpStatus.NOT\_FOUND);

}

}

1. **Content Negotiation**
   * Configure content negotiation to support multiple formats (e.g., JSON, XML).
   * Use @RequestMapping(produces = MediaType.APPLICATION\_JSON\_VALUE) to specify the response format.

**Code:**

@RestController

@RequestMapping("/api")

public class UserController {

@GetMapping(value = "/users", produces = MediaType.APPLICATION\_JSON\_VALUE)

public List<User> getAllUsers() {

return userService.getAllUsers();

}

}

**Spring Boot** is a framework that simplifies the setup and development of Spring applications. It provides a convention-over-configuration approach and allows developers to create stand-alone, production-ready applications with minimal configuration.

1. **Key Features**
   * **Auto-configuration**: Automatically configures Spring and third-party libraries based on the project's dependencies.
   * **Standalone**: Applications can be run as standalone applications without requiring a traditional application server.
   * **Production-ready**: Provides production-ready features such as metrics, health checks, and externalized configuration.
2. **Spring Boot Initializer**
   * A web-based tool to quickly generate a Spring Boot project with the desired dependencies.

**Using Spring Boot**

1. **Setting Up a Spring Boot Project**
   * **Spring Initializer**: Use the Spring Initializer (<https://start.spring.io/>) to create a new Spring Boot project by selecting the required dependencies and generating the project.
   * **Maven/Gradle**: Manually set up a Spring Boot project by adding the necessary dependencies to the pom.xml or build.gradle file.

Xml code:

<!-- Example for Maven -->

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter</artifactId>

</dependency>

groovy

code:

// Example for Gradle

dependencies {

implementation 'org.springframework.boot:spring-boot-starter'

}

1. **Spring Boot Application**
   * **Main Application Class**: The entry point for a Spring Boot application is a class annotated with @SpringBootApplication. It combines @Configuration, @EnableAutoConfiguration, and @ComponentScan.

Code:

@SpringBootApplication

public class Application {

public static void main(String[] args) {

SpringApplication.run(Application.class, args);

}

}

1. **Running the Application**
   * Use the command mvn spring-boot:run or ./gradlew bootRun to start the application.
2. **Externalized Configuration**
   * Use application.properties or application.yml for externalized configuration. These files allow you to configure various aspects of the application.

properties

code:

# application.properties example

server.port=8080

spring.datasource.url=jdbc:mysql://localhost:3306/mydb

spring.datasource.username=root

spring.datasource.password=password

**Spring Boot Essentials**

1. **Spring Boot Starters**
   * Starters are a set of convenient dependency descriptors you can include in your application. For example, spring-boot-starter-web includes dependencies for building web applications.

Xml code:

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

1. **Spring Boot DevTools**
   * DevTools provides features that help in the development process, such as automatic restarts and live reload.

xml code:

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-devtools</artifactId>

<scope>runtime</scope>

<optional>true</optional>

</dependency>

1. **Spring Boot Actuator**
   * Actuator provides production-ready features such as monitoring and managing the application. It includes endpoints for health checks, metrics, and environment information.

Xml code:

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-actuator</artifactId>

</dependency>

1. **Logging**
   * Spring Boot uses Commons Logging for all internal logging, but leaves the underlying log implementation open. By default, it uses Logback for logging.

properties

code:

# application.properties example for logging

logging.level.org.springframework=INFO

logging.file.name=application.log

**Spring Data JPA**

**Spring Data JPA** is a part of the Spring Data project that makes it easy to implement JPA-based repositories. It simplifies database access by reducing boilerplate code.

1. **Setup**
   * Include spring-boot-starter-data-jpa dependency.

Xml code:

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-data-jpa</artifactId>

</dependency>

1. **Entity Classes**
   * Annotate Java classes with @Entity to map them to database tables.

Code:

@Entity

public class User {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

private String email;

// Getters and setters

}

1. **Repository Interfaces**
   * Create repository interfaces by extending JpaRepository.

Code:

public interface UserRepository extends JpaRepository<User, Long> {

List<User> findByName(String name);

}

1. **Service Layer**
   * Create a service layer to handle business logic.

code

@Service

public class UserService {

@Autowired

private UserRepository userRepository;

public List<User> getAllUsers() {

return userRepository.findAll();

}

public User getUserById(Long id) {

return userRepository.findById(id).orElse(null);

}

public User saveUser(User user) {

return userRepository.save(user);

}

public void deleteUser(Long id) {

userRepository.deleteById(id);

}

}

1. **Controller Layer**
   * Create a controller layer to handle web requests.

code

@RestController

@RequestMapping("/api/users")

public class UserController {

@Autowired

private UserService userService;

@GetMapping

public List<User> getAllUsers() {

return userService.getAllUsers();

}

@GetMapping("/{id}")

public User getUserById(@PathVariable Long id) {

return userService.getUserById(id);

}

@PostMapping

public User createUser(@RequestBody User user) {

return userService.saveUser(user);

}

@PutMapping("/{id}")

public User updateUser(@PathVariable Long id, @RequestBody User user) {

user.setId(id);

return userService.saveUser(user);

}

@DeleteMapping("/{id}")

public void deleteUser(@PathVariable Long id) {

userService.deleteUser(id);

}

}

**Spring Data REST**

**Spring Data REST** builds on top of Spring Data repositories to expose hypermedia-driven RESTful web services.

1. **Setup**
   * Include spring-boot-starter-data-rest dependency.

Xml code

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-data-rest</artifactId>

</dependency>

1. **Exposing Repositories**
   * Simply by including the dependency and defining repository interfaces, Spring Data REST will automatically create RESTful endpoints for the repositories.

code

@RepositoryRestResource

public interface UserRepository extends JpaRepository<User, Long> {

List<User> findByName(String name);

}

1. **Customizing Endpoints**
   * Use @RepositoryRestResource to customize the exposed endpoints.

code

@RepositoryRestResource(path = "users", collectionResourceRel = "users")

public interface UserRepository extends JpaRepository<User, Long> {

List<User> findByName(String name);

}

1. **Event Handling**
   * Use @RepositoryEventHandler to handle repository events.

code

@Component

@RepositoryEventHandler(User.class)

public class UserEventHandler {

@HandleBeforeCreate

public void handleBeforeCreate(User user) {

// Custom logic before creating a user

}

@HandleAfterCreate

public void handleAfterCreate(User user) {

// Custom logic after creating a user

}

}