1. Spring MVC Integration

Overview

Spring MVC (Model-View-Controller) is a web framework built on Servlet API that helps in building flexible and loosely coupled web applications.

Key Components:

- **DispatcherServlet**: Front controller that routes requests
- **Controllers**: Handle requests and return responses
- **View Resolvers**: Resolve views to be rendered
- **Model**: Carries data between controller and view

Example: Basic Controller

java
```java
@Controller
@RequestMapping("/products")
public class ProductController {

  @GetMapping("/{id}")
  public String getProduct(@PathVariable Long id, Model model) {
    model.addAttribute("product", productService.findById(id));
    return "productView"; // resolves to productView.html
  }
}
```

2. Building RESTful APIs

REST Principles:

- Client-server architecture
- Stateless
- Cacheable
- Uniform interface
- Layered system

Example REST Controller:

```
@RestController
@RequestMapping("/api/books")
public class BookController {

   @Autowired
   private BookService bookService;

   @GetMapping
   public List<Book> getAllBooks() {
      return bookService.findAll();
   }

   @GetMapping("/{id}")
   public ResponseEntity<Book> getBookById(@PathVariable Long id) {
      return ResponseEntity.ok(bookService.findById(id));
   }

   @PostMapping
   public ResponseEntity<Book> createBook(@RequestBody Book book) {
      return new ResponseEntity<>(bookService.save(book), HttpStatus.CREATED);
   }

   @PutMapping("/{id}")
   public ResponseEntity<Book> updateBook(@PathVariable Long id, @RequestBody Book book)
{
      return ResponseEntity.ok(bookService.update(id, book));
   }

   @DeleteMapping("/{id}")
   public ResponseEntity<Void> deleteBook(@PathVariable Long id) {
      bookService.delete(id);
      return ResponseEntity.noContent().build();
   }
}
```

3. HTTP Methods (GET, PUT, POST, DELETE)

Method	**Description**	**Example Use Case**	**Status Codes**
GET	Retrieve resource(s)	Fetch product details	200 OK, 404 Not Found
POST	Create new resource	Add new user	201 Created, 400 Bad Request
PUT	Update existing resource	Modify product info	200 OK, 404 Not Found
DELETE	Remove resource	Delete user account	204 No Content, 404 Not Found

4. Working with Spring Data JPA

Repository Interfaces:

- **CrudRepository**: Basic CRUD operations
- **JpaRepository**: Extends CrudRepository with JPA-specific methods

Example Entity and Repository:

```
@Entity
public class Employee {
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;
  private String name;
  private String email;
  // getters, setters, constructors
}

public interface EmployeeRepository extends JpaRepository<Employee, Long> {
  // Custom query methods
  List<Employee> findByName(String name);

  @Query("SELECT e FROM Employee e WHERE e.email LIKE %?1%")
  List<Employee> findByEmailContaining(String emailPart);
}
```

5. Spring Security Fundamentals

Key Features:

- Authentication and Authorization
- CSRF protection
- Session management
- Password encoding

Basic Configuration:

```
@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

   @Override
   protected void configure(HttpSecurity http) throws Exception {
      http
         .authorizeRequests()
            .antMatchers("/public/**").permitAll()
            .antMatchers("/admin/**").hasRole("ADMIN")
            .anyRequest().authenticated()
         .and()
         .formLogin()
            .loginPage("/login")
            .permitAll()
         .and()
         .logout()
            .permitAll();
   }

   @Bean
   public PasswordEncoder passwordEncoder() {
      return new BCryptPasswordEncoder();
   }
}
```

6. Monitoring Spring Boot Applications

Spring Boot Actuator:

Add dependency:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

Endpoints:

- `/actuator/health`: Application health
- `/actuator/info`: Application info
- `/actuator/metrics`: Application metrics
- `/actuator/beans`: List all Spring beans

7. Spring Boot Admin

Setup:

1. Create Admin Server:

```
@SpringBootApplication
@EnableAdminServer
public class AdminServerApplication {
  public static void main(String[] args) {
    SpringApplication.run(AdminServerApplication.class, args);
  }
}
```

2. Client Configuration (application.properties):

```
spring.boot.admin.client.url=http://localhost:8080
management.endpoints.web.exposure.include=*
```

8. Distributed Tracing with Zipkin

Setup:

1. Add dependencies:

```
<dependency>
```

```xml
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-sleuth</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-zipkin</artifactId>
</dependency>
```

2. Configuration:

```properties
spring.zipkin.base-url=http://localhost:9411
spring.sleuth.sampler.probability=1.0
```

9. API Documentation with Swagger

Setup:

1. Add dependencies:

```xml
<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-boot-starter</artifactId>
    <version>3.0.0</version>
</dependency>
```

2. Configuration:

```java
@Configuration
@EnableSwagger2
public class SwaggerConfig {

    @Bean
    public Docket api() {
        return new Docket(DocumentationType.SWAGGER_2)
            .select()
            .apis(RequestHandlerSelectors.any())
            .paths(PathSelectors.any())
            .build();
    }
}
```

Access UI at: `http://localhost:8080/swagger-ui/`

10. Testing Spring Boot Applications

Testing Types:

1. **@DataJpaTest**: Tests JPA components

java
```java
@DataJpaTest
public class EmployeeRepositoryTest {

    @Autowired
    private TestEntityManager entityManager;

    @Autowired
    private EmployeeRepository repository;

    @Test
    public void whenFindByName_thenReturnEmployee() {
        Employee emp = new Employee("John", "john@example.com");
        entityManager.persist(emp);
        entityManager.flush();

        Employee found = repository.findByName(emp.getName());
        assertThat(found.getName()).isEqualTo(emp.getName());
    }
}
```

2. **@SpringBootTest**: Full application context test

java
```java
@SpringBootTest
public class ProductServiceIntegrationTest {

    @Autowired
    private ProductService productService;

    @Test
    public void whenValidId_thenProductShouldBeFound() {
        Product found = productService.getProductById(1L);
        assertThat(found.getId()).isEqualTo(1L);
    }
}
```

3. **MockMvc**: Web layer testing

```
@WebMvcTest(BookController.class)
public class BookControllerTest {

    @Autowired
    private MockMvc mockMvc;

    @MockBean
    private BookService bookService;

    @Test
    public void givenBooks_whenGetBooks_thenReturnJsonArray() throws Exception {
        Book book = new Book("Effective Java", "Joshua Bloch");
        List<Book> allBooks = Arrays.asList(book);

        given(bookService.findAll()).willReturn(allBooks);

        mockMvc.perform(get("/api/books")
            .contentType(MediaType.APPLICATION_JSON))
            .andExpect(status().isOk())
            .andExpect(jsonPath("$", hasSize(1)))
            .andExpect(jsonPath("$[0].title", is(book.getTitle())));
    }
}
```