Índice

1. Introducción a Spring Framework

- ¿Qué es Spring Framework?
- Características principales
- Componentes de Spring
- Beneficios de usar Spring

2. Hibernate: Mapeo objeto-relacional

- ¿Qué es Hibernate?
- Mapeo objeto-relacional
- Configuración de Hibernate
- Sesiones y transacciones en Hibernate

3. Templates MVC en Spring

- Arquitectura MVC
- Spring MVC: Visión general
- Configuración de Spring MVC
- Uso de templates en Spring MVC

Recursos adicionales

- Documentación oficial de Spring: spring.io
- Documentación oficial de Hibernate: hibernate.org
- Tutoriales y ejemplos en el sitio web de Spring: spring.io/guides
- Tutoriales y ejemplos en el sitio web de Hibernate: hibernate.org/orm/documentation

marp: true

¿Qué es Spring Framework?

- Framework de desarrollo de aplicaciones empresariales para Java.
- Enfoque en la inversión de control (IoC) y la programación orientada a aspectos (AOP).
- Proporciona un conjunto de módulos y funcionalidades para simplificar el desarrollo de aplicaciones.
- Promueve la modularidad, el desacoplamiento y la reutilización de código.

Características principales

- Inversión de control (IoC): El contenedor de Spring administra y controla la creación y el ciclo de vida de los objetos.
- Programación orientada a aspectos (AOP): Permite separar preocupaciones transversales como el registro, la seguridad y la transacción.
- Contenedor ligero: No requiere servidores de aplicaciones pesados y puede integrarse fácilmente en diferentes entornos.
- Capacidad de integración: Soporte para integración con otros frameworks y tecnologías populares.

Componentes de Spring

- Core Container: Proporciona IoC y administración de dependencias.
- Data Access/Integration: Capa de acceso a datos y soporte para integración con tecnologías de persistencia.
- Web: Capa web para el desarrollo de aplicaciones web.
- AOP: Aspect-Oriented Programming para el manejo de aspectos transversales.
- Testing: Funcionalidades para pruebas unitarias y de integración.

Beneficios de usar Spring

- Facilita el desarrollo de aplicaciones empresariales complejas.
- Promueve el desacoplamiento y la modularidad del código.
- Permite la reutilización de componentes y la integración con otros frameworks.
- Mejora la testabilidad y facilita las pruebas unitarias e de integración.
- Proporciona soluciones para problemas comunes como la transacción y la seguridad.

Características principales de Spring Boot

- Desarrollo de aplicaciones Java de forma rápida y sencilla.
- Enfoque en la configuración por convención y el inicio rápido.
- Autoconfiguración inteligente y opinionada.
- Soporte para contenedor embebido y despliegue sencillo.
- Administración y monitoreo simplificado de la aplicación.

Desarrollo rápido y sencillo

- Minimiza la configuración manual y la complejidad del desarrollo.
- Permite crear aplicaciones Java de forma ágil y eficiente.
- Provee un conjunto de funcionalidades predefinidas para tareas comunes.

Configuración por convención

- Evita la necesidad de configuraciones extensas y repetitivas.
- Utiliza convenciones basadas en las mejores prácticas de Spring.
- Permite centrarse en la lógica de negocio en lugar de la configuración.

Autoconfiguración inteligente

- Analiza el classpath y las dependencias para configurar automáticamente la aplicación.
- Detecta y configura los componentes necesarios para su correcto funcionamiento.
- Proporciona una configuración predeterminada que se puede personalizar según las necesidades.

Contenedor embebido y despliegue sencillo

- Incluye un contenedor de aplicaciones embebido (Tomcat, Jetty o Undertow).
- Permite ejecutar y desplegar la aplicación de manera independiente, sin necesidad de un servidor externo.
- Facilita el empaquetado de la aplicación en un archivo JAR autoejecutable.

Administración y monitoreo simplificado

- Proporciona endpoints actuator para la administración y supervisión de la aplicación.
- Permite obtener información sobre el estado, las métricas y los logs de la aplicación en tiempo de ejecución.
- Facilita la integración con herramientas de monitoreo y solución de problemas.

Recursos adicionales

- Documentación oficial de Spring Boot: spring.io/projects/spring-boot
- Tutoriales y ejemplos en el sitio web de Spring Boot: [spring.io/guides/spring-boot] (https://spring.io/guides/spring-boot

marp: true

Hola Mundo con Spring Boot

Creación de una aplicación básica

1. Configuración del proyecto
2. Creación de una clase controladora
3. Ejecución y prueba de la aplicación

Configuración del proyecto

1. Crear un nuevo proyecto de Spring Boot en el IDE o mediante el Spring Initializr.
2. Configurar las dependencias necesarias, como `spring-boot-starter-web`.
3. Añadir las anotaciones `@SpringBootApplication` en la clase principal del proyecto.

Creación de una clase controladora

1. Crear una clase Java para actuar como controlador de la aplicación.
2. Anotar la clase con `@RestController`.
3. Definir un método en el controlador y anotarlo con `@RequestMapping` para establecer la ruta del endpoint.

Ejemplo de código:

```
@RestController
public class HelloWorldController {

    @RequestMapping("/hello")
    public String helloWorld() {
        return "¡Hola Mundo!";
    }
}
```

Ejecución y prueba de la aplicación

Ejecutar la aplicación Spring Boot desde el IDE o mediante la línea de comandos.
Acceder a la URL del endpoint definido en el controlador, por ejemplo, http://localhost:8080/hello.
Verificar que se muestra el mensaje "¡Hola Mundo!" en el navegador o en una herramienta como Postman.

marp: true

Hibernate con Spring Boot

- Hibernate: Framework de mapeo objeto-relacional.
- Integración con Spring Boot.
- Configuración y uso de Hibernate en una aplicación Spring Boot.

Hibernate

- Framework de mapeo objeto-relacional (ORM).
- Permite el mapeo de objetos Java a tablas de bases de datos.
- Facilita el acceso y manipulación de datos en una base de datos relacional.
- Proporciona consultas flexibles y soporte para transacciones.

Integración con Spring Boot

- Spring Boot simplifica la configuración de Hibernate.
- Ofrece integración transparente con JPA (Java Persistence API).
- Proporciona características adicionales como autoconfiguración y propiedades configurables.

Configuración de Hibernate en Spring Boot

1. Añadir las dependencias necesarias en el archivo `pom.xml`.
2. Configurar las propiedades de Hibernate en el archivo `application.properties` o `application.yml`.
3. Anotar las clases de entidad con las anotaciones de mapeo de Hibernate.
4. Definir un repositorio utilizando la interfaz `JpaRepository` de Spring Data JPA.

Ejemplo de configuración de Hibernate en Spring Boot

```
<!-- Archivo pom.xml -->
<dependencies>
    <!-- Dependencia de Spring Data JPA -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>
    <!-- Dependencia de Hibernate -->
    <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-core</artifactId>
    </dependency>
</dependencies>
```

Ejemplo de configuración de Hibernate en Spring Boot (continuación)

```
# Archivo application.properties
# Configuración de la base de datos
spring.datasource.url=jdbc:mysql://localhost:3306/mydatabase
spring.datasource.username=username
spring.datasource.password=password

# Configuración de Hibernate
spring.jpa.hibernate.ddl-auto=update
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL8Dialect
```

marp: true

Creación de entidades en Hibernate con Spring Boot

Definición de entidades

- Anotaciones de mapeo de Hibernate
- Propiedades y relaciones de las entidades
- Anotación @Entity

Anotaciones de mapeo de Hibernate

- Hibernate utiliza anotaciones para mapear clases de Java a tablas de la base de datos.
- Algunas anotaciones comunes son:
  - `@Entity`: Indica que la clase es una entidad.
  - `@Table`: Especifica el nombre de la tabla en la base de datos.
  - `@Column`: Mapea una propiedad a una columna en la tabla.
  - `@Id`: Indica la clave primaria de la entidad.

Propiedades y relaciones de las entidades

- Cada propiedad de una entidad se mapea a una columna en la tabla.
- Hibernate proporciona anotaciones para especificar el tipo de datos, restricciones y más.
- Las relaciones entre entidades también se pueden definir utilizando anotaciones como `@OneToOne`, `@OneToMany`, `@ManyToOne`, `@ManyToMany`.

Anotación @Entity

- La anotación `@Entity` indica que una clase es una entidad en Hibernate.
- Se coloca encima de la declaración de clase.
- Es obligatorio utilizar esta anotación para todas las clases de entidad en Hibernate.

Ejemplo de código:

```
@Entity
@Table(name = "customers")
public class Customer {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(name = "name")
    private String name;

    // Otras propiedades y relaciones

    // Getters y setters
}
```

```java
@Entity
@Table(name = "customers")
public class Customer {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(name = "name")
    private String name;

    @OneToMany(mappedBy = "customer", cascade = CascadeType.ALL)
    private List<Order> orders;

    // Otros atributos y relaciones

    // Getters y setters
}

@Entity
@Table(name = "orders")
public class Order {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(name = "order_date")
    private LocalDate orderDate;

    @ManyToOne
    @JoinColumn(name = "customer_id")
    private Customer customer;

    // Otros atributos y relaciones

    // Getters y setters
}
```

Repositorios en Hibernate con Spring Boot

Definición de repositorios

- Interfaz JpaRepository
- Métodos de repositorio predefinidos
- Personalización de consultas

Interfaz JpaRepository

- `JpaRepository` es una interfaz proporcionada por Spring Data JPA.
- Proporciona métodos predefinidos para realizar operaciones CRUD en una entidad.
- Extiende las interfaces `CrudRepository` y `PagingAndSortingRepository`.

Ejemplo de código:

```
public interface CustomerRepository extends JpaRepository<Customer, Long> {
    // Métodos personalizados
}
```

Métodos de repositorio predefinidos

Los métodos predefinidos de JpaRepository incluyen operaciones comunes como save, findById, findAll, delete, etc.
Spring Data JPA genera automáticamente la implementación de estos métodos en tiempo de ejecución.

Ejemplo de código:

```
public interface CustomerRepository extends JpaRepository<Customer, Long> {

    List<Customer> findByName(String name);

    List<Customer> findByEmail(String email);

    List<Customer> findByNameAndEmail(String name, String email);
}
```

Personalización de consultas

Además de los métodos predefinidos, puedes personalizar consultas utilizando el mecanismo de consultas derivadas de Spring Data JPA.
Esto permite definir consultas mediante la nomenclatura del nombre del método.

```
public interface CustomerRepository extends JpaRepository<Customer, Long> {
    List<Customer> findByName(String name);
    List<Customer> findByAgeGreaterThan(int age);
    List<Customer> findByAddressCity(String city);
}
```

Personalización de consultas (continuación)

```
public interface CustomerRepository extends JpaRepository<Customer, Long> {
    @Query("SELECT c FROM Customer c WHERE c.name = :name")
    List<Customer> findByName(@Param("name") String name);

    @Query("SELECT c FROM Customer c WHERE c.age > :age")
    List<Customer> findByAgeGreaterThan(@Param("age") int age);

    @Query("SELECT c FROM Customer c WHERE c.address.city = :city")
    List<Customer> findByAddressCity(@Param("city") String city);
}
```

Personalización de consultas (continuación)

```
public interface CustomerRepository extends JpaRepository<Customer, Long> {
    @Query("SELECT c FROM Customer c WHERE c.name = :name")
    List<Customer> findByName(@Param("name") String name);

    @Query("SELECT c FROM Customer c WHERE c.age > :age")
    List<Customer> findByAgeGreaterThan(@Param("age") int age);

    @Query("SELECT c FROM Customer c WHERE c.address.city = :city")
    List<Customer> findByAddressCity(@Param("city") String city);

    @Query("SELECT c FROM Customer c WHERE c.name = :name AND c.age > :age")
    List<Customer> findByNameAndAgeGreaterThan(@Param("name") String name, @Param("age") int age);
}
```

Controladores REST en Hibernate con Spring Boot

Definición de controladores REST

- Anotaciones @RestController y @RequestMapping
- Mapeo de endpoints
- Métodos para manipulación de recursos

Anotaciones @RestController y @RequestMapping

- `@RestController` se utiliza para indicar que una clase es un controlador REST en Spring Boot.
- `@RequestMapping` se utiliza para mapear la URL base del controlador.

Ejemplo de código:

```
@RestController
@RequestMapping("/api/customers")
public class CustomerController {
    // Métodos para manipulación de recursos
}
```

Mapeo de endpoints

- Los métodos de un controlador REST se mapean a los endpoints de la API.
- Se pueden utilizar las anotaciones `@GetMapping`, `@PostMapping`, `@PutMapping` y `@DeleteMapping` para mapear métodos a los verbos HTTP GET, POST, PUT y DELETE respectivamente.

```
@GetMapping("/{id}")
public ResponseEntity<Customer> getCustomerById(@PathVariable Long id) {
    // Lógica para buscar y devolver un Customer por su ID
}

@PostMapping
public ResponseEntity<Customer> createCustomer(@RequestBody Customer customer) {
    // Lógica para crear un nuevo Customer
}
```

marp: true

Spring Boot Templates

Uso de plantillas en Spring Boot

- Plantillas para generar contenido dinámico
- Motor de plantillas Thymeleaf
- Integración con controladores

Plantillas para generar contenido dinámico

- En una aplicación web, a menudo se necesita generar contenido dinámico que se renderice en el navegador.
- Spring Boot proporciona soporte para diferentes motores de plantillas, como Thymeleaf, FreeMarker, Mustache, etc.
- Las plantillas permiten combinar HTML estático con código dinámico para generar páginas web completas.

Motor de plantillas Thymeleaf

- Thymeleaf es un popular motor de plantillas para aplicaciones web en Spring Boot.
- Ofrece una sintaxis sencilla y fácil de usar para la generación de contenido dinámico.
- Permite el uso de expresiones y atributos específicos para manipular datos en las plantillas.

Integración con controladores

- Los controladores en Spring Boot pueden devolver vistas basadas en plantillas.
- Para ello, se utiliza la anotación `@Controller` en la clase del controlador y se retorna el nombre de la plantilla desde los métodos.
- Spring Boot se encarga de procesar la plantilla y devolverla al navegador.

Ejemplo de código:

```
@Controller
public class HomeController {

    @GetMapping("/")
    public String home(Model model) {
        // Lógica para preparar los datos del modelo
        model.addAttribute("message", "¡Hola, mundo!");
        return "home"; // Nombre de la plantilla a utilizar
    }
}
```

Ejemplo de uso de plantillas en un controlador

```
<!-- Archivo home.html -->
<!DOCTYPE html>
<html>
<head>
    <title>Spring Boot Templates</title>
</head>
<body>
    <h1>¡Bienvenido!</h1>
    <p th:text="${message}"></p>
</body>
</html>
```

Plantilla de lista con Thymeleaf

Ejemplo de una plantilla que muestra una lista de elementos

```
<!-- Archivo list.html -->
<!DOCTYPE html>
<html>
<head>
    <title>Lista de elementos</title>
</head>
<body>
    <h1>Lista de elementos</h1>

    <ul>
        <li th:each="item : ${items}" th:text="${item}"></li>
    </ul>
</body>
</html>
```

Enrutador (Router) en una aplicación web con Spring Boot

¿Qué es un enrutador?

- Componente central de una aplicación web.
- Gestiona las solicitudes HTTP y las dirige a los controladores correspondientes.
- Determina la lógica de enrutamiento basada en las rutas y los métodos HTTP.

Enrutamiento en Spring Boot

- En Spring Boot, el enrutamiento se maneja utilizando el enrutador de Spring MVC.
- Se basa en las anotaciones del controlador y las rutas definidas en ellos.

Ejemplo de código:

```
@Controller
@RequestMapping("/users")
public class UserController {

    @GetMapping("/{id}")
    public String getUserById(@PathVariable Long id) {
        // Lógica para buscar y devolver un usuario por su ID
        return "user";
    }

    @PostMapping
    public String createUser(@RequestBody User user) {
        // Lógica para crear un nuevo usuario
        return "redirect:/users";
    }
}
```

Formulario de creación de Customer

Ejemplo de un formulario para crear un nuevo Customer

```
<!-- Archivo create-customer.html -->
<!DOCTYPE html>
<html>
<head>
    <title>Crear Customer</title>
</head>
<body>
    <h1>Crear Customer</h1>

    <form action="/customers" method="post">
        <label for="name">Nombre:</label>
        <input type="text" id="name" name="name" required><br><br>

        <label for="email">Email:</label>
        <input type="email" id="email" name="email" required><br><br>

        <label for="phone">Teléfono:</label>
        <input type="text" id="phone" name="phone" required><br><br>

        <button type="submit">Crear</button>
    </form>
</body>
```

Controlador para crear un Customer

```java
@Controller
@RequestMapping("/customers")
public class CustomerController {

    @GetMapping("/create")
    public String showCreateForm(Model model) {
        model.addAttribute("customer", new Customer());
        return "create-customer";
    }

    @PostMapping
    public String createCustomer(@ModelAttribute Customer customer) {
        // Lógica para crear el nuevo Customer en la base de datos
        return "redirect:/customers";
    }
}
```

Explicación del controlador

- El controlador CustomerController se encarga de manejar la solicitud GET para la ruta /customer-form.
- El método showCustomerForm devuelve el nombre de la plantilla "customer-form" para mostrar el formulario.
- El método createCustomer maneja la solicitud POST para la ruta /customers y recibe un objeto Customer con los datos del formulario.