स्प्रिंग फ्रेमवर्क की व्यापक गाइड

यह ब्लॉग पोस्ट □□□□□□□□-4□ की सहायता से लिखा गया है।

विषय सूची

- निष्कर्ष

परिचय

स्प्रिंग (□□□□□□) जावा में एंटरप्राइज़-ग्रेड एप्लिकेशन बनाने के लिए सबसे लोकप्रिय फ्रेमवर्क्स में से एक है। यह जावा एप्लिकेशन विकसित करने के लिए व्यापक इंफ्रास्ट्रक्चर सपोर्ट प्रदान करता है। इस ब्लॉग में, हम स्प्रिंग इकोसिस्टम के विभिन्न पहलुओं को कवर करेंगे, जिसमें स्प्रिंग बूट (□□□□□□ □□□□), डेटा प्रबंधन, □□□□□□□ सेवाएं बनाना, शेड्यूलिंग, टेस्टिंग, और स्प्रिंग एडवाइस □□□ जैसी उन्नत सुविधाएं शामिल हैं।

स्प्रिंग बूट फ्रेमवर्क

स्प्रिंग बूट एक शक्तिशाली और लोकप्रिय फ्रेमवर्क है जो □□□□ एप्लिकेशन के विकास को सरल और तेज़ बनाता है। यह स्प्रिंग फ्रेमवर्क का एक एक्सटेंशन है जो डेवलपर्स को स्टैंडअलोन, प्रोडक्शन-ग्रेड स्प्रिंग-आधारित एप्लिकेशन बनाने में मदद करता है। स्प्रिंग बूट का उपयोग करके, आप कम से कम कॉन्फ़िगरेशन के साथ तेज़ी से एप्लिकेशन विकसित कर सकते हैं।

स्प्रिंग बूट की मुख्य विशेषताएं:

- **ऑटो-कॉन्फ़िगरेशन**: स्प्रिंग बूट स्वचालित रूप से आपके एप्लिकेशन के लिए आवश्यक कॉन्फ़िगरेशन सेट करता है।
- **स्टैंडअलोन**: आप अपने एप्लिकेशन को एक स्टैंडअलोन जार (□□□) फ़ाइल के रूप में पैकेज कर सकते हैं जिसे किसी भी □□□□ वातावरण में चलाया जा सकता है।
- **एम्बेडेड सर्वर**: स्प्रिंग बूट में टॉमकैट, जेटी, या अंडरटो जैसे एम्बेडेड सर्वर शामिल होते हैं, जिससे आपको अलग से सर्वर सेटअप करने की आवश्यकता नहीं होती।
- **प्रोडक्शन-रेडी फीचर्स**: स्प्रिंग बूट में हेल्थ चेक, मेट्रिक्स, और एक्सटर्नलाइज़्ड कॉन्फ़िगरेशन जैसे प्रोडक्शन-रेडी फीचर्स शामिल हैं।

स्प्रिंग बूट का उपयोग कैसे करें: स्प्रिंग बूट का उपयोग करने के लिए, आप `Spring Initializr` का उपयोग करके एक नया प्रोजेक्ट बना सकते हैं। यह एक वेब-आधारित टूल है जो आपको अपने प्रोजेक्ट के लिए आवश्यक डिपेंडेंसीज़ और कॉन्फ़िगरेशन चुनने में मदद करता है।

```
@SpringBootApplication
public class MyApplication {
    public static void main(String[] args) {
```

```
        SpringApplication.run(MyApplication.class, args);
    }
}
```

उपरोक्त कोड एक बुनियादी स्प्रिंग बूट एप्लिकेशन को दर्शाता है। `@SpringBootApplication` एनोटेशन स्प्रिंग बूट को बताता है कि यह एक स्प्रिंग बूट एप्लिकेशन है और इसे ऑटो-कॉन्फ़िगर करना चाहिए।

स्प्रिंग बूट का उपयोग करके, आप तेज़ी से और कुशलता से □□□□ एप्लिकेशन विकसित कर सकते हैं। यह डेवलपर्स के लिए एक बहुत ही उपयोगी और लोकप्रिय फ्रेमवर्क है।

□□□□□□ □□□□ **के साथ शुरुआत करना** □□□□□□ □□□□ स्टैंड-अलोन, प्रोडक्शन-ग्रेड □□□□□□-आधारित एप्लिकेशन बनाना आसान बनाता है। यह □□□□□□ प्लेटफॉर्म और तीसरे पक्ष की लाइब्रेरीज़ के प्रति एक राय रखता है, जिससे आप न्यूनतम कॉन्फ़िगरेशन के साथ शुरुआत कर सकते हैं।

- **प्रारंभिक सेटअप**: सबसे पहले □□□□□□ □□□□□□□□□□ का उपयोग करके एक नया □□□□□□ □□□□ प्रोजेक्ट बनाएं। आप अपनी आवश्यकताओं के अनुसार डिपेंडेंसीज चुन सकते हैं, जैसे □□□□□□ □□□, □□□□□□ □□□□ □□□, और □□□□□□ □□□□ □□□□□□□□□।
- **एनोटेशन्स**: मुख्य एनोटेशन्स के बारे में जानें, जैसे `@SpringBootApplication`, जो `@Configuration`, `@EnableAutoConfiguration`, और `@ComponentScan` का संयोजन है।
- **एम्बेडेड सर्वर**: □□□□□□ □□□□ आपके एप्लिकेशन को चलाने के लिए □□□□□□, □□□□□, या □□□□□□□□ जैसे एम्बेडेड सर्वर का उपयोग करता है, इसलिए आपको □□□ फ़ाइलों को किसी बाहरी सर्वर पर डिप्लॉय करने की आवश्यकता नहीं है।

डिपेंडेंसी इंजेक्शन डिपेंडेंसी इंजेक्शन (□□) स्प्रिंग का एक मुख्य सिद्धांत है। यह ढीले युग्मित (□□□□□□□ □□□□□□□) कंपोनेंट्स के निर्माण की अनुमति देता है, जिससे आपका कोड अधिक मॉड्यूलर और परीक्षण करने में आसान हो जाता है।

- □□□□□□□□□□: यह एनोटेशन डिपेंडेंसी को स्वचालित रूप से इंजेक्ट करने के लिए उपयोग किया जाता है। इसे कंस्ट्रक्टर्स, फ़ील्ड्स और मेथड्स पर लागू किया जा सकता है। □□□□□□ की डिपेंडेंसी इंजेक्शन सुविधा स्वचालित रूप से सहयोगी बीन्स को आपके बीन में रिज़ॉल्व और इंजेक्ट कर देगी।

फ़ील्ड इंजेक्शन का उदाहरण: “‘□□□□ □□□□□□□□□□ □□□□□□ □□□□□ □□□□□□□□□□ {

```
  @Autowired
  private UserRepository userRepository;

  //
```

} “‘

कंस्ट्रक्टर इंजेक्शन का उदाहरण: “‘□□□□ □□□□□□□□□□ □□□□□□ □□□□□ □□□□□□□□□□ {

```
  private final UserRepository userRepository;

```java
    @Autowired
    public UserService(UserRepository userRepository) {
        this.userRepository = userRepository;
    }

    //
}
```

मेथड इंजेक्शन का उदाहरण: "'□□□□ □□□□□□□□□□□ □□□□□□□ □□□□□□ □□□□□□□□□□□□□ {

```
  private UserRepository userRepository;

```java
    @Autowired
    public void setUserRepository(UserRepository userRepository) {
        this.userRepository = userRepository;
    }

    //
}
```

- □ □□□□□□□□□□□□, □□□□□□□□□□, □□□□□□□□□□□□□□: ये `@Component` एनोटेशन के विशेष रूप हैं, जो यह इंगित करने के लिए उपयोग किए जाते हैं कि एक क्लास एक □□□□□□□ बीन है। ये एनोटेटेड क्लास की भूमिका के लिए संकेत के रूप में भी काम करते हैं।
  - □□□□□□□□□□□□: यह किसी भी □□□□□□□-प्रबंधित घटक के लिए एक सामान्य स्टीरियोटाइप है। इसका उपयोग किसी भी क्लास को □□□□□□□ बीन के रूप में चिह्नित करने के लिए किया जा सकता है।

उदाहरण:

```java
@Component
public class EmailValidator {

    public boolean isValid(String email) {
        //
```

```
        return true;
    }
}
```

- □□□□□□□□□: यह एनोटेशन `@Component` का एक विशेष रूप है और इसका उपयोग किसी क्लास को एक सेवा के रूप में चिह्नित करने के लिए किया जाता है। यह आमतौर पर सर्विस लेयर में उपयोग किया जाता है, जहां आप बिजनेस लॉजिक को लागू करते हैं।

उदाहरण:

```
@Service
public class UserService {

    @Autowired
    private UserRepository userRepository;

    public User findUserById(Long id) {
        return userRepository.findById(id).orElse(null);
    }
}
```

- □□□□□□□□□□□□: यह एनोटेशन भी `@Component` का एक विशेष रूप है। इसका उपयोग यह इंगित करने के लिए किया जाता है कि क्लास ऑब्जेक्ट्स पर स्टोरेज, रिट्रीवल, खोज, अपडेट और डिलीट ऑपरेशन के लिए मैकेनिज्म प्रदान करती है। यह पर्सिस्टेंस एक्सेप्शन्स को □□□□□□□ के □□□□□□□□□□□□□□□□□□□□□□ पदानुक्रम में भी अनुवादित करता है।

  उदाहरण:

```
@Repository
public interface UserRepository extends JpaRepository<User, Long> {
    //
}
```

ये एनोटेशन आपके □□□□□□□ कॉन्फ़िगरेशन को अधिक पठनीय और संक्षिप्त बनाते हैं, और वे □□□□□□□ फ्रेमवर्क को विभिन्न बीन्स के बीच निर्भरताओं को प्रबंधित और वायर करने में मदद करते हैं।

□□□□□□□ **में** □□□□□□□ □□□□□□□ □□□□□□□□□□ में, □□□□□□□ एक महत्वपूर्ण अवधारणा है जो □□□□□□□□□□□□ के विभिन्न घटकों के बीच संचार को सुविधाजनक बनाती है। □□□□□□□ □□□□□□□ का उपयोग करके, आप □□□□□□□□□□□□ में होने वाली विभिन्न घटनाओं (□□□□□□□) को पकड़ सकते हैं और उनके आधार पर कार्रवाई कर सकते हैं।

□□□□□□□ □□□□□□□ **का उपयोग कैसे करें?**

□□□□□□□ में □□□□□□□ का उपयोग करने के लिए, आपको निम्नलिखित चरणों का पालन करना होगा:

1. □□□□□ □□□□□ **बनाएं**: सबसे पहले, आपको एक □□□□□□□ □□□□□ □□□□□ बनानी होगी जो `ApplicationEvent` □□□□□ को □□□□□□□ करती है।

```
import org.springframework.context.ApplicationEvent;

public class CustomEvent extends ApplicationEvent {
    private String message;

    public CustomEvent(Object source, String message) {
        super(source);
        this.message = message;
    }

    public String getMessage() {
        return message;
    }
}
```

2. □□□□□ □□□□□□□□□□ **बनाएं**: □□□□□ को □□□□□□□□ करने के लिए, आपको `ApplicationEventPublisher` का उपयोग करना होगा।

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.ApplicationEventPublisher;
import org.springframework.stereotype.Component;

@Component
public class CustomEventPublisher {

    @Autowired
    private ApplicationEventPublisher applicationEventPublisher;

    public void publishEvent(String message) {
        CustomEvent customEvent = new CustomEvent(this, message);
        applicationEventPublisher.publishEvent(customEvent);
```

```
    }
}
```

3. □□□□□ □□□□□□□□ **बनाएं**: □□□□□ को सुनने (□□□□□□) और □□□□□□ करने के लिए, आपको एक □□□□□ □□□□□□□□ बनाना होगा।

```
import org.springframework.context.event.EventListener;
import org.springframework.stereotype.Component;

@Component
public class CustomEventListener {

    @EventListener
    public void handleCustomEvent(CustomEvent event) {
        System.out.println("Received custom event - " + event.getMessage());
    }
}
```

4. □□□□□ **को** □□□□□□□ **करें**: अब आप □□□□□ □□□□□□□□ का उपयोग करके □□□□□ को □□□□□□□ कर सकते हैं।

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.stereotype.Component;

@Component
public class EventTrigger implements CommandLineRunner {

    @Autowired
    private CustomEventPublisher customEventPublisher;

    @Override
    public void run(String... args) throws Exception {
        customEventPublisher.publishEvent("Hello, this is a custom event!");
    }
}
```

□□□□□□ □□□□□□ **के फायदे**

- □ □□□□□ □□□□□□□□: □□□□□□ का उपयोग करने से □□□□□□□□□□□ के विभिन्न घटकों के बीच □□□□□□□□ कम होता है।

- □□□□□□□□□□□□□ □□□□□□□□□□□□: □□□□□□□ □□□□□□□ को □□□□□□□□□□□□□□ तरीके से □□□□□□□ किया जा सकता है, जिससे □□□□□□□□□□□□□ में सुधार होता है।
- □□□□□□□□□□□: □□□□□□□ का उपयोग करके □□□□□□□□□□□□□ को □□□□□□□□ बनाया जा सकता है, जिससे □□□□□ को □□□□□□□□□ करना आसान हो जाता है।

□□□□□□□ □□□□□□□ एक शक्तिशाली तंत्र है जो □□□□□□□□□□□□□ के विभिन्न हिस्सों के बीच संचार को सरल और प्रभावी बनाता है। □□□□□□□ की इवेंट मैकेनिज्म आपको एप्लिकेशन इवेंट्स को बनाने और सुनने की अनुमति देती है।

- कस्टम इवेंट्स: `ApplicationEvent` को एक्सटेंड करके कस्टम इवेंट्स बनाएं। उदाहरण के लिए:

```
 public class MyCustomEvent extends ApplicationEvent {
     private String message;

public MyCustomEvent(Object source, String message) {
    super(source);
    this.message = message;
}
```

यह एक कस्टम इवेंट क्लास का कंस्ट्रक्टर है। यह `Object source` और `String message` को पैरामीटर के रूप में लेता है। `super(source)` कॉल के माध्यम से यह पैरेंट क्लास के कंस्ट्रक्टर को कॉल करता है, और `this.message` को प्रदान किए गए `message` पैरामीटर के साथ इनिशियलाइज़ करता है।

```
  public String getMessage() {
      return message;
  }
}
```

- इवेंट लिसनर्स: इवेंट को हैंडल करने के लिए `@EventListener` का उपयोग करें या `ApplicationListener` को इम्प्लीमेंट करें। उदाहरण के लिए: "'□□□□ □□□□□□□□□□□ □□□□□□□ □□□□□□ □□□□□□□□□□□□□□□□□ {

```
  @EventListener
  public void handleMyCustomEvent(MyCustomEvent event) {
      System.out.println("Received spring custom event - " + event.getMessage());
  }
}
```

- इवेंट्स प्रकाशित करना: `ApplicationEventPublisher` का उपयोग करके इवेंट्स प्रकाशित करें। उदाहरण के लिए:

```
    @Component
    public class MyEventPublisher {

        @Autowired
        private ApplicationEventPublisher applicationEventPublisher;

public void publishCustomEvent(final String message) {
    System.out.println("                ");
    MyCustomEvent customEvent = new MyCustomEvent(this, message);
    applicationEventPublisher.publishEvent(customEvent);
}
}
```

स्प्रिंग के साथ डेटा प्रबंधन

□□□□□□□ □□□□ □□□□ □□□□□□□ □□□□ □□□□, □□□□□□□ □□□□ परियोजना का एक हिस्सा है, जो □□□□ (□□□□ □□□□□□□□□ □□□□□□□□□□□□□) के साथ काम करने के लिए एक सरल और प्रभावी तरीका प्रदान करता है। यह डेटाबेस इंटरैक्शन को सरल बनाता है और डेवलपर्स को □□□ क्वेरीज़ लिखने और डेटाबेस ऑपरेशन्स को मैनेज करने में मदद करता है।

□□□□□□□ □□□□ □□□□ का उपयोग करके, आप अपने एप्लिकेशन में डेटाबेस ऑपरेशन्स को आसानी से इंटीग्रेट कर सकते हैं। यह डेटाबेस टेबल्स और □□□□ ऑब्जेक्ट्स के बीच मैपिंग को सरल बनाता है और □□□□ (□□□□□□□, □□□□, □□□□□□□, □□□□□□□) ऑपरेशन्स को सपोर्ट करता है।

मुख्य विशेषताएं:

- **सरल कॉन्फ़िगरेशन**: □□□□□□□ □□□□ □□□□ को कॉन्फ़िगर करना आसान है और यह □□□□□□□ □□□□ के साथ अच्छी तरह से इंटीग्रेट होता है।
- **डोमेन-ड्रिवन डिज़ाइन**: यह डोमेन-ड्रिवन डिज़ाइन को सपोर्ट करता है, जिससे डेटाबेस ऑपरेशन्स को डोमेन ऑब्जेक्ट्स के साथ मैप किया जा सकता है।
- □□□ **क्वेरीज़**: □□□□□□□ □□□□ □□□□ □□□ क्वेरीज़ को सीधे एक्सेक्यूट करने की अनुमति देता है, जिससे डेवलपर्स को अधिक लचीलापन मिलता है।
- □□□□ **ऑपरेशन्स**: यह □□□□ ऑपरेशन्स को सपोर्ट करता है और डेटाबेस इंटरैक्शन को सरल बनाता है।

उदाहरण:

```
import org.springframework.data.annotation.Id;
import org.springframework.data.relational.core.mapping.Table;

@Table("users")
public class User {
    @Id
    private Long id;
    private String name;
    private String email;

    // Getters and Setters
}

import org.springframework.data.repository.CrudRepository;

public interface UserRepository extends CrudRepository<User, Long> {
    // Custom query methods can be defined here
}
```

इस उदाहरण में, `User` क्लास एक डेटाबेस टेबल `users` को रिप्रेजेंट करती है। `UserRepository` इंटरफ़ेस `CrudRepository` को एक्सटेंड करता है, जो □□□□ ऑपरेशन्स को सपोर्ट करता है।

□□□□□□ □□□□ □□□□ का उपयोग करके, आप अपने एप्लिकेशन में डेटाबेस ऑपरेशन्स को आसानी से इंटीग्रेट कर सकते हैं और डेटाबेस इंटरैक्शन को सरल बना सकते हैं।

□□□□□□ □□□□ □□□□ सरल और प्रभावी □□□□ एक्सेस प्रदान करता है।

- □ रिपॉजिटरीज़: □□□□ ऑपरेशन करने के लिए रिपॉजिटरीज़ को परिभाषित करें। उदाहरण के लिए:

  ```
  public interface UserRepository extends CrudRepository<User, Long> {
  }
  ```

- □ क्वेरीज़: कस्टम क्वेरीज़ को परिभाषित करने के लिए `@Query` जैसे एनोटेशन का उपयोग करें। उदाहरण के लिए:

  ```
  @Query("SELECT * FROM users WHERE username = :username")
  User findByUsername(String username);
  ```

□□□□□□□ □□□□□ □□□□ □□□□□□□ □□□□□ □□□□, □□□□-आधारित रिपॉजिटरी को लागू करना आसान बनाता है।

- □ इकाई मैपिंग: `@Entity` का उपयोग करके इकाइयों को परिभाषित करें और उन्हें डेटाबेस टेबल्स से मैप करें। उदाहरण के लिए:

```
@Entity
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String username;
    private String password;
    //
}
```

- □ रिपॉजिटरीज: `JpaRepository` का विस्तार करके रिपॉजिटरी इंटरफेस बनाएं। उदाहरण के लिए:

```
public interface UserRepository extends JpaRepository<User, Long> {
}
```

- □ क्वेरी मेथड्स: डेटाबेस ऑपरेशन करने के लिए क्वेरी मेथड्स का उपयोग करें। उदाहरण के लिए:

```
List<User> findByUsername(String username);
```

□□□□□□□ □□□□□ □□□□□□ □□□□□□□ □□□□□ □□□□□□, □□□□□□-आधारित डेटा एक्सेस के लिए बुनियादी ढांचा प्रदान करता है।

- □ □□□□□□□□□□□□□□□: □□□□□□ के साथ इंटरैक्ट करने के लिए `RedisTemplate` का उपयोग करें। उदाहरण के लिए:

```
@Autowired
private RedisTemplate<String, Object> redisTemplate;
```

□□□□□□□ □□□□□ □□□□□(□□□□□□□ □□□□, □□□□□□□ □□□□□□) { □□□□□□□□□□□□□□□□.□□□□□□□□□□□□□□□().□□□□(□□□□, □□□□□□); }

इस कोड को हिंदी में समझाएं:

यह एक □□□□□ मेथड है जो `save` नाम से है। यह मेथड दो पैरामीटर लेता है: `key` और `value`। इस मेथड का उद्देश्य □□□□□□ डेटाबेस में डेटा स्टोर करना है।

- □ `redisTemplate.opsForValue().set(key, value);` यह लाइन □□□□□□ में `key` और `value` को सेट करती है। `redisTemplate` एक □□□□□□□ □□□□□ □□□□□□ का ऑब्जेक्ट है जो □□□□□□ के साथ इंटरैक्ट करने के लिए उपयोग किया जाता है। `opsForValue()` मेथड □□□□□□ के स्ट्रिंग ऑपरेशन्स को एक्सेस करने के लिए उपयोग किया जाता है, और `set(key, value)` मेथड दिए गए `key` के साथ `value` को □□□□□□ में स्टोर करता है।

इस प्रकार, यह मेथड □□□□□ में डेटा को स्टोर करने के लिए उपयोग किया जाता है।

```
public Object find(String key) {
    return redisTemplate.opsForValue().get(key);
}
```

इस कोड को हिंदी में समझाएं:

यह एक □□□□ मेथड है जो `find` नाम से है। यह मेथड एक `String` प्रकार की `key` को पैरामीटर के रूप में लेता है और `Object` प्रकार का मान वापस करता है।

इस मेथड के अंदर, `redisTemplate.opsForValue().get(key)` का उपयोग किया गया है। यह □□□□□ में संग्रहीत डेटा को पुनः प्राप्त करने के लिए है। `redisTemplate` एक □□□□□ टेम्पलेट ऑब्जेक्ट है जो □□□□□ के साथ इंटरैक्ट करने के लिए उपयोग किया जाता है। `opsForValue()` मेथड □□□□□ में स्ट्रिंग ऑपरेशन्स के लिए उपयोग किया जाता है, और `get(key)` मेथड दिए गए `key` के लिए संग्रहीत मान को पुनः प्राप्त करता है।

इस प्रकार, यह मेथड □□□□□ में संग्रहीत किसी विशेष `key` के लिए मान को वापस करता है।

- रिपॉजिटरीज़: `@Repository` का उपयोग करके □□□□□ रिपॉजिटरीज़ बनाएं। उदाहरण के लिए:

```
@Repository
public interface RedisRepository extends CrudRepository<RedisEntity, String> {
}
```

लेन-देन और □□□ समर्थन □□□□□□ लेन-देन (□□□□□□□□□□□□□) और □□□ (□□□□ □□□□□□ □□□□□□) सपोर्ट के प्रबंधन को सरल बनाता है।

- लेन-देन प्रबंधन: लेन-देन प्रबंधन के लिए `@Transactional` का उपयोग करें। उदाहरण के लिए:

```
@Transactional
public void saveUser(User user) {
    userRepository.save(user);
}
```

- □□□ पैटर्न: पर्सिस्टेंस लॉजिक को अलग करने के लिए □□□ पैटर्न को लागू करें। उदाहरण के लिए:

```
public class UserDao {
    @Autowired
    private JdbcTemplate jdbcTemplate;
```

```
public User findById(Long id) {
    return jdbcTemplate.queryForObject("SELECT * FROM users WHERE id = ?", new Object[]{id}, new UserRo
}
}
```

□□□□ **और** □□□ □□□□ (□□□□ □□□□□□□□ □□□□□□□□□□□□□□) और □□□ (□□□□□□□-□□□□□□□□□□□ □□□□□□□□) दोनों ही डेटाबेस के साथ इंटरैक्ट करने के लिए उपयोग किए जाने वाले टूल हैं, लेकिन इनके काम करने का तरीका और उद्देश्य अलग-अलग होता है।

□□□□ **(□□□□ □□□□□□□□ □□□□□□□□□□□□□□)**

□□□□ एक □□□□ □□□ है जो डेटाबेस के साथ कनेक्टिविटी प्रदान करता है। यह डेवलपर्स को □□□ क्वेरीज़ को डेटाबेस पर एक्ज़ीक्यूट करने और परिणाम प्राप्त करने की अनुमति देता है। □□□□ का उपयोग करके, आप सीधे डेटाबेस के साथ इंटरैक्ट कर सकते हैं, लेकिन इसमें □□□ क्वेरीज़ लिखने और डेटाबेस टेबल्स को मैनेज करने की ज़िम्मेदारी डेवलपर पर होती है।

उदाहरण:

```
Connection conn = DriverManager.getConnection("jdbc:mysql://localhost:3306/mydb", "user", "password");
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery("SELECT * FROM users");

while (rs.next()) {
    System.out.println(rs.getString("username"));
}

rs.close();
stmt.close();
conn.close();
```

□□□ **(□□□□□□□-□□□□□□□□□□□ □□□□□□□□)**

□□□ एक तकनीक है जो ऑब्जेक्ट-ओरिएंटेड प्रोग्रामिंग और रिलेशनल डेटाबेस के बीच की खाई को पाटती है। □□□ टूल्स (जैसे □□□□□-□□□□, □□□) डेटाबेस टेबल्स को □□□□ ऑब्जेक्ट्स में मैप करते हैं, जिससे डेवलपर्स को □□□ क्वेरीज़ लिखने की आवश्यकता नहीं होती। □□□ का उपयोग करके, आप डेटाबेस ऑपरेशन्स को ऑब्जेक्ट-ओरिएंटेड तरीके से कर सकते हैं।

उदाहरण:

```java
@Entity
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String username;

    // Getters and Setters
}

// Using JPA to fetch users
EntityManagerFactory emf = Persistence.createEntityManagerFactory("my-pu");
EntityManager em = emf.createEntityManager();
List<User> users = em.createQuery("SELECT u FROM User u", User.class).getResultList();

for (User user : users) {
    System.out.println(user.getUsername());
}

em.close();
emf.close();
```

□□□□ और □□□ में अंतर

- □ □□□□ में डेवलपर को □□□ क्वेरीज़ लिखनी पड़ती हैं, जबकि □□□ में डेटाबेस ऑपरेशन्स ऑब्जेक्ट-ओरिएंटेड तरीके से किए जाते हैं।
- □ □□□□ में डेटाबेस कनेक्शन और ट्रांजैक्शन मैनेजमेंट मैन्युअल होता है, जबकि □□□ में यह ऑटोमेटेड होता है।
- □ □□□□ सीधे डेटाबेस के साथ काम करता है, जबकि □□□ डेटाबेस और ऑब्जेक्ट्स के बीच एक लेयर प्रदान करता है।

दोनों ही तकनीकों के अपने फायदे और नुकसान हैं, और इनका चुनाव प्रोजेक्ट की आवश्यकताओं और डेवलपर की पसंद पर निर्भर करता है।

□□□□□□, □□□□ और □□□ (ऑब्जेक्ट-रिलेशनल मैपिंग) के लिए व्यापक समर्थन प्रदान करता है।

- □ □□□□□□□□□□□□□: □□□□ ऑपरेशन्स को `JdbcTemplate` के साथ सरल बनाएं। उदाहरण के लिए:

```java
@Autowired
private JdbcTemplate jdbcTemplate;
```

□□□□□□□ □□□□□ □□□□□□□□() { □□□□□□□ □□□□□□□□□□□□□□□.□□□□□□(“□□□□□□□ * □□□□□ □□□□□□”, □□□ □□□□□-□□□□□□□□□□□()); } “‘

(नोट: कोड ब्लॉक को अनुवादित नहीं किया जाता है क्योंकि यह प्रोग्रामिंग भाषा का हिस्सा है और इसे अपरिवर्तित रहना चाहिए।)

□ □□□□□□□□□□: □□□□□□□□□□ को □□□□□□□ के साथ एकीकृत करें □□□ समर्थन के लिए। उदाहरण के लिए:

```
@Entity
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String username;
    private String password;
    // getters and setters
}
```

□□□□□□□□ **सेवाएं बनाना**

□□□□□□□ □□□□□ **क्लाइंट्स** □□□□□□□ □□□□□ क्लाइंट्स का उपयोग □□□□□□□□ वेब सेवाओं के साथ बातचीत करने के लिए किया जाता है। □□□□□□□ फ्रेमवर्क □□□□□ क्लाइंट्स के लिए कई विकल्प प्रदान करता है, जिनमें `RestTemplate`, `WebClient`, और `RestClient` शामिल हैं। ये क्लाइंट्स □□□□□ अनुरोधों को भेजने और प्रतिक्रियाओं को संसाधित करने के लिए उपयोग किए जाते हैं।

1. □□□□□□□□□□□□□□□

`RestTemplate` एक सिंक्रोनस क्लाइंट है जो □□□□□□□ 3 से उपलब्ध है। यह □□□□□ अनुरोधों को भेजने और प्रतिक्रियाओं को संसाधित करने के लिए उपयोग किया जाता है।

```
RestTemplate restTemplate = new RestTemplate();
String url = "https://api.example.com/data";
String response = restTemplate.getForObject(url, String.class);
System.out.println(response);
```

2. □□□□□□□□□□

`WebClient` एक नॉन-ब्लॉकिंग, रिएक्टिव क्लाइंट है जो □□□□□□ 5 में पेश किया गया था। यह अधिक आधुनिक और लचीला है, और यह अधिक जटिल अनुरोधों को संभालने के लिए उपयुक्त है।

```
WebClient webClient = WebClient.create("https://api.example.com");
Mono<String> response = webClient.get()
    .uri("/data")
    .retrieve()
    .bodyToMono(String.class);
response.subscribe(System.out::println);
```

3. □□□□□□□□□□□

`RestClient` एक नया क्लाइंट है जो □□□□□□ 6 में पेश किया गया है। यह `RestTemplate` और `WebClient` के बीच एक संतुलन प्रदान करता है, और यह सिंक्रोनस और एसिंक्रोनस अनुरोधों को संभालने के लिए उपयुक्त है।

```
RestClient restClient = RestClient.create();
String response = restClient.get()
    .uri("https://api.example.com/data")
    .retrieve()
    .body(String.class);
System.out.println(response);
```

निष्कर्ष

□□□□□□ □□□□ क्लाइंट्स विभिन्न प्रकार के अनुप्रयोगों में □□□□□□□ सेवाओं के साथ बातचीत करने के लिए शक्तिशाली उपकरण प्रदान करते हैं। `RestTemplate` पुराने अनुप्रयोगों के लिए उपयुक्त है, जबकि `WebClient` और `RestClient` आधुनिक, रिएक्टिव अनुप्रयोगों के लिए बेहतर विकल्प हैं।

□□□□□□ □□□□□□□ क्लाइंट बनाना आसान बनाता है।

- □ □□□□□□□□□□□□□: □□□□ अनुरोध करने के लिए `RestTemplate` का उपयोग करें। उदाहरण के लिए:

  ```
  @Autowired
  private RestTemplate restTemplate;
  ```

```
public String getUserInfo(String userId) {
    return restTemplate.getForObject("https://api.example.com/users/" + userId, String.class);
}
```

 □□□□□□□□□□: गैर-ब्लॉकिंग अनुरोधों के लिए प्रतिक्रियाशील `WebClient` का उपयोग करें। उदाहरण के लिए:

```
@Autowired
private WebClient.Builder webClientBuilder;

public Mono<String> getUserInfo(String userId) {
    return webClientBuilder.build()
            .get()
            .uri("https://api.example.com/users/" + userId)
            .retrieve()
            .bodyToMono(String.class);
}
```

□□□□□□□□□□□□ □□□□□□□□□□□□ एक □□□□□□□□□□□□ □□□□ क्लाइंट है जो □□□□□□□ □□□□□ में उपलब्ध है। यह □□□□□□□□ वेब सेवाओं को आसानी से कॉल करने के लिए एक सरल और सुविधाजनक तरीका प्रदान करता है। □□□□□□□□□□□□ का उपयोग करके, आप इंटरफेस बना सकते हैं और उस इंटरफेस के माध्यम से □□□□ □□□ को कॉल कर सकते हैं। यह कोड को सरल और पठनीय बनाता है।

उदाहरण के लिए:

```
@FeignClient(name = "example-service", url = "http://example.com")
public interface ExampleServiceClient {
    @GetMapping("/api/resource")
    String getResource();
}
```

इस उदाहरण में, `ExampleServiceClient` इंटरफेस `http://example.com/api/resource` पर □□□ अनुरोध भेजेगा और प्रतिक्रिया को `String` के रूप में वापस करेगा।

□□□□□□□□□□□□ का उपयोग करने के लिए, आपको अपने □□□□□□□ □□□□ एप्लिकेशन में `spring-cloud-starter-openfeign` डिपेंडेंसी जोड़नी होगी।

□□□□□ एक घोषणात्मक वेब सेवा क्लाइंट है।

 सेटअप: अपने प्रोजेक्ट में □□□□□ जोड़ें और `@FeignClient` एनोटेशन के साथ इंटरफेस बनाएं। उदाहरण के लिए:

```
@FeignClient(name = "user-service", url = "https://api.example.com")
public interface UserServiceClient {
    @GetMapping("/users/{id}")
    String getUserInfo(@PathVariable("id") String userId);
}
```

 कॉन्फ़िगरेशन: इंटरसेप्टर्स और एरर डिकोडर्स के साथ □□□□□ क्लाइंट्स को कस्टमाइज़ करें। उदाहरण के लिए:

```
@Bean
public RequestInterceptor requestInterceptor() {
    return requestTemplate -> requestTemplate.header("Authorization", "Bearer token");
}
```

ईमेल, कार्य और शेड्यूलिंग

ईमेल सहायता □□□□□□ ईमेल भेजने के लिए समर्थन प्रदान करता है।

 □□□□□□□□□□□□□□□: ईमेल भेजने के लिए `JavaMailSender` का उपयोग करें। उदाहरण के लिए:

```
@Autowired
private JavaMailSender mailSender;

public void sendEmail(String to, String subject, String body) {
    SimpleMailMessage message = new SimpleMailMessage();
    message.setTo(to);
    message.setSubject(subject);
    message.setText(body);
    mailSender.send(message);
}
```

 □□□□□□□□□□□□: संलग्नक और □□□□ सामग्री के साथ समृद्ध ईमेल बनाएं। उदाहरण के लिए:

```
@Autowired
private JavaMailSender mailSender;

public void sendRichEmail(String to, String subject, String body, File attachment) throws MessagingEx
    MimeMessage message = mailSender.createMimeMessage();
```

```
        MimeMessageHelper helper = new MimeMessageHelper(message, true);
        helper.setTo(to);
        helper.setSubject(subject);
        helper.setText(body, true);
        helper.addAttachment(attachment.getName(), attachment);
        mailSender.send(message);
    }
```

टास्क निष्पादन और शेड्यूलिंग □□□□□□ का टास्क एक्ज़ीक्यूशन और शेड्यूलिंग सपोर्ट टास्क को चलाना आसान बनाता है।

- □ □□□□□□□□□□: `@Scheduled` का उपयोग करके टास्क्स को शेड्यूल करें। उदाहरण के लिए:

```
@Scheduled(fixedRate = 5000)
public void performTask() {
    System.out.println("Scheduled task running every 5 seconds");
}
```

- □ □□□□□ □□□□□: `@Async` का उपयोग करके कार्यों को एसिंक्रोनस रूप से चलाएं। उदाहरण के लिए:

```
@Async
public void performAsyncTask() {
    System.out.println("Async task running in background");
}
```

स्प्रिंग में टेस्टिंग

□□□□□□□□ **के साथ टेस्टिंग** □□□□□□□□ टेस्टिंग के लिए एक शक्तिशाली मॉक लाइब्रेरी है।

- □ निर्भरताओं का मॉकिंग: मॉक ऑब्जेक्ट बनाने के लिए `@Mock` और `@InjectMocks` का उपयोग करें। उदाहरण के लिए:

```
@RunWith(MockitoJUnitRunner.class)
public class UserServiceTest {
    @Mock
    private UserRepository userRepository;

@InjectMocks
private UserService userService;
```

यह कोड एक □□□□ क्लास में `UserService` नामक एक प्राइवेट वेरिएबल को `@InjectMocks` एनोटेशन के साथ डिक्लेयर करता है। `@InjectMocks` एनोटेशन का उपयोग मॉक ऑब्जेक्ट्स को टेस्ट क्लास में इंजेक्ट करने के लिए किया जाता है, जो आमतौर पर यूनिट टेस्टिंग के दौरान □□□□□□□□ फ्रेमवर्क के साथ प्रयोग किया जाता है।

```
@Test
public void testFindUserById() {
    User user = new User();
    user.setId(1L);
    Mockito.when(userRepository.findById(1L)).thenReturn(Optional.of(user));
}
```

```
        User result = userService.findUserById(1L);
        assertNotNull(result);
        assertEquals(1L, result.getId().longValue());
    }
}
```

हिंदी अनुवाद:

```
        User result = userService.findUserById(1L);
        assertNotNull(result);
        assertEquals(1L, result.getId().longValue());
    }
}
```

व्याख्या: - `User result = userService.findUserById(1L);` - यह कोड `userService` से `1L` आईडी वाले यूजर को ढूंढता है और उसे `result` वेरिएबल में स्टोर करता है। - `assertNotNull(result);` - यह जांचता है कि `result` वेरिएबल `null` नहीं है। - `assertEquals(1L, result.getId().longValue());` - यह जांचता है कि `result` का आईडी `1L` के बराबर है।

□ व्यवहार सत्यापन: मॉक ऑब्जेक्ट्स के साथ इंटरैक्शन को सत्यापित करें। उदाहरण के लिए:

```
Mockito.verify(userRepository, times(1)).findById(1L);
```

□□□□□□□□ **के साथ टेस्टिंग** □□□□□□□□ आपको □□□□□□□ □□□ कंट्रोलर्स का परीक्षण करने की अनुमति देता है।

□ सेटअप: अपने टेस्ट क्लासेस में □□□□□□□□ को कॉन्फ़िगर करें। उदाहरण के लिए:

```
@RunWith(SpringRunner.class)
@WebMvcTest(UserController.class)
public class UserControllerTest {
    @Autowired
    private MockMvc mockMvc;

@Test
public void testGetUser() throws Exception {
    mockMvc.perform(get("/users/1"))
        .andExpect(status().isOk())
        .andExpect(content().contentType(MediaType.APPLICATION_JSON))
        .andExpect(jsonPath("$.id").value(1));
}
}
```

- अनुरोध बिल्डर्स: □□□□ अनुरोधों को अनुकरण करने के लिए अनुरोध बिल्डर्स का उपयोग करें। उदाहरण के लिए:

```
mockMvc.perform(post("/users")
    .contentType(MediaType.APPLICATION_JSON)
    .content("{\"username\":\"john\",\"password\":\"secret\"}"))
    .andExpect(status().isCreated());
```

मॉनिटरिंग और प्रबंधन

□□□□□□□ □□□□ □□□□□□□□□□ □□□□□□□ □□□□ □□□□□□□□□ आपके एप्लिकेशन की निगरानी और प्रबंधन के लिए प्रोडक्शन-रेडी सुविधाएँ प्रदान करता है।

- एंडपॉइंट्स: एप्लिकेशन की स्वास्थ्य स्थिति और मेट्रिक्स की निगरानी के लिए `/actuator/health` और `/actuator/metrics` जैसे एंडपॉइंट्स का उपयोग करें। उदाहरण के लिए:

```
curl http://localhost:8080/actuator/health
```

- कस्टम एंडपॉइंट्स: कस्टम एक्चुएटर एंडपॉइंट्स बनाएं। उदाहरण के लिए:

```
@RestController
@RequestMapping("/actuator")
public class CustomEndpoint {
```

```
    @GetMapping("/custom")
    public Map<String, String> customEndpoint() {
        Map<String, String> response = new HashMap<>();
        response.put("status", "           ");
        return response;
    }
}
```

उन्नत विषय

□□□□□□□ □□□□□□□ □□□□ □□□□□□□ □□□□□□□ □□□□, □□□□□□□ □□□□ (□□□□□□□-□□□□□□□□□ □□□□□□□□□□□□□) का एक महत्वपूर्ण हिस्सा है। यह आपको क्रॉस-कटिंग कंसर्न्स (□□□□□□-□□□□□□□□ □□□□□□□□□) को आपके एप्लिकेशन कोड से अलग करने की अनुमति देता है। □□□□□□□, एक ऐसा कोड होता है जो किसी विशिष्ट जॉइन पॉइंट (□□□□□ □□□□□□) पर निष्पादित होता है। □□□□□□□ में, □□□□□□□ को निम्नलिखित प्रकारों में वर्गीकृत किया जा सकता है:

1. □□□□□□□ □□□□□□□: यह □□□□□□□ जॉइन पॉइंट से पहले निष्पादित होता है। उदाहरण के लिए, किसी मेथड को कॉल करने से पहले कुछ लॉगिंग करना।
2. □□□□□□ □□□□□□□□□□ □□□□□□□: यह □□□□□□□ तब निष्पादित होता है जब जॉइन पॉइंट सफलतापूर्वक समाप्त हो जाता है (यानी, कोई एक्सेप्शन नहीं फेंकता है)।
3. □□□□□□ □□□□□□□□□ □□□□□□□: यह □□□□□□□ तब निष्पादित होता है जब जॉइन पॉइंट एक एक्सेप्शन फेंकता है।
4. □□□□□□ **(**□□□□□□□□**)** □□□□□□□: यह □□□□□□□ जॉइन पॉइंट के समाप्त होने के बाद निष्पादित होता है, चाहे वह सामान्य रूप से समाप्त हुआ हो या एक्सेप्शन के कारण।
5. □□□□□□□ □□□□□□□: यह □□□□□□□ जॉइन पॉइंट के आसपास निष्पादित होता है और यह जॉइन पॉइंट को निष्पादित करने के लिए जिम्मेदार होता है। यह सबसे शक्तिशाली □□□□□□□ है क्योंकि यह जॉइन पॉइंट के निष्पादन को पूरी तरह से नियंत्रित कर सकता है।

उदाहरण: □□□□□□□ □□□□□□□

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.springframework.stereotype.Component;

@Aspect
@Component
```

```
public class LoggingAspect {

    @Before("execution(* com.example.service.*.*(..))")
    public void beforeAdvice() {
        System.out.println("Before method execution: Logging...");
    }
}
```

इस उदाहरण में, `beforeAdvice` मेथड `com.example.service` पैकेज के अंदर किसी भी मेथड के निष्पादन से पहले निष्पादित होगा।

उदाहरण: □□□□□□□ □□□□□□□

```
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
import org.springframework.stereotype.Component;

@Aspect
@Component
public class TransactionAspect {

    @Around("execution(* com.example.service.*.*(..))")
    public Object aroundAdvice(ProceedingJoinPoint joinPoint) throws Throwable {
        System.out.println("Starting transaction...");
        Object result = joinPoint.proceed();
        System.out.println("Committing transaction...");
        return result;
    }
}
```

इस उदाहरण में, `aroundAdvice` मेथड `com.example.service` पैकेज के अंदर किसी भी मेथड के आसपास निष्पादित होगा और ट्रांजैक्शन को मैनेज करेगा।

□□□□□□□ □□□□□□□ □□□ का उपयोग करके, आप अपने एप्लिकेशन में लॉगिंग, ट्रांजैक्शन मैनेजमेंट, सिक्योरिटी, और अन्य क्रॉस-कटिंग कंसर्न्स को आसानी से हैंडल कर सकते हैं।

□□□□□□□ का □□□□□□□ □□□ उन्नत □□□ (□□□□□□□-□□□□□□□□ □□□□□□□□□□□□) क्षमताएं प्रदान करता है।

- □□□□□□□□: `@Aspect` का उपयोग करके एस्पेक्ट्स को परिभाषित करें। उदाहरण के लिए:

```
@Aspect
@Component
public class LoggingAspect {
    @Before("execution(* com.example.service.*.*(..))")
    public void logBefore(JoinPoint joinPoint) {
        System.out.println("      : " + joinPoint.getSignature().getName());
    }

@After("execution(* com.example.service.*.*(..))")
public void logAfter(JoinPoint joinPoint) {
    System.out.println("      : " + joinPoint.getSignature().getName());
}
}
```

- □□□□ □□□□□□□: एस्पेक्ट्स को कहां लागू किया जाना चाहिए, यह परिभाषित करने के लिए जॉइन पॉइंट्स का उपयोग करें। उदाहरण के लिए:

```
@Pointcut("execution(* com.example.service.*.*(..))")
public void serviceMethods() {}
```

□□□□□□□□(“□□□□□□□□□□□□□□□□□()”) □□□□□□□ □□□□□□□ □□□□□□□□□□□(□□□□□□□□□□□□□□□□□□□□□□ □□□□□-□□□□□□) □□□□□□□ □□□□□□□□□□□ { □□□□□□□.□□□□.□□□□□□□□(“मेथड से पहले:” + □□□□□□□□□□.□□□□□□□□□□□□□□□□().□□□□□□□□□()); □□□□□□□ □□□□□□□ = □□□□□□□□□□.□□□□□□□□□(); □□□□□□□.□□□□.□□□□□□□□□(“मेथड के बाद:” + □□□□□-□□□□□□.□□□□□□□□□□□□□□□□().□□□□□□□□□()); □□□□□□□ □□□□□□□; } “‘

निष्कर्ष

स्प्रिंग एक शक्तिशाली और बहुमुखी फ्रेमवर्क है जो एंटरप्राइज़-स्तरीय एप्लिकेशन के विकास को सरल बना सकता है। स्प्रिंग बूट, स्प्रिंग डेटा, स्प्रिंग □□□□, और अन्य स्प्रिंग प्रोजेक्ट्स की सुविधाओं का उपयोग करके, डेवलपर्स मजबूत, स्केलेबल और रखरखाव योग्य एप्लिकेशन को कुशलता से बना सकते हैं। स्प्रिंग बूट एक्च्यूएटर और टेस्टिंग फ्रेमवर्क जैसे टूल्स के साथ, आप यह सुनिश्चित कर सकते हैं कि आपके एप्लिकेशन प्रोडक्शन-रेडी और अच्छी तरह से टेस्ट किए गए हैं।