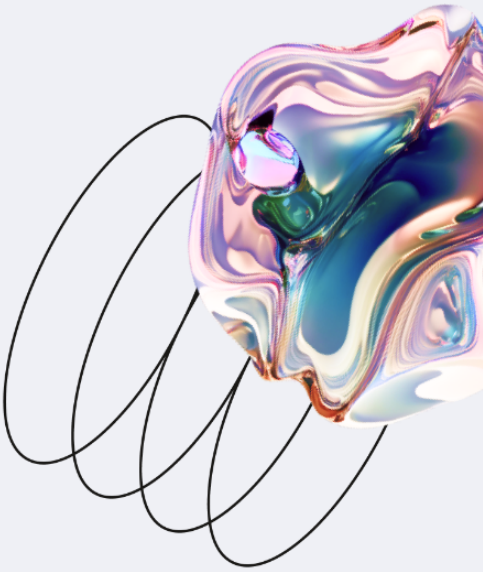
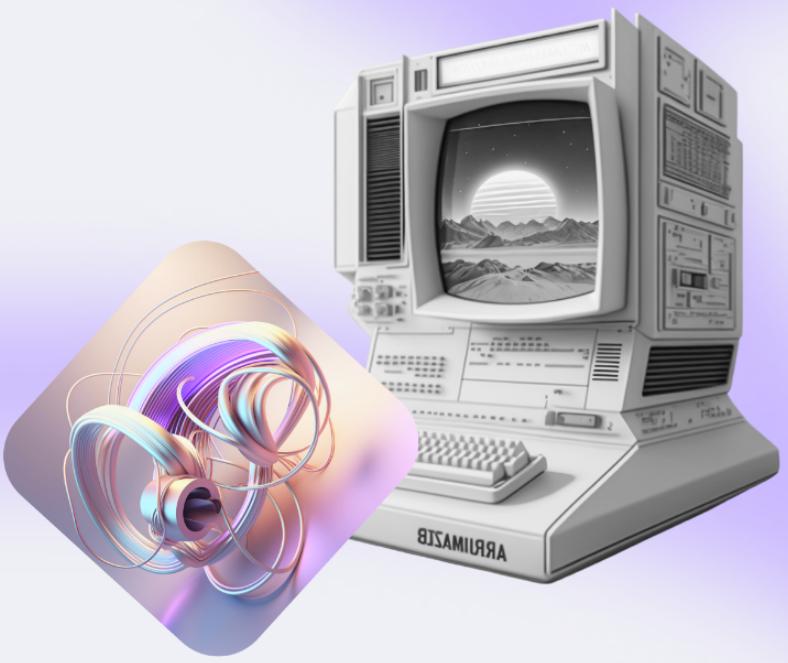
GeekBrains



Паттерны проектирования и GoF паттерны в Spring приложении

Фреймворк Spring



Оглавление

Введение

Привет, ребята! Ну что, готовы продолжить погружение в удивительный мир Spring? Сегодня у нас не просто тема, а настоящая жемчужина в основе знаний каждого разработчика — паттерны проектирования и, конечно же, GoF паттерны в контексте Spring приложений.

Помните наши предыдущие уроки про Spring Testing, Spring Cloud и Spring AOP? Если ты думал, что это было круто, подожди, пока не узнаешь, как все эти знания связаны с паттернами проектирования.

В Spring Testing мы разбирались, как правильно тестировать код, чтобы не стать заложником своих же ошибок в будущем. Но чтобы код был легко тестируемым, его структура должна быть понятной и гибкой. А что может сделать код понятным и гибким? Правильно, паттерны проектирования!

В теме Spring Cloud мы говорили о масштабируемости и облачных решениях. Так вот, хорошо спроектированные паттерны делают ваш код не просто масштабируемым, но и «облачно-готовым».

А в Spring AOP мы обсуждали перехват действий и разделение ответственности в коде. Как думаешь, какие паттерны лучше всего помогут в разделении этой самой ответственности? Ну, сегодня мы на это ответим!

Соединим все точки и покажем, как паттерны проектирования становятся связующим звеном между всеми этими темами. И да, будет код. Много кода. Так что зарядите свои IDE и готовьтесь к действию!

Термины, используемые в лекции

Singleton (Одиночка) — Гарантирует, что класс имеет только один экземпляр и предоставляет глобальную точку доступа к этому экземпляру. В Spring это может быть использовано для создания единственного экземпляра контейнера приложения (ApplicationContext).

Factory Method (Фабричный метод) — Позволяет создавать объекты, не указывая конкретный класс объекта. В Spring это видно в конфигурации бинов через XML-файлы или аннотации, где контейнер создает и возвращает бины.

Dependency Injection (Внедрение зависимостей) — Этот паттерн не из GoF, но он ключевой в Spring. Он обеспечивает инверсию управления и позволяет внедрять

зависимости в бины. В Spring это достигается с помощью аннотаций @Autowired и конфигурации.

Observer (Наблюдатель) — Позволяет объектам следить и реагировать на изменения состояния других объектов. В Spring это может быть использовано в публикации событий и подписке на события через ApplicationEvent и ApplicationListener.

Decorator (Декоратор) — Позволяет добавлять новую функциональность существующим объектам без их изменения. В Spring это может быть использовано, например, для добавления аспектов (Aspects) к бинам.

Proxy (Заместитель) — Позволяет контролировать доступ к объекту, добавляя дополнительную логику. В Spring это может использоваться для создания прокси-объектов для AOP (Аспектно-Ориентированное Программирование).

Template Method (Шаблонный метод) — Определяет скелет алгоритма, но делегирует некоторые шаги подклассам. В Spring это может быть видно в классах-родителях, которые предоставляют базовую логику для создания бинов.

Паттерны Проектирования

Перед тем как углубиться в вихрь кода и примеров, давайте на минутку остановимся и поговорим о том, что такое паттерны проектирования. Это не просто красивые слова, которыми можно похвастаться перед друзьями или на собеседовании. Паттерны — это фундамент хорошего кода, они как строительные блоки, из которых вы строите свои программы.

«А мне-то зачем это?», — могли бы вы спросить. Для бэкенд-разработчика понимание паттернов — это как знание аккордов для музыканта. Это базовые элементы, которые позволяют создавать что-то большое и сложное, не начиная каждый раз с нуля. Сложный проект без паттернов — это как пытаться построить дом, используя только ложки и ведро: теоретически возможно, но зачем, если есть кирпичи и строительный кран?

Ты можешь писать код быстрее, эффективнее, и что важно, так, чтобы другие разработчики понимали, что ты хотел этим сказать. Читаемый и поддерживаемый код — это одна из ключевых ценностей в разработке. Паттерны проектирования делают код более структурированным, прогнозируемым и, в конечном итоге, сохраняют твои нервы и время.

Если ты в бэкенде, то встретишь паттерны на каждом шагу. От организации кода и до масштабирования, от тестирования и до деплоя. Через паттерны можно воплотить наиболее эффективные методы работы с базами данных, API, и так далее.

В следующих разделах мы подробно разберем различные категории паттернов, их применение и как они могут преобразить ваш код. Без них никуда, так что пристегните ремни — полетим в удивительный мир паттернов проектирования в Spring!

GoF Паттерны

GoF или «Gang of Four» паттерны — это не какая-то секретная хакерская группа и не рок-бэнд. Это четверка умных парней, которые собрали в одной книге «Design Patterns» базовые принципы и паттерны, которые можно использовать при проектировании программ. Эти паттерны стали настоящей золотой жилой для разработчиков. В сфере программирования, узнать эту «банду» всё равно что узнать «Битлз» в мире музыки.

Кодовая База Становится Чище: Когда вы используете эти паттерны, ваш код становится более организованным. Представьте, что вы строите дом из Лего. Если у вас есть инструкция, как лучше это делать, ваш дом будет крепче и красивее.

Облегчает Командную Работу: Эти паттерны широко известны, и когда весь ваш тим знает их, вы можете быстрее понять, что делает код другого разработчика. Это как шифр, который знают все ваши коллеги.

Повторное Использование Кода: Паттерны предлагают решения, которые можно применять в разных частях приложения или даже в разных проектах. Это экономит время и усилия на написание нового кода.

Повышение Производительности: Когда код структурирован и хорошо организован, его легче отлаживать и модифицировать. Это сокращает время на разработку и облегчает жизнь разработчика.

Итак, как это выглядит на практике? Допустим, у вас есть магазин, и вы хотите добавить возможность разных способов оплаты. Вы можете использовать паттерн «Стратегия» (Strategy), чтобы легко добавлять новые методы оплаты в будущем.

Или представим, что вы создаете игру, и у вас есть разные типы врагов. Используя паттерн «Фабричный Метод» (Factory Method), вы сможете создавать разные типы врагов, не засоряя основной код игры.

В общем, кто бы что ни говорил, паттерны GoF — это как набор инструментов для разработчика. И чем лучше вы их знаете, тем круче и надежнее будут ваши проекты.

Архитектурные Паттерны

Теперь, когда мы знаем основные категории GoF паттернов, давайте поднимем уровень сложности и перейдем к архитектурным паттернам. Эти паттерны — это как умный и опытный архитектор, который знает, как построить здание так, чтобы оно было не только красивым, но и функциональным.

Микросервисы, MVC и Co.

Если ты слышал такие слова как «Микросервисы», «MVC» или «Event-Driven Architecture», поздравляю, ты уже сталкивался с архитектурными паттернами. Эти паттерны определяют высокоуровневую организацию кода и системы в целом. Они дают рамки, в которых твое приложение будет не просто набором функций, а цельной, хорошо организованной системой.

Архитектура — Это Важно

Архитектурные паттерны смотрят на приложение с высоты 30 000 футов. Они не заботятся о том, как создать один объект или как один класс взаимодействует с другим. Они задаются вопросами типа: «Какие у нас будут основные компоненты системы?», «Как они будут взаимодействовать?» и «Как можно масштабировать это все дело?».

Почему Это Важно в Spring?

Следуя архитектурным паттернам, ты можешь создать Spring приложение, которое не только хорошо спроектировано внутри, но и отлично впишется в любую экосистему, будь то микросервисы или монолитная архитектура. Это сделает твою жизнь гораздо проще, когда дело доходит до тестирования, деплоя и масштабирования.

В следующих разделах мы углубимся в каждую из этих категорий архитектурных паттернов и рассмотрим, как они применяются в Spring. По пути мы столкнемся с

некоторыми довольно крутыми примерами и узнаем, как правильно применять эти знания на практике.

Паттерны Интеграций

А теперь давайте перейдем к не менее важным ребятам — паттернам интеграций. Если архитектурные паттерны — это как архитекторы зданий, то паттерны интеграций — это как инженеры коммуникаций. Их задача — убедиться, что все системы и сервисы могут плавно и эффективно работать вместе.

Склеиваем Все Вместе

Когда у тебя есть разные системы, базы данных, внешние сервисы и API, возникает вопрос: «Как это все сделать совместимым?» Вот тут на помощь приходят паттерны интеграции. Они предлагают проверенные способы, чтобы эти разные части могли не только общаться друг с другом, но и делать это эффективно.

Вот Он, Мир Реальных Задач

В реальной жизни очень редко приходится работать с приложением, которое полностью изолировано от всего мира. Обычно есть как минимум несколько точек взаимодействия с внешними системами. Именно в этих случаях паттерны интеграции становятся настоящими спасателями, позволяя избежать головной боли и ошибок при объединении разных частей системы.

Причем Тут Spring?

Spring Framework предоставляет множество инструментов для работы с интеграцией, начиная от Spring Integration и заканчивая поддержкой различных протоколов и стандартов. Поэтому знание паттернов интеграции сильно упрощает работу с Spring в реальных проектах, где интеграция с внешними системами — это не роскошь, а необходимость.

Так что подготовьтесь, дальше мы погрузимся в примеры и узнаем, как эти паттерны интеграций применять на практике в вашем следующем Spring проекте.

Паттерны Безопасности

А вот мы и подходим к теме, которую нельзя обойти стороной — паттерны безопасности. Все мы любим крутые фичи и гладкую интеграцию, но что из этого, если наши данные утекут, или хуже, если кто-то получит несанкционированный доступ к системе?

Секьюрити — Наш Всё!

Паттерны безопасности в первую очередь нужны для того, чтобы уберечь ваше приложение от различных угроз. Мы говорим о защите данных, аутентификации, авторизации, шифровании и прочих важных аспектах, которые обеспечивают надежную работу системы.

Не Всё Так Просто

Это не просто пароль и логин на входе. Это целая система мер, которая помогает обеспечить безопасную работу не только пользователям, но и самому приложению. Например, как обеспечить, чтобы токен доступа не был украден или как избежать SQL-инъекций.

Паттерны безопасности и Spring

Spring Security — это один из модулей Spring, который позволяет эффективно решать задачи в области безопасности. Он предоставляет инструменты, которые поддерживают множество паттернов безопасности из коробки. Так что знание этих паттернов будет крайне полезным при работе с Spring, чтобы сделать ваше приложение не просто функциональным, но и безопасным.

В следующих главах мы пройдемся по примерам, где покажем, как эти паттерны безопасности можно применять в реальной жизни с использованием Spring.

Структурные GoF Паттерны

Перед тем как погрузиться в конкретику, давай определимся: что такое структурные паттерны в GoF (Gang of Four)? Это паттерны, которые помогают разным компонентам системы лучше сочетаться и взаимодействовать. В контексте Spring это особенно актуально, потому что здесь мы работаем с большим количеством разнородных бинов и компонентов.

Adapter: Переводчик между двумя Мирами

Итак, поговорим про один из самых популярных структурных паттернов — Adapter. Представьте, что у вас есть два класса с разными интерфейсами, которые нужно как-то заставить работать вместе. Adapter в этом случае выступает как переводчик между ними.

Пример на Adapter в Spring

В Spring часто сталкиваются с ситуациями, когда нужно интегрировать сторонние библиотеки или сервисы, которые имеют свою логику и интерфейсы. Идеальный кейс для Adapter!

```
// Сторонний класс, который мы не можем изменить
public class ThirdPartyLibrary {
    public void doSomethingSpecific() {
        // логика
    }
}

// Наш интерфейс
public interface OurInterface {
    void doSomething();
}

// Adapter
public class Adapter implements OurInterface {
    private ThirdPartyLibrary thirdPartyLibrary;

    public Adapter(ThirdPartyLibrary thirdPartyLibrary) {
        this.thirdPartyLibrary = thirdPartyLibrary;
    }

    public void doSomething() {
        thirdPartyLibrary.doSomethingSpecific();
    }
}
```

Здесь Adapter реализует OurInterface и адаптирует методы ThirdPartyLibrary под наш интерфейс. В Spring, вы можете сделать этот адаптер бином и внедрить его там, где нужно использовать OurInterface.

Этот паттерн спасает жизни, когда вы работаете с уже существующими системами или библиотеками и нужно их как-то интегрировать в свой Spring проект без большого рефакторинга.

Так что следующий раз, когда вы столкнетесь с несовместимыми интерфейсами в вашем Spring проекте, вспомните про Adapter. Он может сделать вашу жизнь намного проще!

Bridge и Composite

Знакомство с Adapter было весьма познавательным, не так ли? Но в мире структурных паттернов есть ещё кое-что интересное. Давайте переключимся на два других героя этого сегмента: Bridge и Composite. Они также могут выручить ваши Spring проекты на разных этапах разработки.

Bridge: Разделяй и Властвуй

Первый в списке — Bridge. Этот паттерн помогает разделить абстракцию от реализации, так что обе стороны могут изменяться независимо друг от друга. В Spring это полезно, когда у вас есть разные реализации одного и того же интерфейса и вы хотите динамически переключаться между ними.

Пример на Bridge в Spring

```
public interface Logger {
    void log(String message);
}

public class ConsoleLogger implements Logger {
    public void log(String message) {
        System.out.println("Console: " + message);
    }
}

public class FileLogger implements Logger {
    public void log(String message) {
        // логика записи в файл
    }
}

public class App {
    private Logger logger;

    public App(Logger logger) {
        this.logger = logger;
    }

    public void doSomething() {
        logger.log("Doing something");
```

```
    }
}
```

Так, у нас есть интерфейс Logger и две его реализации. В App мы можем динамически изменить реализацию Logger, не затрагивая остальной код. Используйте Spring для управления этими бинами, и ваш код станет намного гибче.

Composite: Один за Всех и Все за Одного

Следующий в очереди — Composite. Этот паттерн позволяет считать единичный объект и композицию объектов одинаково. Круто для работы с деревьями или графами в Spring!

Пример на Composite в Spring

```
public interface Graphic {
    void draw();
}

public class Circle implements Graphic {
    public void draw() {
        // рисуем круг
    }
}

public class GraphicComposite implements Graphic {
    private List<Graphic> graphics = new ArrayList<>();

    public void addGraphic(Graphic graphic) {
        graphics.add(graphic);
    }

    public void draw() {
        for (Graphic graphic : graphics) {
            graphic.draw();
        }
    }
}
```

Здесь GraphicComposite может содержать как отдельные Graphic элементы, так и другие GraphicComposite, создавая таким образом дерево или граф. С такой структурой удобно работать в Spring, особенно если у вас есть иерархические данные или сложные структуры.

Итак, Bridge и Composite — два мощных инструмента в вашем Spring арсенале. Они помогут сделать ваш код более модульным и легко расширяемым.

Singleton

Так, мы уже покопались в структурных паттернах, таких как Bridge и Composite, и узнали, как они помогают нам в разработке на Spring. Но давайте теперь поговорим про порождающие паттерны. Эти ребята фокусируются на оптимизации процесса создания объектов, что бывает чертовски полезно.

Один из самых популярных (и иногда спорных) порождающих паттернов — это Singleton. Он гарантирует, что класс имеет только один экземпляр и предоставляет глобальную точку доступа к этому экземпляру. В контексте Spring это особенно актуально, потому что Spring по умолчанию создает все бины как Singleton.

Пример на Singleton в Spring

```
@Service
public class SingletonService {
    private static SingletonService instance;

    private SingletonService() {
    }

    public static SingletonService getInstance() {
        if (instance == null) {
            instance = new SingletonService();
        }
        return instance;
    }
}
```

Однако, давайте будем честными: в Spring вам редко придется самостоятельно реализовывать Singleton, потому что Spring делает это за вас. Определите бин, и вот, он уже Singleton!

```
@Service
public class AutomaticallySingletonService {
```

```
    // Этот бин будет Singleton по умолчанию, благодаря Spring
}
```

Если у вас есть сервис, который хранит какое-то общее состояние или кеширует данные, Singleton — ваш выбор. Но помните, Singleton не всегда является лучшим решением. В многопоточной среде или когда требуется больше гибкости, может быть полезно посмотреть на другие порождающие паттерны или даже области Spring бинов, но об этом позже.

Так что Singleton — это как старый добрый швейцарский нож в вашем наборе разработчика на Spring. Прост, но многофункционален.

Factory Method и Prototype

Если Singleton был вашим швейцарским ножом в Spring, тогда Factory Method и Prototype — это как топор и нож для выживания. Они предлагают ещё больше контроля и гибкости при создании объектов, чем Singleton.

Factory Method в Spring

Factory Method стоит использовать, когда у вас есть класс с несколькими подклассами, и на основе входных данных нужно вернуть один из этих подклассов. Это особенно полезно, когда вы хотите обеспечить гибкость и расширяемость вашего приложения. Суть в том, чтобы делегировать логику создания объекта самому классу или его подклассам.

```
public interface PaymentService {
    void pay();
}

@Service("paypal")
public class PaypalPaymentService implements PaymentService {
    //...
}

@Service("creditCard")
public class CreditCardPaymentService implements PaymentService {
    //...
}

@Component
```

```
public class PaymentServiceFactory {
    @Autowired
    private Map<String, PaymentService> services;

    public PaymentService getService(String method) {
        return services.get(method);
    }
}
```

Здесь, PaymentServiceFactory использует Factory Method для возвращения нужного сервиса оплаты. В Spring вы можете это даже автоматизировать с помощью @Autowired и Map<String, Service>.

Prototype: Клонируем с умом

Противоположность Singleton — это Prototype, паттерн, который создаёт новый экземпляр класса каждый раз, когда вы его запрашиваете. Это может быть полезно, когда объекты имеют много внутреннего состояния, которое не должно быть общим.

В Spring для этого есть аннотация @Scope("prototype").

```
@Service
@Scope("prototype")
public class PrototypeService {
    // Каждый раз при запросе будет создан новый экземпляр этого бина
}
```

В отличие от Singleton, вам придётся чаще взаимодействовать с Prototype в вашем коде, потому что Spring не будет управлять жизненным циклом этих бинов за вас.

Короче говоря, Factory Method и Prototype дают вам гибкость и контроль, когда дело доходит до создания объектов в вашем приложении на Spring. Они могут быть полезными в разных сценариях, и чем раньше вы с ними познакомитесь, тем быстрее начнёте писать более эффективный и поддерживаемый код.

Observer и Сила Реактивности в Spring

Окей, мы уже поняли, как создавать объекты с помощью порождающих паттернов, но что насчет взаимодействия между ними? Введем тему поведенческих паттернов, которые как раз и решают эту задачу. И одним из наиболее элегантных решений является паттерн Observer.

Observer: Зачем и Почему

Вы когда-нибудь задавались вопросом, как бы быстро и удобно реализовать механизм подписки в вашем приложении? Например, чтобы пользователь мог подписаться на определенные события и получать уведомления? Тут как раз на помощь приходит Observer.

Observer — это поведенческий паттерн, который предполагает, что у вас есть объект (назовем его «Субъект») и множество «Наблюдателей». Каждый раз, когда в Субъекте происходит какое-то изменение, все его Наблюдатели автоматически получают уведомление. Весьма удобно, правда?

Как это работает в Spring

В Spring вы можете реализовать Observer через механизм событий и слушателей. Для примера, допустим у нас есть приложение для управления задачами, и мы хотим отправлять уведомления каждый раз, когда задача обновляется.

1. Создадим событие:

```
public class TaskUpdatedEvent extends ApplicationEvent {
    private Task task;

    public TaskUpdatedEvent(Object source, Task task) {
        super(source);
        this.task = task;
    }

    // getters
}
```

2. Теперь реализуем слушатель:

```
@Component
public class TaskUpdatedListener implements
ApplicationListener<TaskUpdatedEvent> {
    @Override
```

```
    public void onApplicationEvent(TaskUpdatedEvent event) {
        // Сделать что-то с event.getTask()
    }
}
```

3. И, наконец, опубликуем событие, когда задача обновляется:

```
@Service
public class TaskService {
    @Autowired
    private ApplicationEventPublisher publisher;

    public void updateTask(Task task) {
        // обновляем задачу
        publisher.publishEvent(new TaskUpdatedEvent(this, task));
    }
}
```

Теперь каждый раз, когда метод updateTask() вызывается, все слушатели, подписанные на TaskUpdatedEvent, будут уведомлены.

Таким образом, Observer в Spring это не просто хороший стиль программирования. Это еще и мощный инструмент для создания расширяемых и поддерживаемых систем.

Strategy и Command

Так, друзья, переходим к следующим двум персонажам в нашей истории — паттернам Strategy и Command. Если Observer помогал нам следить за событиями, то эти ребята занимаются тем, чтобы делать наш код гибким в части бизнес-логики и операций.

Strategy: Хитроумный Выбор Логики

Сначала поговорим про Strategy. Этот паттерн призван решить проблему выбора алгоритма во время выполнения программы. Например, у нас есть интернет-магазин, и мы хотим предложить пользователям разные способы оплаты: карточкой, PayPal, криптовалютой и так далее.

В Spring, применение Strategy выглядит как-то так:

1. Создаём интерфейс для стратегии:

```
public interface PaymentStrategy {
    void pay(int amount);
}
```

2. Реализуем конкретные стратегии:

```
public class CardPayment implements PaymentStrategy {
    public void pay(int amount) {
        // Реализация оплаты через карточку
    }
}

public class PayPalPayment implements PaymentStrategy {
    public void pay(int amount) {
        // Реализация оплаты через PayPal
    }
}
```

3. Используем их:

```
@Service
public class PaymentService {
    private PaymentStrategy paymentStrategy;

    public void setPaymentStrategy(PaymentStrategy paymentStrategy) {
        this.paymentStrategy = paymentStrategy;
    }

    public void pay(int amount) {
        paymentStrategy.pay(amount);
    }
}
```

Таким образом, мы можем легко подменять алгоритмы оплаты без изменения основного кода.

Command: Сложные Действия Просто и Понятно

Теперь перейдем к Command. Этот паттерн реализует идею оборачивания запросов или простых операций в объекты. Это особенно удобно, когда у вас есть какие-то операции, которые нужно выполнять асинхронно, или, скажем, отменять.

В контексте Spring, использование Command выглядит примерно так:

```
public interface Command {
    void execute();
}

public class StartCommand implements Command {
    public void execute() {
        // Запуск чего-либо
    }
}

public class StopCommand implements Command {
    public void execute() {
        // Остановка чего-либо
    }
}

@Service
public class CommandExecutor {
    private Command command;

    public void setCommand(Command command) {
        this.command = command;
    }

    public void runCommand() {
        command.execute();
    }
}
```

Теперь, не зависимо от того, что именно нам нужно сделать, мы просто подставляем нужный объект команды и вызываем execute(). Это делает код намного чище и понятнее.

Вот таким образом, Strategy и Command помогают сделать ваш код гибким и удобным для тестирования и расширения. Как видите, в мире Spring паттерны играют не последнюю роль.

MVC: Как Сложить Всё По Полочкам

Окей, давайте немного переключим фокус с GoF паттернов и обратим внимание на архитектурные паттерны. Сейчас мы поговорим про MVC, что расшифровывается

как Model-View-Controller. Этот паттерн, друзья, особенно важен, когда у нас есть какой-то интерфейс пользователя или API.

Зачем это всё нужно?

MVC помогает нам разделить логику приложения на три большие категории:

1. **Model** — этот ребята хранят данные и бизнес-логику.
2. **View** — это наш интерфейс. В вебе это могут быть HTML, CSS и JS файлы.
3. **Controller** — это чуваки, которые слушают, что хочет пользователь, и решают, какой модели и какой вид использовать.

Короче, MVC помогает не превратить ваш проект в кашу, где всё на всём зависит.

Пример на Spring MVC

Помните наш урок про Spring MVC? Так вот, Spring MVC реализует именно этот паттерн. Как это выглядит на практике?

1. **Model** — это ваши сущности и сервисы. Сущность это, скажем, класс User, а сервис UserService управляет этими пользователями: добавляет, удаляет, ищет и так далее.

```
public class User {
    private String name;
    // getters and setters
}
```

2. **View** — это ваш front-end. Например, Thymeleaf шаблоны в Spring.
3. **Controller** — здесь всё интересно. В Spring MVC, контроллеры это классы с аннотацией @Controller, которые принимают HTTP запросы и возвращают HTTP ответы.

```
@Controller
public class UserController {
    @Autowired
    private UserService userService;

    @GetMapping("/users")
    public String listUsers(Model model) {
        model.addAttribute("users", userService.findAll());
        return "users";
```

```
    }
}
```

В этом примере UserController прослушивает GET запрос на /users, получает всех пользователей из UserService (наша Model) и отправляет их на отображение в users шаблон (наш View).

Таким образом, MVC облегчает нам жизнь, позволяя разделить и упорядочить код. А в мире Spring, где у нас уже есть куча инструментов для каждой части MVC, это становится ещё проще и приятнее.

Microservices

Если MVC помогает нам упорядочить код внутри одного приложения, то что делать, когда приложений много и они все разные? Вот тут-то и выходят на сцену микросервисы. Этот архитектурный паттерн предлагает разбить большое, монолитное приложение на множество маленьких, независимых сервисов. Каждый такой сервис делает что-то одно, но делает это хорошо.

Зачем нам микросервисы?

Каждый микросервис можно разрабатывать, развертывать и масштабировать независимо от других. Это упрощает разработку и поддержку. Команда разработчиков на один микросервис может быть меньше, и это позволяет быстрее итерировать и внедрять нововведения.

Про Spring Cloud и микросервисы

Помните, на позапрошлом уроке мы разбирали Spring Cloud? Этот фреймворк отлично подходит для построения микросервисов. В нем есть все, чтобы сделать вашу жизнь проще: сервис для обнаружения других микросервисов, балансировка нагрузки, обработка сбоев и так далее.

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

```
  }
}
```

В этом примере наш «UserService» является Eureka клиентом. Это значит, что он автоматически регистрируется в Eureka сервере и становится доступным для других микросервисов.

Таким образом, микросервисы + Spring Cloud — это комбо, которое дает нам независимость и гибкость на уровне архитектуры всего приложения.

Message Bus

После того как мы с вами поразбирались с микросервисами, возникает следующий вопрос: как же все эти сервисы общаются между собой? Именно здесь на помощь приходят паттерны интеграций. Один из них — это Message Bus.

Что такое Message Bus и зачем он нужен?

Представьте себе автобус, который курсирует между разными частями вашего приложения. Только вместо пассажиров он перевозит сообщения. Этот паттерн позволяет различным компонентам (или даже разным приложениям) общаться друг с другом, не зная деталей реализации каждого.

Пример на Spring Integration

Spring Framework предлагает для этих целей проект под названием Spring Integration. С его помощью можно легко реализовать Message Bus в вашем приложении.

```
@EnableIntegration
public class AppConfig {

    @Bean
    public MessageChannel messageChannel() {
        return new DirectChannel();
    }

    @ServiceActivator(inputChannel = "messageChannel")
    @Bean
    public MessageHandler messageHandler() {
        return message -> System.out.println("Received message: " + message);
```

```
    }
}
```

В этом примере у нас есть канал сообщений messageChannel, и обработчик этих сообщений messageHandler. Все, что вам нужно сделать, это отправить сообщение в этот канал, и Spring Integration позаботится о дальнейшей его доставке.

В общем и целом, Message Bus позволяет упростить общение между разными частями системы и делает ваш код менее связанным. А разве это не то, чего мы все хотим для наших проектов?

Publish/Subscribe

Теперь, когда мы разобрались с тем, как Message Bus может улучшить коммуникацию между компонентами, давайте поговорим о другом паттерне интеграции, который вы просто полюбите. Он называется Publish/Subscribe или Pub/Sub, если хотите сократить.

Что это и для чего нужен Pub/Sub?

В простых словах, Pub/Sub — это паттерн, где одна часть вашего приложения (издатель) отправляет сообщения, а другие части (подписчики) их принимают. И самое крутое — издатель не знает о существовании подписчиков, и подписчики могут подписываться и отписываться на лету.

Как это работает на примере Spring?

Spring предоставляет класс ApplicationEventPublisher для реализации Pub/Sub. Взгляните на пример:

```
@Component
public class Publisher {

    @Autowired
    private ApplicationEventPublisher publisher;

    public void doStuffAndPublishEvent() {
        // Делаем что-то полезное
        publisher.publishEvent(new MyCustomEvent(this, "Event message"));
    }
}
```

```
@Component
public class Subscriber implements ApplicationListener<MyCustomEvent> {

    @Override
    public void onApplicationEvent(MyCustomEvent event) {
        System.out.println("Received: " + event.getMessage());
    }
}
```

Здесь у нас есть издатель, который отправляет MyCustomEvent, и подписчик, который этот MyCustomEvent получает и обрабатывает. Всё просто!

Почему это круто?

Этот паттерн добавляет еще один уровень гибкости к вашей архитектуре. Теперь у вас есть возможность динамически добавлять или удалять подписчиков, делая ваш код ещё более расширяемым.

Итак, Pub/Sub — это ещё один инструмент в вашем арсенале для построения надежных и масштабируемых приложений. Это особенно полезно, когда у вас есть несколько независимых модулей, которые должны реагировать на какие-то события, но при этом не должны знать друг о друге.

Singleton Security Context

Окей, говорили о паттернах интеграции, которые супер важны для обмена информацией между разными частями вашей системы. Теперь давайте переключим фокус на что-то, что каждый разработчик должен знать и уважать: безопасность. Особенно хочу рассмотреть паттерн Singleton Security Context.

Что это такое и зачем нужно?

Singleton Security Context — это паттерн безопасности, который организует хранение контекста безопасности в единственном экземпляре. В других словах, весь ваш код будет обращаться к одному и тому же «хранилищу» для проверки прав доступа, ролей и так далее. Это обеспечивает унификацию и согласованность данных о безопасности на всем протяжении жизненного цикла приложения.

Пример на Spring Security

В Spring Security вы можете увидеть этот паттерн в действии. Контекст безопасности хранится в SecurityContextHolder. Посмотрите:

```
import org.springframework.security.core.context.SecurityContextHolder;

public class SecurityService {
    public void performSecureAction() {
        var authentication =
SecurityContextHolder.getContext().getAuthentication();

        if (authentication != null &&
"ROLE_ADMIN".equals(authentication.getAuthorities())) {
            // Выполняем какое-то действие
        } else {
            throw new SecurityException("Недостаточно прав!");
        }
    }
}
```

Здесь SecurityContextHolder действует как Singleton, храня контекст безопасности. И каждый раз, когда нам нужно что-то проверить, мы просто обращаемся к нему.

Почему это так важно?

Возможно, вы думаете, что безопасность — это что-то, что можно добавить в конце разработки. Ошибка! Внедрение безопасности с самого начала может существенно упростить вашу жизнь. И этот паттерн — прекрасный способ сделать это эффективно и надёжно.

Так что Singleton Security Context не просто «еще один паттерн». Это ключевой элемент для построения безопасных приложений, и если вы ещё не использовали его в своих проектах на Spring, пора начать.

Sharding

Супер, мы поговорили о безопасности и как Singleton Security Context может быть полезным. Но что если ваше приложение разрастается как на дрожжах и теперь у вас тонны данных, которые нужно как-то эффективно хранить и обрабатывать? Вот тут на помощь приходят паттерны распределенных систем, а именно — Sharding.

Что это и зачем?

Sharding — это техника, при которой ваша база данных делится на меньшие, более управляемые части, называемые шардами. Каждый шард работает независимо, что позволяет улучшить производительность, масштабируемость и управляемость базы данных. В двух словах, это как класть книги не в одну огромную коробку, а распределить их по меньшим коробкам, чтобы было проще найти нужную.

Sharding на примере

Допустим, у вас есть приложение на Spring для онлайн-магазина, и вы используете базу данных для хранения информации о заказах. Вы решаете использовать Sharding, чтобы разделить данные по разным серверам на основе ID пользователя.

В Spring Data JPA это можно сделать с помощью @Sharded аннотации или кастомной настройки ShardResolver. Но честно говоря, это уже выходит за рамки Spring и требует специализированных баз данных, которые поддерживают шардинг на уровне движка.

```
// Код только для иллюстрации, Spring Data JPA не предоставляет нативную поддержку шардинга
public class OrderService {

    public void saveOrder(Order order) {
        // Определение шарда на основе ID пользователя
        String shard = shardResolver.resolveShard(order.getUserId());

        // Сохранение заказа в соответствующем шарде
        orderRepository.saveToShard(shard, order);
    }
}
```

Итак, почему это круто?

С шардингом вы можете легко масштабировать вашу базу данных горизонтально, просто добавляя новые шарды. Это особенно полезно, если у вас большие объемы данных и вы хотите сохранить высокую производительность. Всё это без изменения кода приложения!

Так что если вы работаете с большими данными или же стремитесь сделать свою систему максимально отзывчивой и быстрой, обязательно рассмотрите возможность использования шардинга.

Leader Election

А что, если в вашей распределенной системе нужно как-то координировать действия между различными узлами? Помните, мы говорили о шардинге, который отлично решает проблему масштабирования? Но масштабирование — это не единственная проблема в распределенных системах. Иногда нужно определить, какой из узлов будет отвечать за что-то важное. Здесь на помощь приходит паттерн Leader Election.

Суть паттерна

В системе с несколькими узлами Leader Election помогает автоматически выбрать один узел, который будет выполнять специфические задачи. Этот узел становится «лидером», а остальные узлы становятся «подчиненными» и выполняют действия согласно указаниям лидера.

Пример в контексте Spring

В Spring Cloud Cluster представлены абстракции для реализации Leader Election. Обычно этот паттерн активно используется в системах, работающих с консенсусными алгоритмами, такими как Paxos или Raft.

```
@Service
public class LeaderService implements SmartLifecycle {

    private boolean isRunning = false;

    @Autowired
    private LeaderInitiator leaderInitiator;

    @Override
    public void start() {
        isRunning = true;
        leaderInitiator.start(); // Инициация процесса выбора лидера
    }

    // ...
}
```

При успешном выборе лидера можно запускать определенные задачи, например, бэтч-обработку данных или управление распределенными транзакциями.

Понимание и использование Leader Election важно, потому что это помогает обеспечить согласованное и эффективное выполнение задач в распределенной системе. Вы как бы говорите: «Окей, ты сегодня босс, решай проблемы». Это очень полезно в сценариях, когда нужна высокая доступность и надежность.

Так что если вы создаете распределенные системы и хотите улучшить их стабильность и эффективность, паттерн Leader Election точно стоит вашего внимания.

Вывод

А вот и подошли к концу нашего путешествия по миру паттернов в контексте Spring. Кажется, как много всего мы обсудили, правда? И вы можете спросить: «Зачем мне все это нужно? Я же просто хочу кодить!» Давайте разберем, почему изучение паттернов проектирования и их применение в разработке на Spring — это не просто “nice to have”, а скорее «должен знать» для любого бэкенд разработчика.

1. **Эффективность:** Паттерны существуют не просто так. Они — это проверенные временем решения типичных задач, которые вам неизбежно придется решать. Зачем изобретать велосипед, если можно взять готовый и просто кататься?

2. **Читаемость и Поддержка:** Когда вы используете стандартные паттерны, другим разработчикам проще понять, что происходит в вашем коде. Это делает процесс поддержки и развития приложения намного легче.

3. **Карьерный Рост:** Знание паттернов открывает перед вами двери в более сложные и интересные проекты. Серьезно, на собеседованиях очень любят спрашивать про паттерны.

4. **Spring Love:** И последнее, но не менее важное — Spring уже построен на применении этих паттернов. Чем глубже вы погружаетесь в этот фреймворк, тем явнее становится, как все эти паттерны переплетены в его архитектуре.

Так что, как видите, паттерны — это не просто академическая тема для «умных разговоров», это реальные инструменты, которые сделают вашу жизнь проще и ваш код лучше. Если вы хотите быть в тренде, построить качественные и масштабируемые приложения на Spring, знание паттернов для вас — must have.

Заключение

Круто, ребята, мы с вами прошли через целую гору материала! От того, как строить Java приложения с Maven и Gradle, до тонкостей Spring Cloud и микросервисов. Научились даже тестировать всё это дело с JUnit и Mockito. Вкупе с навыками по Spring Security, Spring AOP, Spring Data и не только, вы теперь более чем подготовлены к реальной жизни бэкенд-разработчика.

Что мы получили?

1. **Практические Навыки:** Всё, что мы учили, напрямую применимо в вашей повседневной работе. Это не теория ради теории, это реальные инструменты и практики.
2. **Понимание Архитектуры:** Сложно переоценить, как важно понимать, что происходит “под капотом”. Знание паттернов и принципов проектирования дает вам возможность не просто копировать код, но и создавать что-то новое, эффективное и красивое.
3. **Безопасность и Масштабирование:** Мы рассмотрели, как защитить ваше приложение и как его масштабировать. Это ключевые навыки для любого современного разработчика.
4. **Тестирование и Мониторинг:** Стабильность и надежность — это то, что отличает хорошего разработчика от отличного. И вот тут наши уроки по JUnit, Mockito, Spring Actuator и интеграции с Prometheus и Grafana встают на стражу качества вашего кода.

Что Дальше?

Вы думаете, это все? Не-а, в бэкенд разработке всегда есть куда расти и что изучать. Интересные темы, такие как работа с большими данными, асинхронная обработка, машинное обучение и многое другое, ждут вас в будущем. И каждый раз, сталкиваясь с новыми вызовами, вы будете вспоминать, что проходили на этом курсе. Поверьте, это будет работать на вас, как хорошо отточенный инструмент.

Ребята, было круто провести с вами это время. Дерзайте, кодьте, расти и удачи во всех ваших начинаниях!

Что можно почитать еще?

1. Изучение Spring Boot 2.0. Грег Тернквист
2. Освоение Spring Boot 2.0. Динеш Раджпут

Используемая литература

1. Spring Boot в действии. Крейг Уоллс.
2. Spring Microservices в действии. Джон Карнелл
3. Cloud Native Java. Джош Лонг и Кенни Бастани