Занятие 5

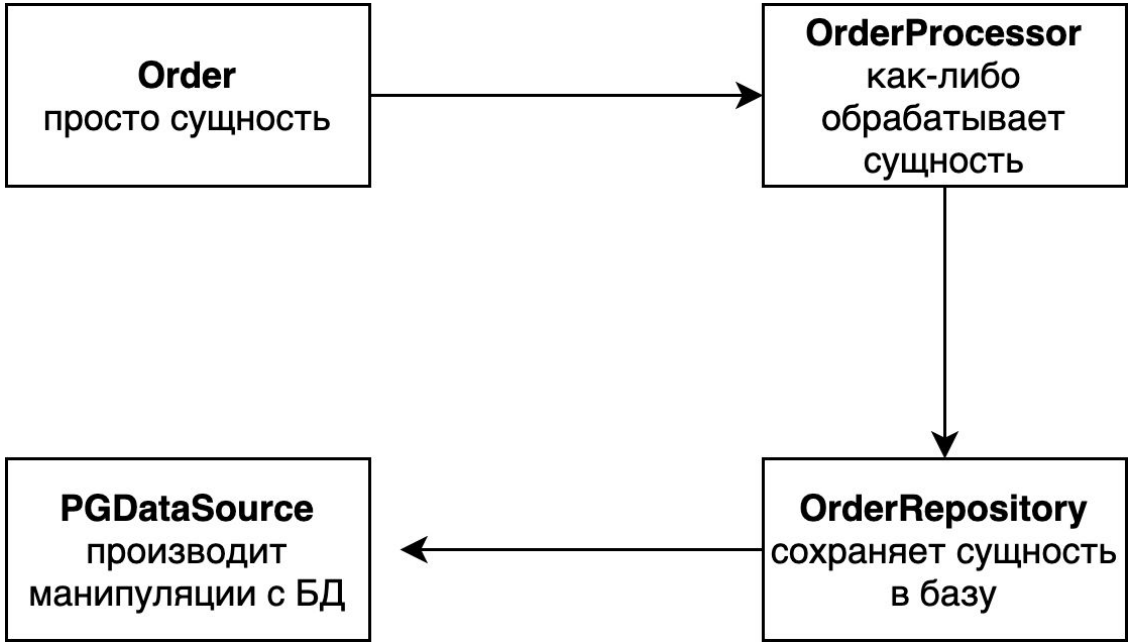
IoC & DI

Инверсия контроля (IoC)

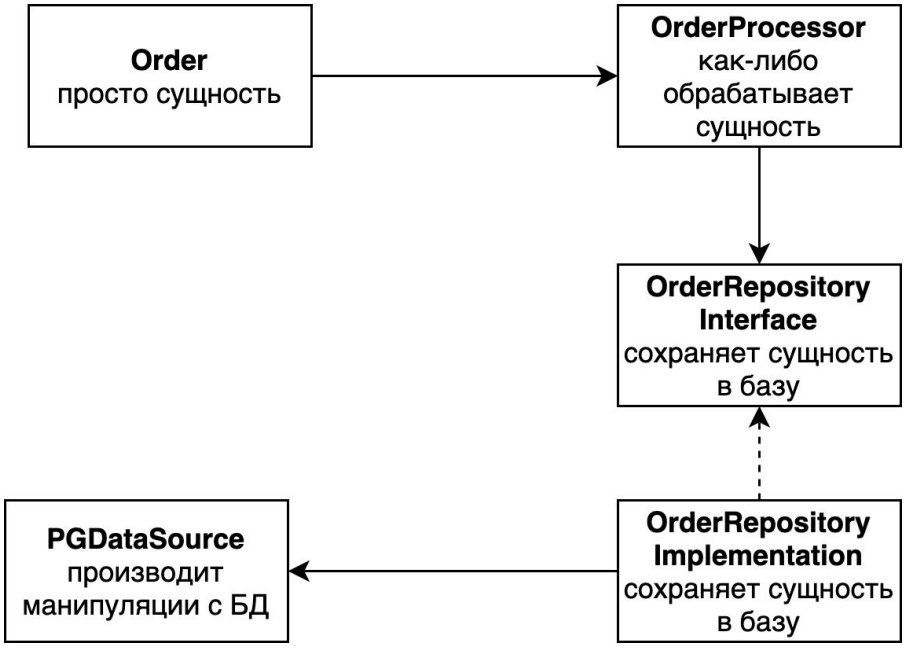
Нестрогое определение

Инверсия контроля - это не когда функции вызываются в нужной последовательности напрямую разработчиком, а реализовываются функции, которые в нужный момент вызываются фреймворком/библиотекой самостоятельно

Инверсия контроля (IoC)



Инверсия контроля (IoC)



Инверсия контроля (IoC)

1. Сущности верхнего уровня не должны зависеть от сущностей нижнего уровня. Они должны зависеть от абстракций

2. Абстракции не должны зависеть от конкретных деталей реализации

Варианты реализации IoC

1. Callbacks
2. Абстрактная фабрика
3. Внедрение зависимостей (DI) (here Spring comes)

DI

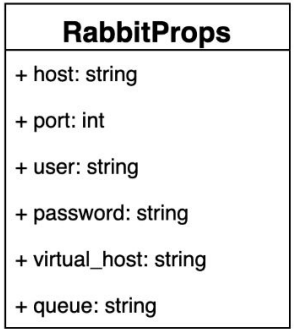
Внедрение зависимостей - это принцип, согласно которому объект пассивен и не предпринимает ничего для разрешения собственных зависимостей

Вместо этого, он формально описывает, какие зависимости ему требуются. А сами зависимости предоставляются кем-то другим. Этот кто-то другой может называться DI контейнером

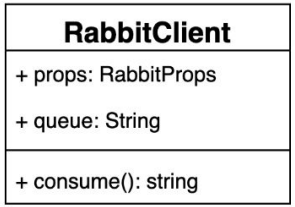
DI Example

Представим, что пишем приложение, который получает строки из RabbitMQ и сохраняет в БД. Представим, какие есть компоненты

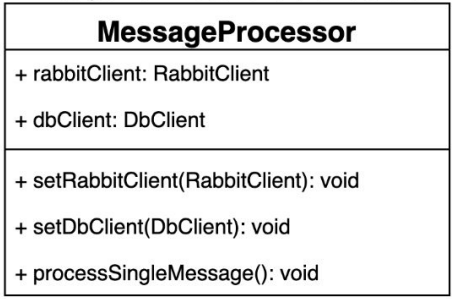
Компонент, в котором
хранятся настройки для подключения
к RabbitMQ

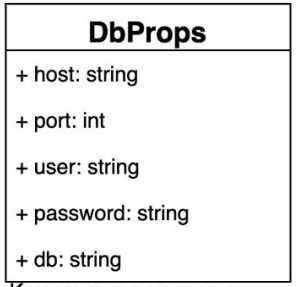


Компонент, который настраивает
подключение к RabbitMQ.
Умеет получать 1 сообщение из очереди



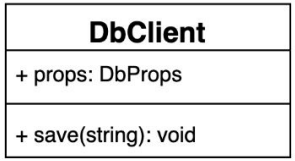
Компонент с бизнес логикой
При вызове метода **processSingleMessage**
получает 1 сообщение с помощью RabbitClient
и сохраняет в БД с помощью DbClient



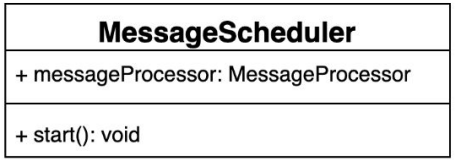


Компонент, в котором
хранятся настройки для подключения
к БД

Компонент, который настраивает
подключение к БД.
Умеет записывать 1 сообщение в БД



Запускает цикл,
время от времени вызывая
processSingleMessage



Как это все связывать?

Можно добавить еще один компонент! Конфигуратор. Появляется какой-то такой код

```
RabbitProps rabbitProps = …// читаем из файла

DbProps dbProps = … // читаем из файлы

DbClient dbClient = new DbClient();

RabbitClient rabbitClient = new RabbitClient();

MessageProcessor messageProcessor = new MessageProcessor();

MessageScheduler messageScheduler = new MessageScheduler();
```

Как это все связывать?

Далее необходимо написать код для связывания

```
dbClient.setProps(dbProps);

rabbitClient.setProps(rabbitProps);

messageProcessor.setDbClient(dbClient);

messageProcessor.setRabbitClient(rabbitClient);

messageSheduler.setMessageProcessor(messageProcessor);

messageScheduler.start(); // ура! запустились
```

DI контейнер

DI контейнер позволяет “свалить в кучу” все объекты, которые нужно создать, и затем выстроить связи автоматически. Более того, “в кучу” можно сваливать не объекты, а классы!
В таком случае DI контейнер

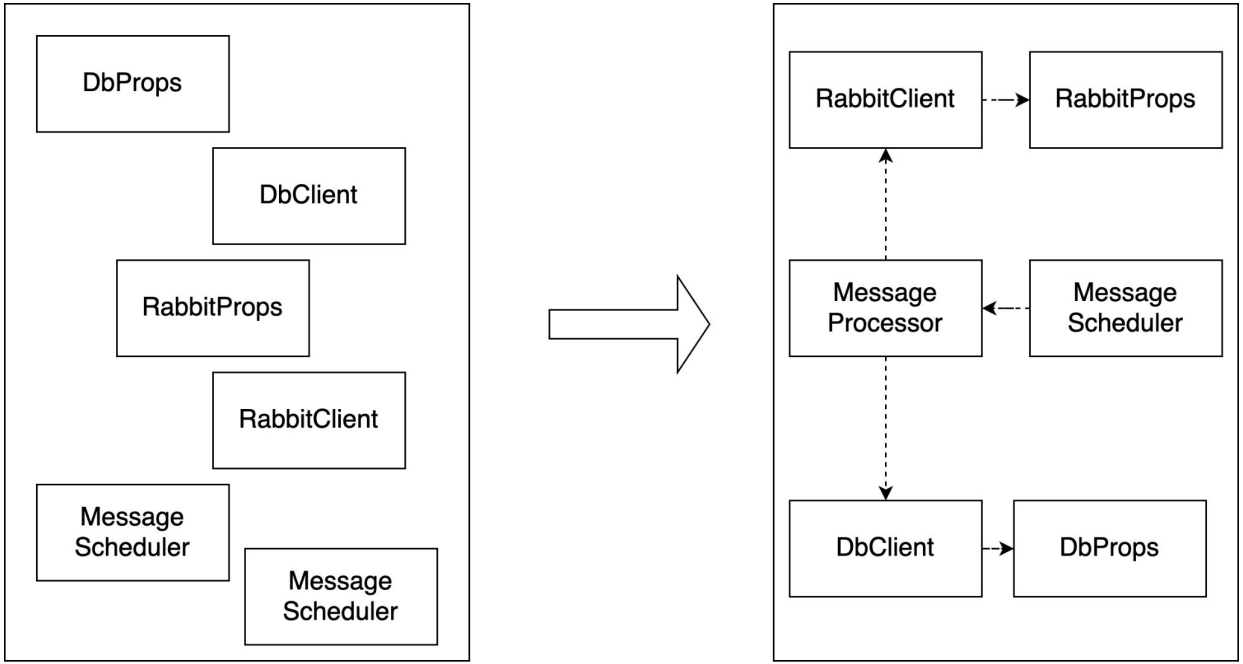
1. Создаст объекты нужных классов
2. Сам проставит зависимости между классами

DI контейнер

DI контейнер позволяет “свалить в кучу” все объекты, которые нужно создать, и затем выстроить связи автоматически. Более того, “в кучу” можно сваливать не объекты, а классы!
В таком случае DI контейнер

1. Создаст объекты нужных классов
2. Сам проставит зависимости между классами

DI контейнер



Spring

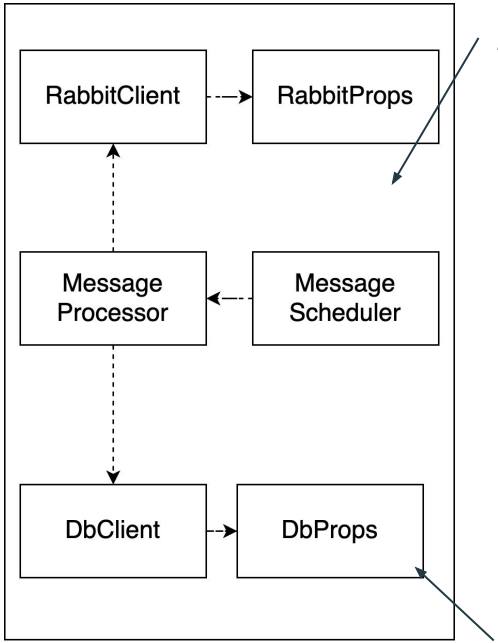
Spring - реализация DI контейнера, позволяющая гибко управлять жизненным циклом объектов

Альтернативы:

- Guice
- Jakarta Context and Dependency Injection
- PlayFramework
- Micronaut
- ...

Spring

Application Context



ApplicationContext - это создание и учет объектов внутри приложения

Bean - это объект, созданный и управляемый средствами Spring Framework

Основные этапы создания Spring приложения

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-context</artifactId>
  <version>5.2.12.RELEASE</version>
</dependency>
```

0. Добавление зависимостей или создание проекта с зависимостями
1. Описание классов объектов, которые будут созданы
2. Описание конфигурации, как и когда должны быть созданы объекты
3. Запуск конфигурации

Классы компонентов

```
package io.ylab.practice.spring;

@Component
public class Lightsaber {
  public void use() {
    System.out.println("Вжух вжух вжух (звуки размахивания световым мечом)");
  }
}
```

```
package io.ylab.practice.spring;

@Component
public class Jedi {
  @Autowired
  private Lightsaber lightsaber;

  public void fight() {
    lightsaber.use();
  }
}
```

`@Component` означает, что необходимо создать экземпляр класса и зарегистрировать его как bean

`@Autowired` означает, что найти bean класса **`Lightsaber`** и поместить его значение в поле `lightsaber`

Классы компонентов

```
package io.ylab.practice.spring;

@Component
public class Lightsaber {
  public void use() {
    System.out.println("Вжух вжух вжух (звуки размахивания световым мечом)");
  }
}
```

```
package io.ylab.practice.spring;

@Component
public class Jedi {
  @Autowired
  private Lightsaber lightsaber;

  public void fight() {
    lightsaber.use();
  }
}
```

`@Component` означает, что необходимо создать экземпляр класса и зарегистрировать его как bean

`@Autowired` означает, что найти bean класса **`Lightsaber`** и поместить его значение в поле `lightsaber`

Конфигурация

```
package io.ylab.practice.spring;

@Configuration
@ComponentScan("io.ylab.practice.lesson05.components")
public class Config {
}
```

Запуск!

```
public class SpringMain {
  public static void main(String[] args) {
    ApplicationContext applicationContext =
                  new AnnotationConfigApplicationContext(Config.class);
    Jedi jedi = applicationContext. getBean(Jedi.class);
    jedi.fight();
  }
}
```

```
> Вжух вжух вжух (звуки размахивания световым мечом)
```

Виды конфигураций: @Configuration vs @Bean

Можно убрать `@Component` на классах, при этом явно управляя способами создания объектов

```
@Configuration
public class Config {

  @Bean
  public Lightsaber lightsaber() {
    return new Lightsaber();
  }

  @Bean
  public Jedi jedi() {
    return new Jedi();
  }
}
```

Однако необходимо оставить `@Autwired` чтобы автоматически связать компоненты

```
@Configuration
public class Config {

  @Bean
  public Lightsaber lightsaber() {
    return new Lightsaber();
  }

  @Bean
  public Jedi jedi() {
    return new Jedi(lightsaber());
  }
}
```

Вариант без использования `@Autwired`. Однако, нужен конструктор или setter для инициализации

All @Configuration classes are subclassed at startup-time with CGLIB. In the subclass, the child method checks the container first for any cached (scoped) beans before it calls the parent method and creates a new instance.

Виды конфигураций: @Configuration vs @Bean

Можно убрать `@Component` на классах, при этом явно управляя способами создания объектов

```
@Configuration
public class Config {

  @Bean
  public Lightsaber lightsaber() {
    return new Lightsaber();
  }

  @Bean
  public Jedi jedi() {
    return new Jedi();
  }
}
```

```
@Configuration
public class Config {

  @Bean
  public Lightsaber lightsaber() {
    return new Lightsaber();
  }

  @Bean
  public Jedi jedi() {
    return new Jedi(lightsaber());
  }
}
```

All @Configuration classes are subclassed at startup-time with CGLIB. In the subclass, the child method checks the container first for any cached (scoped) beans before it calls the parent method and creates a new instance.

Однако необходимо оставить `@Autwired` чтобы автоматически связать компоненты

Вариант без использования `@Autwired`. Однако, нужен конструктор или setter для инициализации

@Autowired

Аннотация `@Autowired` может располагаться над конструктором (опционально с 4.3), над полями и над setter'ами

1. `Constructor`. Все зависимости проставляются в момент создания объекта, обеспечивая целостность. С другой стороны, если зависимостей много - конструктор станет очень большим. Но это может свидетельствовать о проблемах с архитектурой
2. `Setter`. Пустой конструктор, просто создавать "заглушки" для тестов, но необходимо писать setter для каждой зависимости
3. `Field`. Самый короткий и простой способ добавить инъектировать зависимость. Но сложно тестировать, сложно добавлять "заглушки" для тестов

Обычно рекомендуется использовать инъекции через конструктор (но это не точно)

Live Coding Section

Вариант использования setter injection для регистрации Converter в ConverterRegistry

Квалификаторы и @Autowired

```
public interface Animal {
  void makeSound();
}
```

```
@Component
public class Cat implements Animal{
  @Override
  public void makeSound() {
    System.out.println("Мяу мяу мяу");
  }
}

@Component
public class Dog implements Animal {
  @Override
  public void makeSound() {
    System.out.println("Гав Гав Гав");
  }
}

@Component
public class Hedgehog implements Animal {
  @Override
  public void makeSound() {
    System.out.println("Пых пых пых");
  }
}
```

Квалификаторы и @Autowired

```
@Component
public class AnimalOwner {

  @Autowired
  private Animal animal;

  public void poke() {
    System.out.println("Тыкаем пальцем питомца. А он такой:");
    animal.makeSound();
  }
}
```

1. Что будет инжектировано?
2. Как управлять тем, что будет инжектировано?

```
@Autowired
@Qualifier("hedgehog")
private Animal animal;
```

Будет вставлен ежик. Потому что бин типа `Hedgehog` неявно получает имя (квалификатор) **`hedgehog`**

Квалификаторы и @Autowired

```
@Component
public class AnimalOwner {

  @Autowired
  @Qualifier("Sonic")
  private Animal animal;

  public void poke() {
    System.out.println("Тыкаем пальцем питомца");
    animal.makeSound();
  }
}
```

```
@Component("Sonic")
public class Hedgehog implements Animal
```

```
@Component
@Qualifier("Sonic")
public class Hedgehog implements Animal
```

Обзор домашнего задания