Лекция 6
Разработка АС реального времени (часть 2)

ФИО преподавателя: Зорина Наталья Валентиновна

e-mail: zorina_n@mail.ru

Тема лекции:

«Реализация REST API с помощью Spring Framework»

Внедрение зависимостей

Определение

- **Внедрение зависимостей** — это стиль настройки объекта, при котором поля объекта задаются внешней сущностью.
- Другими словами, объекты настраиваются **внешними объектами**. **DI** — это альтернатива **самонастройке** объектов.

Проблема ☹

```
public static void main(String[] args) {
    System.out.println(getClass4());
}

public static Class4 getClass4(){
    Class1 class1=new Class1( someString: "class1");
    Class2 class2=new Class2( someString: "class2");
    Class3 class3=new Class3(class1,class2);
    Class1 class11=new Class1( someString: "clas11");
    return new Class4(class3,class11);
}
```

А если переменные приходят из вне?
А если классов больше например 50?
Как это поддерживать?

Решение ☺

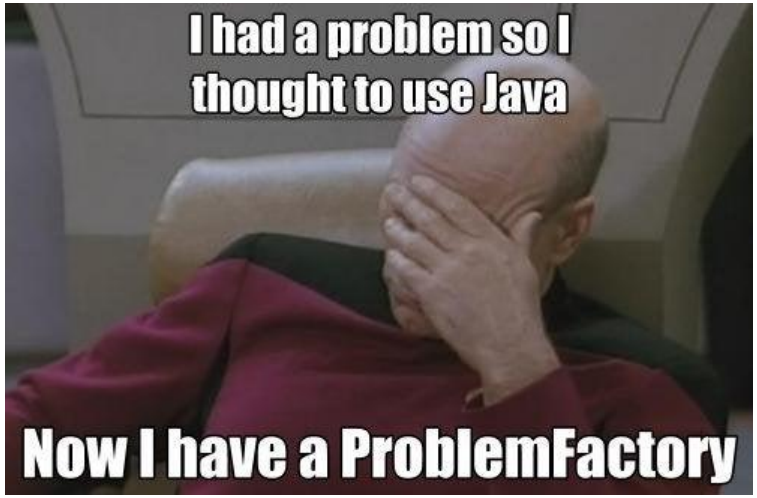
- Давайте воспользуемся фабрикой

```
public class Class1Factory {

    public Class1 getClass1(String classVar) { return new Class1(classVar); }
}
```

Проблема ☹

Теперь вместо кучи объектов мы создаем кучу фабрик



Решение ☺

```
    private static ObjectFactory ourInstance = new ObjectFactory();
    private Config config;

    public static ObjectFactory getInstance() { return ourInstance; }

    private ObjectFactory() {
        config = new JavaConfig( packageToScan: "com.sevod", new HashMap<>(Map.of(Policeman.class, A
    }


    @SneakyThrows
    public <T> T createObject(Class<T> type) throws FileNotFoundException, IllegalAccessExceptio
        Class<? extends T> implClass = type;
        if (type.isInterface()) {
            implClass = config.getImpClass(type);
        }

        T t = implClass.getDeclaredConstructor().newInstance();

        return t;
    }

}
```

Паттерн ObjectFactory
Идея: создание объекта в отрыве от его реализации на основе конфига

Структура Java конфигуратора

```java
public class JavaConfig implements Config {

    private Reflections scanner;
    private Map<Class, Class> ifc2ImplClass;

    public JavaConfig(String packageToScan, Map<Class, Class> ifc2ImplClass) {
        this.ifc2ImplClass = ifc2ImplClass;
        this.scanner = new Reflections(packageToScan);
    }

    public <T> Class<? extends T> getImpClass(Class<T> ifc) {
        return ifc2ImplClass.computeIfAbsent(ifc, aClass -> {
            Set<Class<? extends T>> classes = scanner.getSubTypesOf(ifc);
            if (classes.size() != 1) {
                throw new RuntimeException(ifc + " has 0 or more than one impl please update your config");
            }
            return classes.iterator().next();
        });
```

Идея: сканирование пакетов

Анатомия jar

- **Jar** это **zip** архив специального вида!
- Внутри jar находятся **байт код программы**, **ресурсы** а также сохранена переменная **classpath** для данной программы.
- **Напоминание!** Classpath это переменная в которой указаны пути до всех классов программы и самое главное мэин класс.

Сканирование пакетов

- Одной из ключевых особенностей JVM является **динамическая загрузка классов**. Достигается это благодаря сущности **загрузчика классов**.
- Идея состоит в том что мы можем двигаться по классам переменной classpath и загружать их при помощи classloader. Загружая классы являющие **подттипом** данного что реализует библиотека **reflections**.

Настройка объектов

```
leakyThrows
blic <T> T createObject(Class<T> type) throws FileNotFoundException, IllegalAccessException, NoSuchMetho
 Class<? extends T> implClass = type;
 if (type.isInterface()) {
     implClass = config.getImpClass(type);
 }

 T t = implClass.getDeclaredConstructor().newInstance();

 for (Field field : implClass.getDeclaredFields()) {
     InjectProperty annotation = field.getAnnotation(InjectProperty.class);
     String path = ClassLoader.getSystemClassLoader().getResource( name: "application.properties")
             .getPath();

     Stream<String> lines = new BufferedReader(new FileReader(path)).lines();

     Map<String, String> propertiesMap = lines.map(line -> line.split( regex: "="))
             .collect(toMap(arr -> arr[0], arr -> arr[1]));
     if (annotation != null) {
         String value = annotation.value().isEmpty() ? propertiesMap.get(field.getName()) :
                 propertiesMap.get(annotation.value());
         field.setAccessible(true);
         field.set(t, value);
     }
 }
 return t;
```

1)Создавать объекты мы умеем теперь хочется уметь их **настраивать**.
2)Хотим мы например чтобы у нас из **проперти файлов** заполнялись свойства классов **аннотированные InjectProperty**.
3)**Изменим** исходный код нашей фабрики.

Больше аннотаций

- Итак мы умеем **внедрять** свойство в объект. Этого достаточно для реализации концепта паттерна **инъекции зависимостей**. Только вместо свойств мы будем внедрять объекты по сигнатуре их **типа.** Мы можем написать код который будет внедрять объекты по их типу. *Оставляю данную задачу слушателем как упражнение :-).*
- Но есть **проблема** когда разных аннотаций станет много у нас получится раздутая фабрика объектов и любая правка будет выглядеть так: исправь пожалуйста вызов на 100500 строчке метода createObject. Корень этой проблемы в том что мы нарушили **Single Responsibility и Open Close** принципы. Исправим это.

Решение ☺

- Нашу проблему решит паттерн **цепочка обязанностей**. Определим класс конфигуратора с методом конфигурации. И **добавим в фабрику** вызов цепочки конфигураторов для создаваемого объекта.

```
private <T> void configure(T t) {
    configurators.forEach(objectConfigurator -> objectConfigurator.configure(t,context));
}
```

Решение ☺

- **Вынесем** аннотацию внедрения свойства в **конфигуратор**.

```
public class InjectPropertyAnnotationObjectConfigurator implements ObjectConfigurator {

    private Map<String, String> propertiesMap;

    @SneakyThrows
    public InjectPropertyAnnotationObjectConfigurator() {
        String path = ClassLoader.getSystemClassLoader().getResource( name: "application.properties").getPath();
        Stream<String> lines = new BufferedReader(new FileReader(path)).lines();
        propertiesMap = lines.map(line -> line.split( regex: "=")).collect(toMap(arr -> arr[0], arr -> arr[1]));

    }

    @Override
    @SneakyThrows
    public void configure(Object t,ApplicationContext context) {
        Class<?> implClass = t.getClass();
        for (Field field : implClass.getDeclaredFields()) {
            InjectProperty annotation = field.getAnnotation(InjectProperty.class);
           if (annotation != null) {
                String value = annotation.value().isEmpty() ? propertiesMap.get(field.getName()) :
                        propertiesMap.get(annotation.value());
                field.setAccessible(true);
                field.set(t,value);
            }
        }
    }
}
```

Проблема ☹

- Итак проблему **управления зависимости** мы **решили**.
- Однако в наших проектах есть большое количество **шаблонной функциональности** которую нам хотелось **обобщить**. Например открытие закрытие транзакций или работа с http запросами.

Решение ☺

- На помощь приходит паттерн **динамическое прокси**.
- Динамическое прокси это класс прокладка который имеет точно такую же структуру как и вызываемый класс и вызывает оригинальные методы класса но с добавлением дополнительной функциональности.
- Такой подход стал развитием ООП и называется **Аспектно Ориентированным Программированием (АОП)**.
- Для реализации **добавим** еще один **тип конфигураторов** которые добавляют такие прокси классы к оригинальной реализации.

Решение ☺

Добавим еще одну цепочку обязанностей в нашу фабрику.

```
private <T> T wrapWithProxyIfNeeded(Class<T> implClass, T t) {
    for (ProxyConfigurator proxyConfigurator : proxyConfigurators) {
        t = (T) proxyConfigurator.replaceWithProxyIfNeeded(t, implClass);
    }
    return t;
}
```

Решение ☺

Пример реализации прокси

```
@Override
public Object replaceWithProxyIfNeeded(Object t, Class implClass) {
    //todo make support for @Deprecate above methods, not class
    if (implClass.isAnnotationPresent(Deprecated.class)) {

        if (implClass.getInterfaces().length == 0) {
            return Enhancer.create(implClass, new net.sf.cglib.proxy.InvocationHandler() {
                @Override
                public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
                    return getInvocationHandlerLogic(method, args, t);
                }
            });
        }


        return Proxy.newProxyInstance(implClass.getClassLoader(), implClass.getInterfaces(), new InvocationHandler() {
            @Override
            public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
                return getInvocationHandlerLogic(method, args, t);
            }
        });
    } else {
        return t;
    }

}
```

Последний элемент

- Итак мы умеем **создавать и настраивать объекты** разными методами. Однако нам не хватает какой-то структуры где все эти объекты будут лежать и откуда **жизненным циклом** этих объектов можно будет **управлять**.
- Таким объектом является **ApplicationContext**. Если просто то это **фабрика** плюс **хэш мапа** в которой **лежат** объекты и **выдаются** по запросу с учетом **конфигурационных аннотаций** позволяя определять объект как **синглтон** или как **прототип**.

Пример реализации application context

```java
public class ApplicationContext {
    @Setter
    private ObjectFactory factory;
    private Map<Class, Object> cache = new ConcurrentHashMap<>();
    @Getter
    private Config config;

    public ApplicationContext(Config config) { this.config = config; }

    public <T> T getObject(Class<T> type) {
        if (cache.containsKey(type)) {
            return (T) cache.get(type);
        }

        Class<? extends T> implClass = type;

        if (type.isInterface()) {
            implClass = config.getImplClass(type);
        }
        T t = factory.createObject(implClass);

        if (implClass.isAnnotationPresent(Singleton.class)) {
            cache.put(type, t);
        }

        return t;
    }
}
```

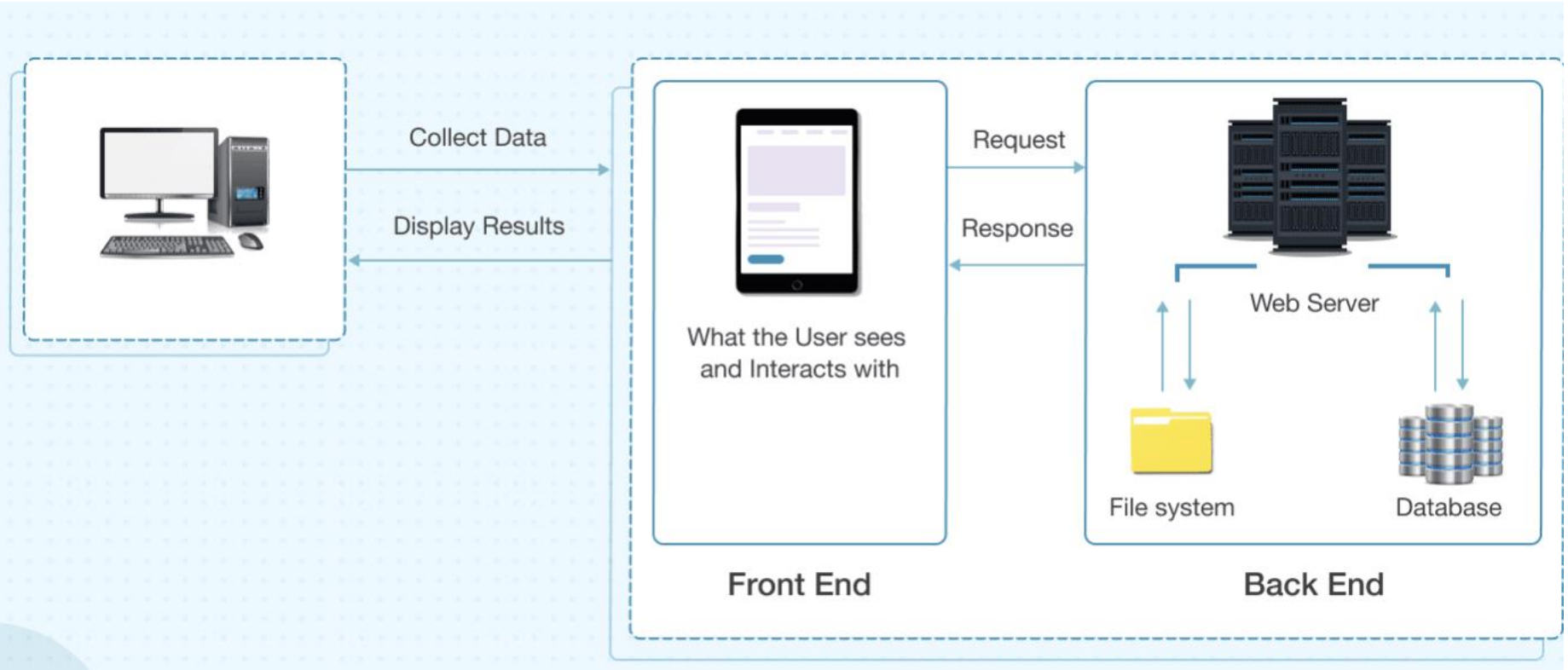
И зачем все это?

- Ключевая проблема которую решает данный подход это управление зависимостями.
- На первый взгляд кажется что это **гиперусложнение**, но такой подход оправдывает себя в **больших** и **сложных проектах** где очень **много зависимостей** и где управлять ими **очень сложно**.

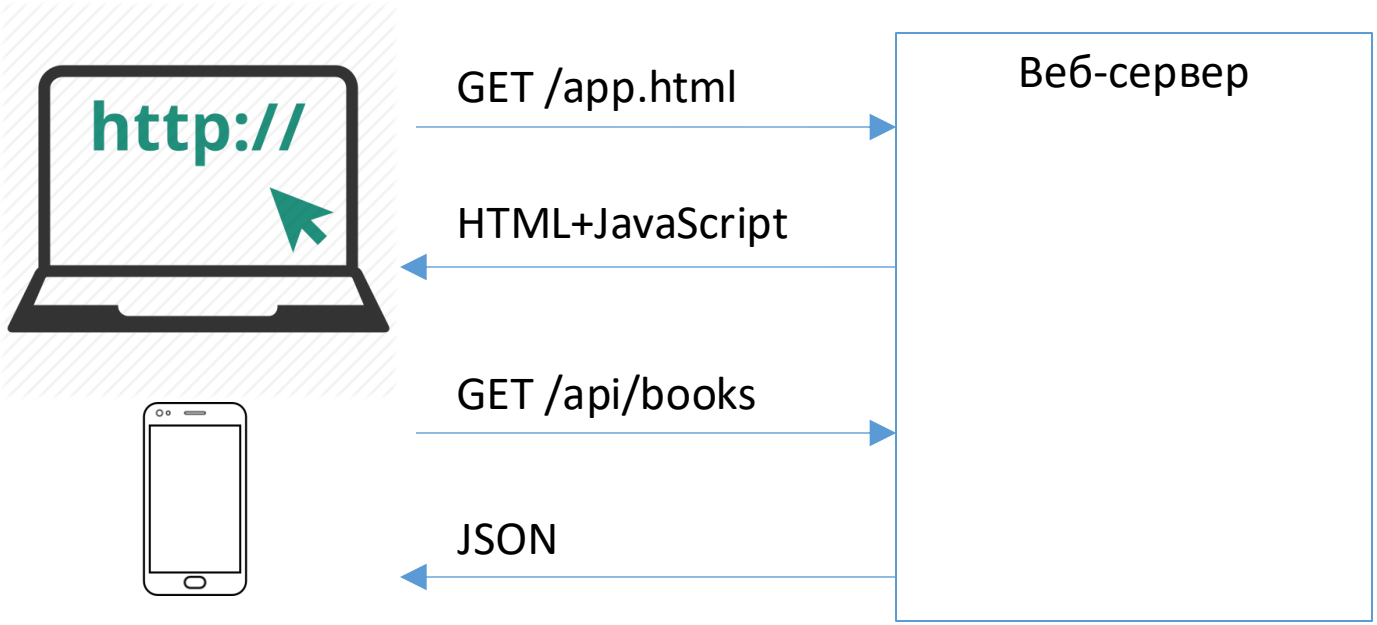
Выводы

- Мы написали свой маленький Spring☺

Архитектура веб-приложений



Архитектура веб-приложений



Архитектура веб-приложений

Пример документа JSON:

```
{
  "books": [
    { "id": 11, "author": "Лев Толстой", "title": "Война и мир" },
    { "id": 15, "author": "Алан Милн", "title": "Винни-пух" },
    { "id": 17, "author": "Дж. Роулинг", "title": "Гарри Поттер" }
  ],
  "totalBooks": 20,
  "page": 1,
  "pageSize": 3
}
```

Архитектура веб-приложений

Задача веб-сервера – выдать данные для отображения в браузере (обычно в формате JSON).

Эти данные обрабатываются отдельным frontend приложением:

- в браузере на языке JavaScript
- в мобильном приложении:
  - Android: Java/Kotlin
  - iPhone: ObjectiveC/Swift

Архитектура веб-приложений

REST – стиль взаимодействия клиента с сервером. Обычно он подразумевает запросы и ответы в формате JSON, где адрес запроса содержит информацию о том, что хочет сделать клиент:

- GET /api/books – получить список всех книг
- GET /api/books/11 – получить информацию о книге с id=11
- POST /api/books – создание записи о книге; данные о создаваемой книге передаются в теле запроса
- PUT /api/books/15 – обновление записи о книге с id=15; новые данные о книге передаются в теле запроса

Архитектура веб-приложений

Документ JSON не формируется вручную, вместо этого используются библиотеки для преобразования между объектами Java и документами JSON:

- Jackson
- GSON

Сериализация: объект Java -> JSON

Десериализация: JSON -> объект Java

JSON – Java Script Object Notation- — это стандартный текстовый формат для представления структурированных данных на основе синтаксиса объектов JavaScript

Архитектура веб-приложений

```
public class Book {

   private final int id;
   private final String author;
   private final String title;

   public Book(int id, String author, String title) {
      this.id = id;
      this.author = author;
      this.title = title;
   }

   public int getId() { return id; }

   public String getAuthor() { return author; }

   public String getTitle() { return title; }
}
```

Архитектура веб-приложений

```
public class BookSearchResult {

   private final int totalBooks;
   private final int page;
   private final int pageSize;
   private final List<Book> books;

   public BookSearchResult(int totalBooks, int page, int pageSize, List<Book> books) {
      this.totalBooks = totalBooks;
      this.page = page;
      this.pageSize = pageSize;
      this.books = books;
   }

   public int getTotalBooks() { return totalBooks; }

   public int getPage() { return page; }

   public int getPageSize() { return pageSize; }

   public List<Book> getBooks() { return books; }
}
```

Архитектура веб-приложений

```
import com.fasterxml.jackson.core.JsonProcessingException;
import com.fasterxml.jackson.databind.ObjectMapper;
import com.fasterxml.jackson.module.paramnames.ParameterNamesModule;

public class JacksonExample {

    public static void main(String[] args) throws JsonProcessingException {
        ObjectMapper mapper = new ObjectMapper();
        mapper.registerModule(new ParameterNamesModule());
        String json = mapper.writeValueAsString(new Book(11, "Лев Толстой", "Война и мир"));
        System.out.println(json);
        Book book = mapper.readValue(json, Book.class);
        System.out.println(book.getTitle());
    }
}
```

Внимание! Чтобы этот код работал, класс Book должен быть скомпилирован с параметром компилятора javac "-parameters", чтобы Jackson мог сопоставить имена полей в JSON с параметрами конструктора класса Book

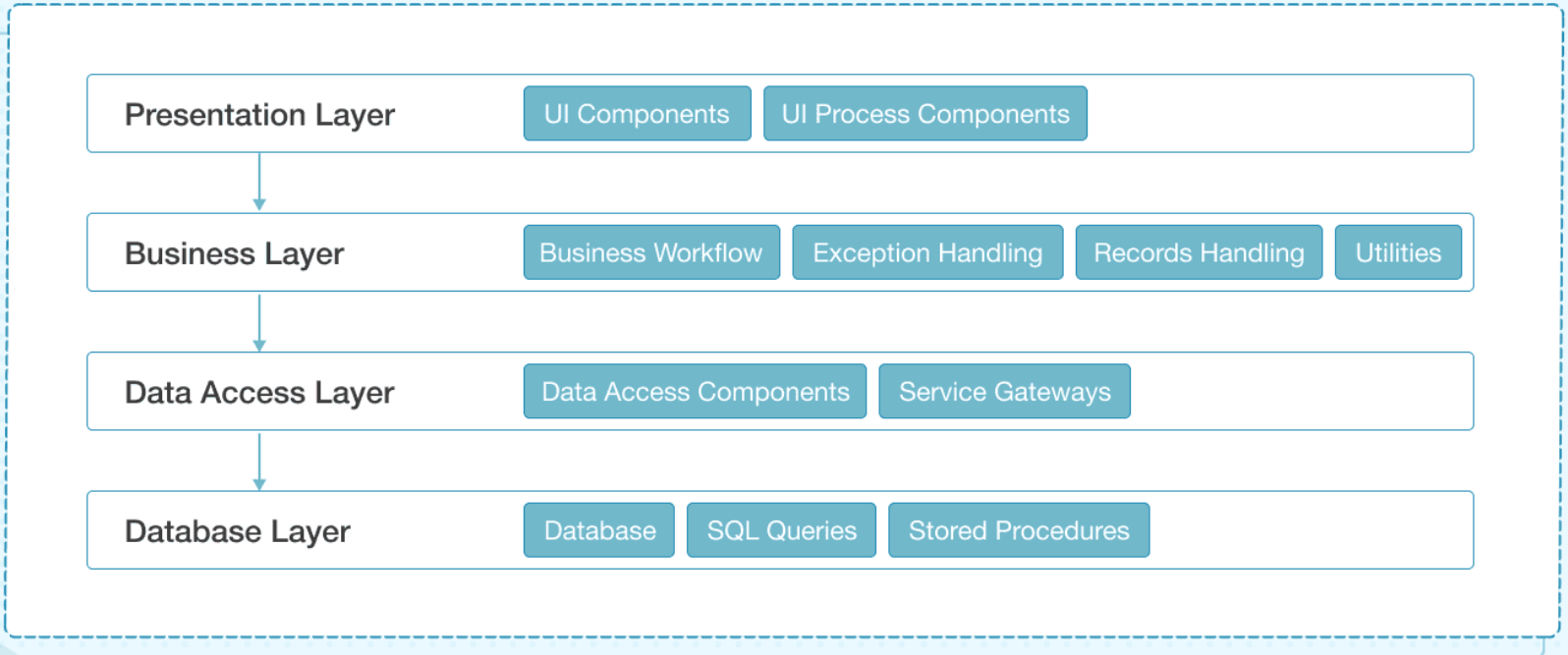
Учебник библиотека Джексон https://www.baeldung.com/jackson

Архитектура веб-приложений

Итого задача серверной части веб-приложения при обработке запроса:

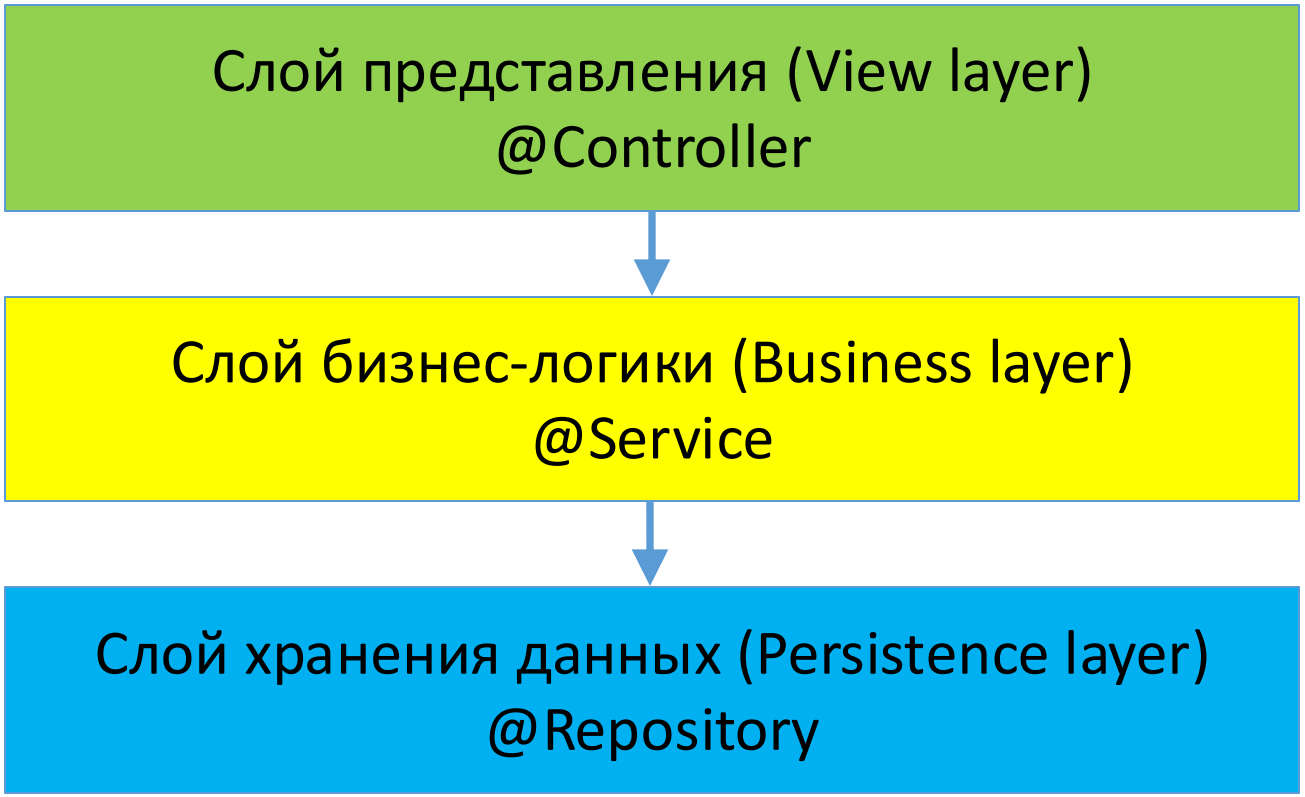
1. Понять, какой запрос пришел (например, запрос /api/books для получения списка всех книг)
2. Обратиться к базе данных (или другому источнику данных) для получения запрошенных данных в виде обычных Java-объектов (в нашем примере списка книг)
3. Сериализовать Java-объекты в формат JSON и отправить в качестве ответа на запрос

Архитектура веб-приложений



SIMFORM

Архитектура веб-приложений



Архитектура веб-приложений

1. Слой представления отвечает за взаимодействие с клиентом: сериализацию/десериализацию JSON, маршрутизацию (routing) – определение, какое действие было запрошено (/api/books или /api/books/{id})
2. Слой бизнес-логики отвечает за логику работы приложения (в простейшем случае просто перенаправление запроса к слою хранения данных)
3. Слой хранения данных отвечает за взаимодействие с базой данных: получение данных из БД и сохранение данных в БД (в случае запроса /api/books – получение из БД списка книг)

Архитектура веб-приложений

Взаимодействие между слоями может быть только сверху вниз, т.е. слой представления может обращаться к слою бизнес-логики, но не наоборот.

Нельзя "перепрыгивать" через слои, т.е. слой представления не может обращаться напрямую к слою хранения данных.

Архитектура веб-приложений

Базой для слоя представления в веб-приложениях является веб-сервер:

- Tomcat
- Jetty
- Netty
- Undertow

Веб-сервер осуществляет всю низкоуровневую работу с протоколом HTTP. Все эти серверы поддерживают Servlet API.

Архитектура веб-приложений

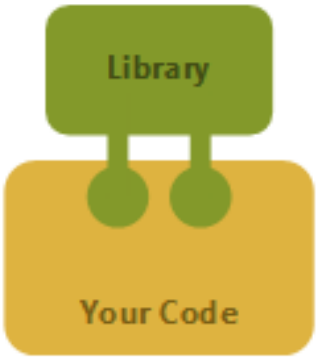
Поверх веб-сервера часто используется фреймворк, упрощающий рутинную работу:

- Spring/Spring Boot
- JakartaEE
- Quarkus
- Micronaut
- Dropwizard
- Play
- Javalin
- Jooby
- Vert.X

Архитектура веб-приложений

Отличие фреймворка от библиотеки

- Фреймворк вызывает наш код
- Наш код вызывает библиотеку



Фреймворк или библиотека. Что в итоге?

- фреймворки и библиотеки - это код, написанный кем-то другим, который решает некоторые общие задачи, не утруждая вас реализацией этого решения.
- Фреймворк инвертирует управление программой и говорит программисту, что ему нужно.
- Библиотека не вмешивается в поток программы. Ее методы можно вызывать только тогда, когда они нужны.

Архитектура веб-приложений

Для работы слоя хранения данных используются:

- JDBC
- Hibernate
- JDBI
- jOOQ
- Spring JDBC
- Spring Data

Пример REST API на Spring

Spring Framework

Плюсы:

- Магия
  - Код для выполнения многих стандартных задач не надо писать, Spring генерирует его автоматически

Минусы:

- Магия
  - Если что-то идет не так, сложно понять почему

Пример REST API на Spring

Spring Boot – расширение Spring Framework, предоставляющее конфигурации для решения стандартных задач, таких как веб-приложения.

Включает в себя:

- предустановленный набор библиотек
- встроенный веб-сервер
- средства мониторинга приложения

Пример REST API на Spring

```
plugins {
    id("java")
    id("org.springframework.boot") version "2.4.5"
}

group = "ru.mirea"
version = "0.1"

repositories {
    mavenCentral()
}

compileJava.options.compilerArgs << "-parameters"

dependencies {
    implementation("org.springframework.boot:spring-boot-starter-web:2.4.5")
}
```

Пример REST API на Spring

```
package ru.mirea.hello;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class RestApp {

    public static void main(String[] args) {
        SpringApplication.run(RestApp.class, args);
    }
}
```

Пример REST API на Spring

```
package ru.mirea.hello;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping(produces = "application/json")
public class RestHello {

    @GetMapping("/hello")
    public String hello(@RequestParam(defaultValue = "world") String name) {
        return String.format("Hello, %s!", name);
    }
}
```

Пример REST API на Spring

http://localhost:8080/hello?name=Spring

```
Hello, Spring!
```

Пример REST API на Spring

- логика работы во многом задается аннотациями
- главный класс RestApp никак не ссылается на класс RestHello
- используется порт по умолчанию 8080

Пример REST API на Spring

Аннотации @RestController:

- @GetMapping/@PostMapping/@PutMapping/@DeleteMapping (частные случаи @RequestMapping)
- @RequestParam – описывает параметр, значение которого берется из строки запроса (?name=...)
- @PathVariable – описывает параметр, значение которого берется из пути запроса

Пример использования @PathVariable:

```
@GetMapping("/api/books/{bookId}")
public Book getBook(@PathVariable("bookId") int bookId)
```

Пример REST API на Spring

Исходный код

```
@RestController
@RequestMapping(path = "/api/books", produces = "application/json")
public class BookRest {

   private final List<Book> books = new CopyOnWriteArrayList<>();
   private final AtomicInteger idGenerator = new AtomicInteger();

   @GetMapping
   public List<Book> getAllBooks() {
      return books;
   }

   @GetMapping("/{id}")
   public Book getBook(@PathVariable("id") int id) {
      for (Book book : books) {
         if (book.getId() == id) {
            return book;
         }
      }
      throw new ResponseStatusException(HttpStatus.NOT_FOUND);
   }
```

Обрабатывает запросы вида /api/books/{id}
Полный путь – комбинация аннотаций класса ("/api/books") и метода ("/{id}")

Пример REST API на Spring

```
@PostMapping(consumes = "application/json")
public Book addBook(@RequestBody BookDetails details) {
   if (details.getAuthor() == null || details.getTitle() == null)
      throw new ResponseStatusException(HttpStatus.BAD_REQUEST);
   int id = idGenerator.addAndGet(1);
   Book book = new Book(id, details.getAuthor(), details.getTitle());
   books.add(book);
   return book;
}
```

```
public class BookDetails {

   private final String author;
   private final String title;

   public BookDetails(String author, String title) {
      this.author = author;
      this.title = title;
   }

   public String getAuthor() { return author; }

   public String getTitle() { return title; }
}
```

Пример REST API на Spring

- для счетчика книг мы используем AtomicInteger, так как к нашему веб-серверу могут обращаться одновременно много клиентов
- параметры создаваемой книги мы передаем в теле запроса POST (@RequestBody) в формате JSON (consumes="application/json")
- мы проверяем корректность параметров и выдаем ошибку 400, если они заданы некорректно

Пример REST API на Spring

Как тестировать REST API?

- встроенный в IntelliJ IDEA клиент HTTP
- curl – утилита командной строки
- Postman

Пример REST API на Spring

Пример сессии в IntelliJ IDEA:

```
###
GET http://localhost:8080/api/books

###
POST http://localhost:8080/api/books
Content-Type: application/json

{"author": "Лев Толстой", "title": "Война и жир"}

###
GET http://localhost:8080/api/books/1
```

Пример REST API на Spring

Ответы нашего сервиса на запросы:

GET http://localhost:8080/api/books

HTTP/1.1 200
Content-Type: application/json

[]

Response code: 200; Time: 11ms; Content length: 2 bytes

Пример REST API на Spring

Ответы нашего сервиса на запросы:

```
POST http://localhost:8080/api/books
Content-Type: application/json

{"author": "Лев Толстой", "title": "Война и жир"}

HTTP/1.1 200
Content-Type: application/json

{
  "id": 1,
  "author": "Лев Толстой",
  "title": "Война и жир"
}

Response code: 200; Time: 71ms; Content length: 53 bytes
```

Пример REST API на Spring

```
@PutMapping(path = "/{id}", consumes = "application/json")
public Book updateBook(@PathVariable("id") int id, @RequestBody BookDetails details) {
   if (details.getAuthor() == null || details.getTitle() == null)
      throw new ResponseStatusException(HttpStatus.BAD_REQUEST);
   for (int i = 0; i < books.size(); i++) {
      Book book = books.get(i);
      if (book.getId() == id) {
         Book newBook = new Book(book.getId(), details.getAuthor(), details.getTitle());
         books.set(i, newBook);
         return newBook;
      }
   }
   throw new ResponseStatusException(HttpStatus.NOT_FOUND);
}

@DeleteMapping("/{id}")
public void deleteBook(@PathVariable("id") int id) {
   books.removeIf(book -> book.getId() == id);
}
```

Пример REST API на Spring

Ответы нашего сервиса на запросы:

```
PUT http://localhost:8080/api/books/1
Content-Type: application/json

{"author": "Лев Толстой", "title": "Война и мир"}

HTTP/1.1 200
Content-Type: application/json

{
  "id": 1,
  "author": "Лев Толстой",
  "title": "Война и мир"
}

Response code: 200; Time: 80ms; Content length: 53 bytes
```

Пример REST API на Spring

Как Spring понимает, что нужно использовать класс RestHello или BookRest?

Для этого используется сканирование доступных классов на наличие у них аннотаций:

- @Component
- @Controller, @RestController
- @Service
- @Repository
- @Configuration

Пример REST API на Spring

Так как наша программа может содержать очень много классов, то как правило сканируются не все они, а только классы из выбранного пакета (и его подпакетов). По умолчанию аннотация @SpringBootApplication сканирует пакет, в котором находится главный класс приложения (плюс подпакеты). В примере RestHello это ru.mirea.hello. При необходимости можно задать список пакетов явно:

```
@SpringBootApplication(scanBasePackages = {"ru.mirea.books"})
```

Пример REST API на Spring

Для найденных классов создаются их экземпляры ("beans") и они регистрируются в системе. Для классов @RestController при этом для указанных путей автоматически создаются точки входа в веб-сервере.

Пример REST API на Spring

Как реализовано сканирование классов?

Классы Java загружаются с помощью механизма загрузчика классов – ClassLoader. У каждого класса есть classloader, который его загрузил. Этот механизм доступен в коде Java:

```
ClassLoader cl = "Hello".getClass().getClassLoader();
```

Как правило, classloader'ы загружают классы из файлов .class, находящихся на диске или внутри архива .jar.

Пример REST API на Spring

Кроме классов, classloader'ы позволяют загружать ресурсы – произвольные файлы, которые нужны нашему приложению (например, картинки или конфигурационные файлы). У класса ClassLoader есть метод getResources(), позволяющий получить ресурс по пути к нему:

```
ClassLoader cl = ...;

Enumeration<URL> resources =
    cl.getResources("ru/mirea/hello/RestHello.class");
```

Так мы получим доступ к байт-коду класса RestHello

Пример REST API на Spring

Если имя ресурса – не имя файла, и имя папки, мы получим доступ ко всей папке:

```
ClassLoader cl = ...;

Enumeration<URL> resources =
    cl.getResources("ru/mirea/hello/");
```

Далее мы можем, исходя из того, на что указывают URL, сканировать:

- если это file://..., то сканируем дерево файлов
- если это jar://..., то сканируем архив

Код примера: https://github.com/osobolev/restdemo/blob/master/src/main/java/ru/mirea/my_di/ClasspathScanner.java

Пример REST API на Spring

Пока архитектура нашего проекта неправильная. Разделим его на слой представления и слой бизнес-логики.

Пример REST API на Spring

Исходный код

```
import org.springframework.stereotype.Service;

@Service
public class BookService {

   private final List<Book> books = new CopyOnWriteArrayList<>();
   private final AtomicInteger idGenerator = new AtomicInteger();

   public List<Book> getAllBooks() {
      return books;
   }

   public Optional<Book> getBook(int id) {
      for (Book book : books) {
         if (book.getId() == id) {
            return Optional.of(book);
         }
      }
      return Optional.empty();
   }
```

Пример REST API на Spring

```
public Book addBook(BookDetails details) {
   if (details.getAuthor() == null || details.getTitle() == null)
      throw new IllegalArgumentException();
   int id = idGenerator.addAndGet(1);
   Book book = new Book(id, details.getAuthor(), details.getTitle());
   books.add(book);
   return book;
}

public Optional<Book> updateBook(int id, BookDetails details) {
   if (details.getAuthor() == null || details.getTitle() == null)
      throw new IllegalArgumentException();
   for (int i = 0; i < books.size(); i++) {
      Book book = books.get(i);
      if (book.getId() == id) {
         Book newBook = new Book(book.getId(), details.getAuthor(), details.getTitle());
         books.set(i, newBook);
         return Optional.of(newBook);
      }
   }
   return Optional.empty();
}

public boolean deleteBook(int id) {
   return books.removeIf(book -> book.getId() == id);
}
```

Пример REST API на Spring

Слой бизнес-логики не должен быть завязан на HTTP (так как HTTP – только одно из возможных представлений), поэтому:

- вместо выдачи ошибки NOT_FOUND возвращаем Optional.empty()
- вместо выдачи ошибки BAD_REQUEST генерируем исключение IllegalArgumentException

Пример REST API на Spring

```
@RestController
@RequestMapping(path = "/api/books", produces = "application/json")
public class BookRest {

   private final BookService service;

   public BookRest(BookService service) {
      this.service = service;
   }

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

   @GetMapping("/{id}")
   public Book getBook(@PathVariable("id") int id) {
      Optional<Book> found = service.getBook(id);
      return found.orElseThrow(() -> new ResponseStatusException(HttpStatus.NOT_FOUND));
   }
```

Dependency Injection (DI)

Пример REST API на Spring

Dependency Injection (внедрение зависимостей) – паттерн композиции классов:

вместо

```
    this.service = new BookService();
```

мы передаем BookService извне в конструкторе:

```
public BookRest(BookService service) {
    this.service = service;
}
```

Пример REST API на Spring

Это позволяет, например, при тестировании передавать тестовый экземпляр BookService в контроллер.

Так как экземпляр BookRest создает Spring, а не мы, он должен знать, как это делать.

Для этого используется механизм Dependency Injection Container – основа Spring Framework.

Пример REST API на Spring

В простейшем случае Spring DI работает так:

- у Spring Bean (т.е. класса с аннотацией @Component, @Controller/@RestController, @Service, @Repository или @Configuration) есть единственный конструктор
- все параметры этого конструктора тоже являются Spring Beans
- тогда Spring вызывает этот конструктор с нужными Spring beans в качестве параметров

Пример REST API на Spring

Т.е. вручную написанный код выглядел бы примерно так:

```
public static void main(String[] args) {
    BookService service = new BookService();
    BookRest rest = new BookRest(service);
    ...
}
```

Spring заменяет этот простой код на черную магию DI Container.

Пример REST API на Spring

DI container:

- сканирует классы и находит среди них Spring beans
- определяет зависимости между ними
- создает граф объектов со связями между ними

Обратите внимание, что если мы уберем аннотацию @Service у BookService, приложение перестанет работать с ошибкой при запуске.

Пример REST API на Spring

Простейшую реализацию DI container можно посмотреть здесь:

https://github.com/osobolev/restdemo/blob/master/src/main/java/ru/mirea/my_di/DIContainer.java

Она работает только если:

- у каждого класса только один конструктор
- типы параметров, которые передаются в конструктор, совпадают с типом параметров (т.е. нельзя объявить параметр типа Animal, а ожидать реальный параметр типа "Cat extends Animal")

Пример REST API на Spring

Первую проблему – наличие более одного конструктора – Spring решает несколькими способами, главный из которых – отметить вызываемый Spring конструктор аннотацией @Autowired.

Пример REST API на Spring

Рассмотрим вторую проблему:

```
@Service
public class SomeService {
    private final Animal animal;
    // Конструктор
}
public interface Animal { ... }
@Component public class Cat implements Animal { ... }
@Component public class Dog implements Animal { ... }
```

Пример REST API на Spring

Решения:

- использовать аннотацию @Primary для Cat или для Dog
- использовать именованные бины:
  @Component("cat") **public class** Cat **implements** Animal { ... }
  ...
  // Spring догадывается, что нужно использовать Cat:
  private final Animal cat;

Пример REST API на Spring

По умолчанию обычные исключения Java в RestController транслируются в ошибку 500. Если мы хотим это изменить, можно использовать аннотацию @ExceptionHandler в controller:

```
@ExceptionHandler(IllegalArgumentException.class)
public ResponseEntity<String> handleException(IllegalArgumentException ex) {
   return ResponseEntity.badRequest().body(ex.getMessage());
}
```

В общем случае ResponseEntity может содержать любой объект, который будет сериализован в JSON и передан в теле ответа.

Пример REST API на Spring

Подход с аннотацией @Service имеет недостаток: код слоя бизнес-логики становится завязан на Spring – аннотация @Service должна быть доступна во время компиляции.

Таким образом, такой слой бизнес-логики трудно использовать в проектах, которые не используют Spring.

Пример REST API на Spring

В Spring существует возможность задания конфигурации бинов отдельно от кода самих бинов с помощью классов конфигурации с аннотацией @Configuration.

Таким образом мы можем писать бизнес-логику как обычный Java-класс, не полагаясь на Spring, а в основном приложении мы связываем все компоненты вместе через конфигурацию.

Пример REST API на Spring

Исходный код

```
package ru.mirea.books3;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import ru.mirea.books3.nonspring.BookService;

@Configuration
public class BookConfig {

    @Bean
    public BookService bookService() {
        return new BookService();
    }
}
```

Пример REST API на Spring

Бины создаются не только для классов, отмеченных аннотациями (@Component, @Controller, @Service, @Repository), но и при вызове методов конфигурации, отмеченных аннотацией @Bean.

Эти методы могут содержать произвольную логику, написанную на обычном языке Java. Это немного уменьшает магичность работы DI.

Пример REST API на Spring

Конфигурация позволяет сослаться из одного компонента на другой:

```
@Configuration
public class SomeConfig {

    @Bean public SomeRepo someRepo() {
        return new SomeRepo();
    }

    @Bean public SomeService someService() {
        return new SomeService(someRepo());
    }
}
```

Пример REST API на Spring

Может создаться впечатление, что при инициализации программы будет создано два объекта SomeRepo:

1. Когда Spring создает объект SomeRepo, вызывая метод someRepo()
2. Когда Spring создает объект SomeService, вызывая метод someService: при этом вызывается метод someRepo(), создающий еще один объект SomeRepo.

Пример REST API на Spring

На самом деле Spring генерирует класс-наследник исходного класса, который кэширует вызовы методов:

```
public class CachingConfig extends SomeConfig {
    private final Map<String, Object> cachedBeans = new HashMap<>();
    public SomeRepo someRepo() {
        return cachedBeans.computeIfAbsent(
            "someRepo", k -> super.someRepo()
        );
    }
    public SomeService someService() {
        return cachedBeans.computeIfAbsent(
            "someService", k -> super.someService()
        );
    }
}
```

Пример REST API на Spring

Эти классы генерируются при запуске программы динамически, после того как сканирование классов нашло аннотации @Configuration.

Динамическая генерация кода возможна в стандартной библиотеке Java через класс java.lang.reflect.Proxy, но он умеет генерировать только код, реализующий интерфейсы. В нашем случае конфигурация – это класс, а не интерфейс, поэтому Proxy не применим.

Пример REST API на Spring

Вместо этого используется библиотека динамической генерации байт-кода CGLIB. Она тоже использует ClassLoader для определения сгенерированных классов:

```
byte[] byteCode = ...; // генерируем байт-код

class MyClassLoader extends ClassLoader {

    ...

    defineClass("GeneratedName", byteCode, 0, byteCode.length);

    ...

}
```

Пример REST API на Spring

Пример использования CGLIB для генерации класса, кэширующего результаты вызова методов с аннотацией @Bean:

https://github.com/osobolev/restdemo/blob/master/src/main/java/ru/mirea/my_di/MethodCacher.java

Пример REST API на Spring

Рекомендации:

- Бизнес-логику лучше писать без привязки к Spring DI
- Приложение при этом конфигурировать с помощью методов с аннотацией @Bean в классе с аннотацией @Configuration
- Классы @RestController можно не создавать в @Configuration, так как они все равно завязаны на Spring

Пример REST API на Spring

Более развернутые примеры:

- https://spring.io/guides
- https://github.com/spring-petclinic/spring-petclinic-rest

Пример REST API на Spring

Сводка механизмов, используемых Spring:

- Classpath scanning – поиск объектов с нужными аннотациями
- Построение графа объектов исходя из зависимостей между ними
- Использование генерации кода для добавления своей логики к пользовательским методам