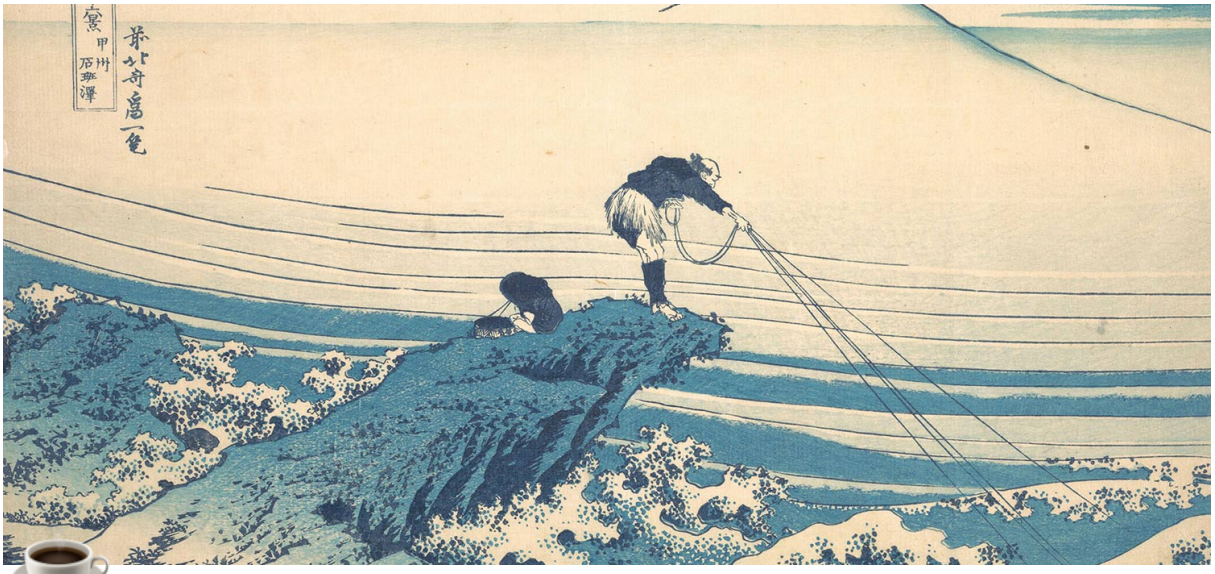

JAVA WEB THEORY

▼ LAB 1

▼ code

▼ model

```
package com.byak.currencyconverter.model;

import lombok.Data;

import java.util.Map;

@Data
public class CurrencyConverterResponse {
    private String status;
    private String updated_date;
    private String base_currency_code;
    private double amount;
    private String base_currency_name;
    private Map<String, CurrencyRate> rates;
}
```

```
package com.byak.currencyconverter.model;

import lombok.Data;
```

```
@Data
public class CurrencyRate {
    private String currency_name;
    private String rate;
    private String rate_for_amount;
}
```

В Java, `Map` является **интерфейсом**, представляющим коллекцию пар ключ-значение. Он позволяет хранить данные в виде пар ключ-значение, где каждый ключ уникален.

`Map` предоставляет методы для работы с этими парами, такие как добавление, удаление, поиск и т.д.

В Java существует несколько реализаций интерфейса `Map`

1. HashMap

- Не гарантирует порядок элементов.
- Можно `null` в качестве ключа и значения.
- Предоставляет быстрый доступ по ключу.

Использование: подходит, когда требуется **быстрый доступ** к элементам по ключу, и **порядок не важен**.

2. TreeMap

- Сортирует ключи по их естественному порядку или по заданному компаратору.
- Можно `null` в качестве **ключа, но не значения.**
- Предоставляет доступ к элементам в отсортированном порядке ключей.

Использование: подходит, когда требуется работать с элементами **в отсортированном порядке** ключей.

3. LinkedHashMap

- Сохраняет порядок вставки элементов.
- Можно `null` в качестве ключа и значения.
- Предоставляет доступ к элементам в порядке их вставки.

Использование: подходит, когда требуется **сохранить порядок вставки** элементов.

4. ConcurrentHashMap

- Не гарантирует порядок элементов.
- Можно `null` в качестве ключа и значения.
- Предназначен для использования в многопоточных средах и обеспечивает потокобезопасность.
- **Использование**: подходит там, где требуется потокобезопасное хранение пар ключ-значение, особенно в многопоточных приложениях.

5. Hashtable

- Не гарантирует порядок элементов.
- Можно `null` в качестве ключа и значения.
- Потокобезопасна, но **медленнее, чем** `ConcurrentHashMap`, из-за синхронизации.
- **Использование**: подходит, когда требуется потокобезопасное хранение пар ключ-значение, но **производительность не является критическим фактором**.

Используя `RestTemplate` для выполнения HTTP запросов и получения ответа в виде объекта Java, `RestTemplate` автоматически использует **Jackson** (или другую библиотеку для работы с JSON, если она настроена в проекте) для преобразования JSON-ответа в объект Java.

Это означает, что структура JSON-ответа должна соответствовать структуре класса Java, чтобы данные могли быть корректно сохранены в объекте.

Если ответ содержит данные для `rates`, они будут автоматически сохранены в соответствующем поле `Map<String,CurrencyRate>` объекта

`CurrencyConverterResponse`.

Имена полей в классах точно соответствуют ключам в JSON-ответе.

Если имена полей в JSON отличаются от имен полей в классах, можно использовать аннотацию `@JsonProperty` для указания соответствующего ключа JSON для каждого поля.

▼ service

```
package com.byak.currencyconverter.service;
```

```
import com.byak.currencyconverter.model.CurrencyConverterResp
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Service;
import org.springframework.web.client.RestTemplate;

@Service
public class CurrencyConverterService {
    @Value("${api.key}")
    private String apiKey;
    private static final String API_URL = "https://api.getge

    public CurrencyConverterResponse convertCurrency(String f
                                                     double a
        String url = API_URL + "?api_key=" + apiKey + "&from=
                + "&to=" + toCurrency + "&amount=" + amount +
        RestTemplate restTemplate = new RestTemplate();
        return restTemplate.getForObject(url, CurrencyConvert
    }
}
```

API ключи (API токены или ключи доступа), являются чувствительными данными, которые предоставляют доступ к API. Размещение API ключей в открытом доступе может привести к нескольким проблемам безопасности:

1. **Несанкционированный доступ**: Любой, кто увидит API ключ, может использовать его для доступа к API и к данным, которые он контролирует. Это может привести к несанкционированным действиям, таким как чтение, изменение или удаление данных.

2. **Утечка данных**: Если API ключ используется для доступа к чувствительным данным, размещение в открытом доступе может привести к утечке этих данных.

3. **Уязвимости безопасности**: Размещение API ключей в открытом доступе может увеличить риск уязвимостей безопасности, поскольку злоумышленники могут использовать эти ключи для атак на систему.

Варианты скрытия API ключей:

1. **Использование переменных окружения**: Сохранение API ключей в переменных окружения является распространенным способом скрытия ключей. Это позволяет легко изменять ключи без необходимости изменения кода и обеспечивает их безопасное хранение.

```
String apiKey = System.getenv("API_KEY");
System.out.println("API Key: " + apiKey);
```

Для использования установить переменную окружения API_KEY в вашей операционной системе или в среде выполнения приложения.

2. **Конфигурационные файлы**: Хранение API ключей в конфигурационных файлах, которые не включены в систему контроля версий (например, `.gitignore` для Git), также является безопасным способом.

```
import java.util.Properties;
String apiKey = prop.getProperty("api.key");
```

или

```
import org.springframework.beans.factory.annotation.Value;
@Value("${api.key}")
    private String apiKey;
```

при этом файл .properties

```
api.key=your_api_key_here
```

3. **Секретные менеджеры**: Использование специализированных инструментов для управления секретами, таких как HashiCorp Vault, AWS Secrets Manager или Azure Key Vault, позволяет безопасно хранить и управлять API ключами и другими секретами.

4. **Серверы конфигурации**: Использование серверов конфигурации, таких как Spring Cloud Config для Spring-приложений, позволяет централизованно управлять конфигурацией, включая API ключи, и обеспечивает их безопасное распределение.

Серверы конфигурации — это инструменты, которые помогают хранить и управлять настройками для программ и веб-сайтов. Они позволяют разработчикам и администраторам управлять всеми настройками в одном месте, а затем автоматически применять эти настройки в разных частях системы или приложения. Это упрощает процесс обновления настроек и гарантирует, что все части системы используют одни и те же настройки.

Примеры серверов конфигурации включают Spring Cloud Config для приложений на Java, Consul от HashiCorp, Etcd от CoreOS, AWS Systems Manager Parameter Store от Amazon и Azure App Configuration от Microsoft.

Использование серверов конфигурации помогает упростить управление настройками, повышает безопасность (поскольку настройки хранятся в одном месте и контролируются), и делает систему более гибкой и легкой в обслуживании.

5. **Сервисы хранения ключей**: Использование специализированных сервисов для хранения ключей, таких как AWS KMS или Google Cloud KMS, позволяет шифровать ключи и управлять доступом к ним.

Важно регулярно обновлять API ключи и использовать меры безопасности для защиты от несанкционированного доступа.

▼ controller

```
package com.byak.currencyconverter.controller;

import com.byak.currencyconverter.model.CurrencyConverterResp
import com.byak.currencyconverter.service.CurrencyConverterSe
import lombok.AllArgsConstructor;
import org.springframework.web.bind.annotation.*;

@RestController
@AllArgsConstructor
@RequestMapping("/convert")
public class CurrencyConverterController {
    private CurrencyConverterService currencyConverterServic

    @GetMapping("/{fromCurrency}/{toCurrency}/{amount}")
    public CurrencyConverterResponse convertCurrency(@PathVa
                                                     @PathVa
                                                     @PathVa
        return currencyConverterService.convertCurrency(from
    }
}
```

▼ theory

▼ JSON

JSON (JavaScript Object Notation) — это легкий формат обмена данными, основанный на тексте, который используется для хранения и передачи данных между клиентом и сервером в веб-приложениях.
JSON был разработан
для упрощения обмена данными между браузером и сервером, но

сейчас он широко используется **в различных областях, включая веб-разработку, мобильные приложения и API**.

Основные характеристики JSON:

- **Легкость**: JSON легко читается и пишется человеком, что делает его удобным для разработчиков.
- **Простота**: JSON поддерживает простые структуры данных, такие как объекты, массивы, строки, числа, булевы значения и `null`.
- **Интерпретируемость**: JSON легко интерпретируется как в JavaScript, так и в других языках программирования, что облегчает его использование в различных приложениях.
- **Кроссплатформенность**: Благодаря своей простоте и легкости интерпретации, JSON стал стандартом для обмена данными в веб-приложениях и API.

Пример JSON:

```
{
 "name": "John Doe",
 "age": 30,
 "isStudent": false,
 "courses": [
    "Mathematics",
    "Physics",
    "Computer Science"
 ],
 "address": {
    "street": "123 Main St",
    "city": "Anytown",
    "country": "USA"
 }
}
```

В этом примере JSON представляет собой объект с несколькими полями: `name`, `age`, `isStudent`, `courses` и `address`. Поле `courses` является массивом строк, а `address` — вложенным объектом.

Использование JSON в Java:

В Java для работы с JSON часто используются библиотеки, такие как Jackson, Gson и JSON-P (Java API for JSON Processing). Эти библиотеки предоставляют удобные API для сериализации объектов Java в JSON и десериализации JSON в объекты Java.

Пример сериализации объекта Java в JSON с использованием Jackson:

```
import com.fasterxml.jackson.databind.ObjectMapper;

public class User {
    private String name;
    private int age;
    // Конструкторы, геттеры и сеттеры
}
```

```
User user = new User("John Doe", 30);
ObjectMapper objectMapper = new ObjectMapper();
String jsonString = objectMapper.writeValueAsString(user);
System.out.println(jsonString);
```

▼ Аннотации

@Service

- **Описание**: Аннотация `@Service` используется для **обозначения классов, которые содержат бизнес-логику приложения**. Эти классы обычно используются для выполнения операций с данными, таких как сохранение, извлечение, обновление и удаление записей в базе данных.
- **Цель**: **Предоставление метаданных для Spring,** чтобы автоматически определить класс как компонент, который должен быть автоматически сканирован и управляем Spring IoC контейнером.

@Controller

- **Описание**: Аннотация `@Controller` используется для о**бозначения классов, которые обрабатывают входящие HTTP-запросы и возвращают ответы**. Эти классы обычно содержат методы, аннотированные `@RequestMapping` или другими аннотациями для обработки конкретных URL-адресов.
- **Цель**: **Предоставление метаданных для Spring**, чтобы автоматически определить класс как контроллер, который должен быть автоматически сканирован и управляем Spring IoC контейнером.

@Value

- **Описание**: Аннотация `@Value` используется для внедрения значений из свойств файла конфигурации (например, `application.properties` или `application.yml`) в поля класса. Это позволяет легко управлять конфигурационными параметрами приложения.
- **Цель**: Предоставление метаданных для Spring, чтобы автоматически внедрить значение из свойства конфигурации в поле класса.

@Component

Эта аннотация используется для **объявления класса как компонента Spring**.
Spring автоматически создает экземпляр этого класса и управляет его жизненным циклом.

@Autowired

Эта аннотация используется для **автоматического внедрения зависимостей в компонент**.
Spring ищет и внедряет зависимости, которые соответствуют типу и имени переменной.

@Repository

Эта аннотация используется для **объявления класса как репозитория, который обращается к базе данных**.
Spring автоматически обрабатывает исключения, связанные с БД, и предоставляет дополнительные функции для упрощения работы с БД.

Когда Spring **запускает приложение**, он сканирует классы, аннотированные `@Configuration`, и вызывает все методы, аннотированные `@Bean`, внутри этих классов. Эти методы создают и настраивают бины, которые затем становятся доступными для внедрения в другие компоненты приложения.

@Configuration

Эта аннотация используется для **объявления класса как источника конфигурации** Spring.
Классы с этой аннотацией могут содержать методы, аннотированные `@Bean`, которые определяют бины для контейнера Spring.

Конфигурация — это просто описание доступных бинов. Spring дает несколько вариантов, как можно описать набор бинов, которые сформируют приложение. Исторический вариант — это через набор **xml файлов**. В наши дни ему на смену пришли **Java аннотации**. Spring Boot построен на аннотациях чуть более, чем полностью и большинство современных библиотек в принципе тоже можно сконфигурировать через аннотации. В третьем своем поколении, конфигурация бинов пришла к подходу функциональной регистрации (**functional bean registration)**

@Bean

Эта аннотация используется для **объявления метода как фабрики для создания бина** Spring.
Методы, аннотированные `@Bean`, должны быть частью класса, аннотированного `@Configuration`.

Бин (bean) — это не что иное, как самый обычный объект. Разница лишь в том, что бинами принято называть те **объекты, которые управляются Spring-ом и живут внутри его DI-контейнера**. Бином является почти все в

Spring — сервисы, контроллеры, репозитории, по сути все приложение состоит из набора бинов. Их можно регистрировать, получать в качестве зависимостей, проксировать, мокать и т.п.

@Scope

Эта аннотация используется для **определения области видимости бина** Spring. Она может быть применена к методу, возвращающему бин, или к классу, объявленному как бин.

@Qualifier

Эта аннотация используется для **уточнения, какой бин следует внедрить, когда есть несколько кандидатов** для внедрения.

REST аннотации

1. @RestController

- **Описание**: Аннотация, используемая для обозначения класса как контроллера, который будет обрабатывать HTTP-запросы и возвращать данные в формате JSON или XML.
  Объединяет функциональность `@Controller` и `@ResponseBody`. Это означает, что класс, помеченный как `@RestController`, автоматически возвращает данные в формате JSON или XML, что делает его идеальным выбором для создания RESTful веб-сервисов.

Преимущества перед @Controller

- **Автоматическое преобразование в JSON/XML**: Когда вы используете `@RestController`, Spring автоматически преобразует возвращаемые объекты в JSON или XML, что упрощает разработку веб-сервисов.
- **Упрощение кода**: Использование `@RestController` уменьшает количество аннотаций, необходимых для каждого метода контроллера.
- **Чистота и ясность**: Использование `@RestController` делает код более чистым и ясным, поскольку оно явно указывает, что контроллер предназначен для возвращения данных в формате JSON или XML.

@Controller с @ResponseBody

- **Описание**: `@Controller` используется для обозначения класса как контроллера, который обрабатывает HTTP-запросы. Для того чтобы возвращаемые данные автоматически преобразовывались в JSON или

XML, необходимо добавить аннотацию `@ResponseBody` к каждому методу контроллера.

2. @RequestMapping

- **Описание**: Аннотация, используемая для указания маршрута, который будет обрабатывать метод контроллера. Может быть применена как к классу, так и к методу.
- **Пример**:

```
@RestController
@RequestMapping("/users")
public class UserController {
    // Методы контроллера
}
```

3. @GetMapping, @PostMapping, @PutMapping, @DeleteMapping

- **Описание**: Эти аннотации являются специализированными версиями `@RequestMapping`, предназначенными для обработки конкретных типов HTTP-запросов (GET, POST, PUT, DELETE). Они упрощают код и делают его более читаемым.

4. @PathVariable

- **Описание**: Аннотация, используемая для связывания переменной метода с параметром пути в URL.
- **Пример**:

```
@GetMapping("/{id}")
public User getUser(@PathVariable Long id) {
    // Получение пользователя по ID
}
```

5. @RequestParam

- **Описание**: Аннотация, используемая для связывания переменной метода с параметром запроса.
- **Пример**:

```
@GetMapping("/search")
public List<User> searchUsers(@RequestParam String name) {
    // Поиск пользователей по имени
}
```

6. @RequestBody

- **Описание**: Аннотация, используемая для связывания переменной метода контроллера с телом запроса. Обычно используется для получения данных в формате JSON или XML.
- **Пример**:

```
@PostMapping("/users")
public User createUser(@RequestBody User user) {
```

```
    // Создание нового пользователя
}
```

▼ API

API (**Application Programming Interface**, или интерфейс программирования приложений) — *это набор правил и протоколов, которые определяют, как программы могут взаимодействовать друг с другом*.

В контексте веб-разработки и программирования, API часто используется для описания **способа взаимодействия между клиентскими приложениями** (например, веб-браузерами или мобильными приложениями) и **серверными приложениями или сервисами**.

Основные аспекты API:

- **Определение функциональности**: API определяет, какие операции доступны для выполнения, и как эти операции должны быть вызваны.
- **Формат данных**: API определяет формат данных для передачи информации между клиентом и сервером. Обычно это JSON или XML.
- **Протокол**: API определяет протокол, который должен использоваться для передачи данных, например, HTTP или HTTPS.
- **Аутентификация и авторизация**: API может определять, как клиенты должны аутентифицироваться и авторизоваться для доступа к определенным ресурсам или операциям.

Примеры использования API:

- **Веб-API**: Веб-API позволяют в**еб-приложениям взаимодействовать с внешними сервисами или базами данных**. Например, API для погодной службы позволяет веб-приложению получать информацию о погоде.
- **REST API**: REST (Representational State Transfer) API — это стиль архитектуры, который использует **стандартные HTTP-методы (GET, POST, PUT, DELETE и т.д.)** для взаимодействия с ресурсами.
- **GraphQL API**: GraphQL API предоставляют более гибкий и эффективный способ взаимодействия с данными, позволяя клиентам запрашивать только те данные, которые им нужны.

Преимущества использования API:

- **Интеграция**: API позволяют интегрировать различные системы и сервисы, обеспечивая их взаимодействие и обмен данными.
- **Гибкость**: API предоставляют гибкость в разработке приложений, позволяя разработчикам использовать существующие сервисы и

функциональность.

- **Масштабируемость**: API облегчают масштабирование приложений, позволяя добавлять новые функции и сервисы без необходимости изменения существующего кода.

Реализация API включает в себя:

- *Определение функций и методов, которые будут доступны для вызова.*
- *Определение формата данных для запросов и ответов (часто JSON).*
- *Обеспечение безопасности и аутентификации для защиты доступа к API.*
- *Обработку ошибок и предоставление информации о статусе выполнения запросов.*

REST (R**epresentational State Transfer**) API — это стиль архитектуры, который использует стандартные HTTP-методы (GET, POST, PUT, DELETE и т.д.) для взаимодействия с ресурсами. REST API позволяет клиентам (например, веб-браузерам или мобильным приложениям) взаимодействовать с сервером, выполняя *операции над ресурсами, представленными в виде URL*.

Основные Принципы REST API:

- **Ресурсы**: В REST API **ресурсы представляются в виде URL**. Каждый ресурс имеет уникальный идентификатор, который может быть использован для доступа к нему.
- **Методы HTTP**: REST API **использует стандартные методы HTTP** для выполнения операций над ресурсами (GET, POST...).
- **Статусы HTTP**: REST API **использует коды состояния HTT**P для информирования клиента о результате выполнения запроса. Например, 200 OK для успешного выполнения, 404 Not Found для отсутствия ресурса, и 500 Internal Server Error для ошибки на сервере.
- **Представление данных**: Данные, передаваемые между клиентом и сервером, обычно представлены в формате JSON или XML. JSON является более популярным выбором из-за своей легкости и совместимости с JavaScript.

Преимущества REST API:

- **Простота**: REST API легко понять и использовать благодаря использованию стандартных HTTP-методов и статусов.
- **Масштабируемость**: REST API легко масштабируются, поскольку они не зависят от состояния и могут быть легко распределены по

нескольким серверам.

- **Кроссплатформенность**: REST API могут быть использованы в любой среде, поддерживающей HTTP, что делает их универсальными для разработки веб-приложений и мобильных приложений.

Примеры использования REST API:

- **Веб-сервисы**: REST API часто используются для создания веб-сервисов, которые предоставляют данные или функциональность другим приложениям.
- **Мобильные приложения**: Мобильные приложения могут использовать REST API для взаимодействия с сервером, получения данных или выполнения операций.
- **Интернет вещей (IoT)**: Устройства IoT могут использовать REST API для обмена данными с сервером или другими устройствами.

Безопасность REST API:

- **Аутентификация и авторизация**: REST API часто используют механизмы аутентификации и авторизации, такие как OAuth 2.0 или JWT (JSON Web Tokens), для защиты доступа к ресурсам.
- **HTTPS**: Использование HTTPS для защиты данных в передаче между клиентом и сервером.
- **Ограничение доступа**: Ограничение доступа к определенным ресурсам или операциям на основе ролей пользователей или других критериев.

▼ HTTP методы

GET

- Получение данных от сервера

Запрос на получение списка пользователей.

POST

- Отправка данных на сервер для создания нового ресурса.

Создание нового пользователя.

PUT

- Обновление существующего ресурса на сервере.

Обновление информации о пользователе.

DELETE

- Удаление ресурса на сервере.

Удаление пользователя.

OPTIONS

- Получение информации о возможностях ресурса, включая поддерживаемые методы.

Проверка, поддерживает ли сервер метод PATCH для обновления пользователя.

HEAD

- Похож на GET запрос, но возвращает только заголовки ответа, без тела ответа. Используется для получения метаданных ресурса.

Проверка, существует ли ресурс.

PATCH

- Частичное обновление ресурса на сервере.

Изменение пароля пользователя.

TRACE

- Получение диагностической информации о пути, по которому запрос проходит от клиента к серверу и обратно.

Отладка сетевых проблем.

CONNECT

- Установление сетевого соединения с сервером, обычно для использования с прокси-серверами.

Соединение с сервером через прокси-сервер.

Разница между PUT и PATCH:

- **Полное обновление**: PUT требует отправки **полного набора данных** для ресурса, в то время как PATCH позволяет отправлять **только измененные поля**.
- **Семантика**: PUT **предполагает**, что клиент имеет полное представление о ресурсе и хочет его **полностью обновить**, в то время как PATCH **предполагает**, что клиент хочет **применить изменения** к существующему ресурсу.
- **Производительность**: PATCH **может быть более эффективным** для обновления больших ресурсов, где изменяется только небольшая часть данных, поскольку он т**ребует отправки меньшего объема данных**.

Разница между методами:

- **Изменение состояния**: GET, OPTIONS и HEAD запросы не должны изменять состояние сервера, в то время как POST, PUT, DELETE, PATCH, TRACE и CONNECT могут.
- **Использование**: GET используется для получения данных, POST для создания новых ресурсов, PUT для обновления существующих ресурсов, DELETE для удаления ресурсов, PATCH для частичного обновления ресурсов, OPTIONS для получения информации о возможностях ресурса, HEAD для получения заголовков ответа, TRACE для диагностики и CONNECT для установления сетевого соединения.
- **Безопасность**: Некоторые методы, такие как DELETE, могут быть более рискованными, поскольку они могут привести к необратимым изменениям данных.

HTTP методы используются для взаимодействия между клиентом и сервером, выполняя различные операции над ресурсами.

▼ Слои

В архитектуре проекта на Java с использованием Spring Framework, типовые слои обычно включают в себя следующие компоненты: Controller, Model, Service, DAO (Data Access Object), DTO (Data Transfer Object), и Entity.

Controller

Обработка входящих HTTP-запросов и возвращение ответов. Контроллеры ***принимают запросы от клиентов, обрабатывают их,*** используя сервисы, и возвращают данные в формате, который может быть легко обработан клиентом (например, JSON).

Разница: *Контроллеры **являются точкой входа** в приложение и обычно **содержат минимальную логику**, связанную с обработкой запросов. Они делегируют большую часть бизнес-логики сервисам.*

Model

Модель представляет собой **объекты** бизнес-логики приложения. Они содержат **данные и методы для работы с этими данными**. Модель может включать в себя классы, которые представляют сущности бизнес-процессов, такие как пользователи, продукты, заказы и т.д.

Разница: *Модель отражает бизнес-логику и **данные** приложения, в то время как контроллеры и сервисы **обрабатывают входящие запросы и возвращают данные**.*

Service

Содержит бизнес-логику приложения.
Он
обрабатывает данные, полученные от контроллеров, выполняет необходимые операции с моделями и возвращает результаты обратно в контроллеры.
Сервисы могут включать в себя валидацию данных - проверку входных данных на соответствие определенным критериям или правилам, выполнение бизнес-правил (бизнес-правила могут быть сложными и включать в себя логику, которая определяет, какие действия разрешены, какие данные должны быть сохранены или обновлены, и как должны взаимодействовать разные части системы) и взаимодействие с другими сервисами.

Разница: *Сервисы сосредоточены на выполнении **бизнес-логики и обработке данных**, в то время как контроллеры и модели отвечают за **обработку запросов и представление данных соответственно**.*

DAO (Data Access Object)

DAO слой отвечает за **взаимодействие с базой данных**. Он содержит методы для создания, чтения, обновления и удаления данных в базе данных.
DAO обычно использует технологии ORM (Object-Relational Mapping), такие как Hibernate (фреймворк, который автоматизирует процесс преобразования объектов Java в таблицы базы данных и наоборот), для упрощения работы с базой данных.

Разница: *DAO абстрагирует детали работы с базой данных, предоставляя высокоуровневый интерфейс для взаимодействия с данными. Это позволяет изменять механизмы доступа к данным без изменения бизнес-логики приложения.*

DTO (Data Transfer Object)

DTO используется для **передачи данных между слоями приложения**. Они обычно содержат *только данные и не содержат бизнес-логики*. DTO могут использоваться для передачи данных **между клиентом и сервером**, а также между слоями приложения.

Разница: *DTO служат для **передачи данных и не содержат бизнес-логики**, в отличие от моделей, которые могут содержать методы для работы с данными.*

Entity

Entity представляет собой **классы, которые сопоставляются с таблицами базы данных**. Они используются для представления данных в базе данных и могут содержать аннотации ORM, такие как `@Entity`, `@Table`, `@Id`, для настройки сопоставления с базой данных.

Разница: *Entity* ***отражает структуру данных в БД*** *и используется для* ***взаимодействия с БД через*** *DAO* ***слой****.*

DTO используются для ***передачи данных между слоями приложения****,*

Entity обычно содержит ***больше бизнес-логики и методов для работы с данными****.*

Эти слои вместе обеспечивают четкую разделение ответственности в архитектуре приложения, что упрощает его разработку, тестирование и поддержку.

▼ Spring и составляющие

Spring Framework — это мощный и гибкий фреймворк для разработки Java-приложений, который предоставляет широкий спектр функциональности для упрощения разработки, тестирования и развертывания приложений. Spring состоит из нескольких модулей, каждый из которых предназначен для решения конкретных задач разработки. Вот основные составляющие Spring и их описание:

Core Container

- **Основной модуль Spring**, который предоставляет основные функции, такие как внедрение зависимостей, конфигурацию и жизненный цикл бинов.
- **@Component**, **@Autowired**, **@Configuration**, **@Bean** — аннотации, используемые для определения и настройки бинов.

Spring MVC

- **Модуль для создания веб-приложений**, который предоставляет мощный и гибкий механизм для обработки HTTP-запросов.
- Использует аннотации, такие как **@Controller**, **@RequestMapping**, **@GetMapping**, **@PostMapping** для определения контроллеров и маршрутов.

Spring Data

- **Модуль для упрощения доступа к данным**, предоставляющий абстракции для работы с различными источниками данных.
- Поддерживает различные базы данных и предоставляет универсальный API для выполнения CRUD-операций.

Spring Security

- **Модуль для обеспечения безопасности приложений**, предоставляющий функции аутентификации, авторизации и защиты от атак.
- Использует аннотации, такие как **@Secured**, **@PreAuthorize** для управления доступом к методам и ресурсам.

Spring Boot

- **Модуль для упрощения создания и развертывания Spring-приложений**, предоставляющий автоматическую конфигурацию и упрощенный процесс развертывания.
- **@SpringBootApplication** — аннотация, которая объединяет все необходимые аннотации для быстрого старта Spring-приложения.

Различия между составляющими

- **Core Container** фокусируется на основных аспектах Spring, таких как **внедрение зависимостей и конфигурация.**
- **Spring MVC** специализируется на разработке веб-приложений, предоставляя мощные инструменты для обработки HTTP-запросов.
- **Spring Data** упрощает доступ к данным, предоставляя абстракции для работы с различными источниками данных.
- **Spring Security** фокусируется на обеспечении безопасности приложений, предоставляя функции для аутентификации и авторизации.
- **Spring Boot** упрощает процесс создания и развертывания Spring-приложений, предоставляя **автоматическую конфигурацию и упрощенный процесс развертывания**.

Все эти модули вместе создают мощный и гибкий фреймворк для разработки Java-приложений, позволяя разработчикам сосредоточиться на бизнес-логике, а не на технических деталях.

Spring Web и Spring MVC часто используются как синонимы, но на самом деле они относятся к разным аспектам Spring Framework. Давайте разберемся в различиях и связях между ними.

Spring Web

Spring Web — это проект в рамках Spring Framework, который предоставляет поддержку для создания веб-приложений. Он включает в себя несколько модулей, каждый из которых предназначен для решения конкретных задач разработки веб-приложений. Основные модули Spring Web включают:

- **Spring MVC**: Модуль для создания веб-приложений с использованием шаблона проектирования Model-View-Controller (MVC).
- **Spring WebFlux**: Модуль для создания реактивных веб-приложений, используя реактивное программирование.
- **Spring WebSocket**: Модуль для поддержки WebSocket-соединений, позволяющих реализовать двустороннюю связь между клиентом и сервером.

Spring MVC

Spring MVC — это конкретный модуль в рамках Spring Web, который предоставляет реализацию шаблона проектирования Model-View-Controller (MVC) для создания веб-приложений. MVC — это шаблон проектирования, который разделяет приложение на три компонента:

- **Model**: Представляет данные и бизнес-логику приложения.
- **View**: Отвечает за отображение данных модели пользователю.
- **Controller**: Обрабатывает входящие запросы, взаимодействует с моделью и выбирает представление для отображения.

Spring MVC предоставляет механизмы для определения маршрутов, обработки запросов, валидации данных, а также интеграции с другими модулями Spring, такими как Spring Security и Spring Data.

Различия и связь

- **Spring Web** — это более широкий проект, включающий в себя несколько модулей для разработки веб-приложений, включая Spring MVC.
- **Spring MVC** — это конкретный модуль в рамках Spring Web, который предоставляет реализацию шаблона MVC для создания веб-приложений.

Важно отметить, что Spring MVC является частью Spring Web, и термины часто используются взаимозаменяемо. Однако, когда говорят о Spring Web, они обычно имеют в виду более широкий набор инструментов и функций для разработки веб-приложений, включая Spring MVC, Spring WebFlux и другие модули.

▼ Maven и Gradle

▼ Ошибки

HTTP-ошибки классифицируются по кодам состояния, которые состоят из трех цифр. Эти коды состояния разделены на пять классов, каждый из которых указывает на тип ответа, который сервер отправляет клиенту. Вот основные классы HTTP-ошибок и их наиболее распространенные коды:

1xx (Предварительные ответы)

- **100 Continue**: Сервер получил запрос и ожидает продолжения.
- **101 Switching Protocols**: Сервер переключается на другой протокол, как указано в заголовке запроса.

2xx (Успешные ответы)

- **200 OK**: Запрос успешно обработан.
- **201 Created**: Запрос успешно обработан, и в результате был создан новый ресурс.
- **202 Accepted**: Запрос принят, но обработка еще не завершена.
- **204 No Content**: Запрос успешно обработан, но не возвращает содержимое.

3xx (Перенаправления)

- **300 Multiple Choices**: Запрос имеет несколько возможных ответов.
- **301 Moved Permanently**: Запрашиваемый ресурс был перемещен на другой URL.
- **302 Found**: Запрашиваемый ресурс временно находится на другом URL.
- **304 Not Modified**: Ресурс не изменился с момента последнего запроса.

4xx (Ошибки клиента)

- **400 Bad Request**: Запрос содержит синтаксические ошибки.
- **401 Unauthorized**: Требуется аутентификация для доступа к ресурсу.
- **403 Forbidden**: Доступ к ресурсу запрещен.
- **404 Not Found**: Запрашиваемый ресурс не найден.
- **405 Method Not Allowed**: Используемый метод HTTP не поддерживается для запрашиваемого ресурса.

5xx (Ошибки сервера)

- **500 Internal Server Error**: На сервере произошла непредвиденная ошибка.
- **501 Not Implemented**: Запрашиваемый метод HTTP не поддерживается сервером.

- **502 Bad Gateway**: Сервер, действуя в качестве шлюза или прокси, получил недействительный ответ от другого сервера.
- **503 Service Unavailable**: Сервер временно не может обработать запрос.
- **504 Gateway Timeout**: Сервер, действуя в качестве шлюза или прокси, не получил ответ от другого сервера вовремя.

Эти коды состояния помогают клиентам и серверам понимать, что произошло во время обработки HTTP-запроса, и как на него реагировать.

▼ Варианты передачи запроса

1. Java URLConnection

`URLConnection` — это класс, предоставляемый стандартной библиотекой Java, который позволяет выполнять HTTP-запросы. Он может использоваться для отправки GET и POST запросов.

2. Apache HttpClient

Apache HttpClient — это мощная библиотека для выполнения HTTP-запросов, которая предоставляет гибкие возможности для настройки запросов и обработки ответов.

3. OkHttp

OkHttp — это высокопроизводительная библиотека для выполнения HTTP-запросов, разработанная командой Square. Она предлагает простой и удобный API для отправки запросов.

4. Spring WebClient

Spring WebClient — это неблокирующий HTTP-клиент, который предоставляет удобный API для выполнения HTTP-запросов в асинхронном режиме. Он является частью Spring WebFlux, но также может использоваться в традиционных Spring-приложениях.

▼ RestTemplate

`RestTemplate` является классом в Spring Framework, который предоставляет удобный способ для выполнения HTTP запросов. Он был разработан для упрощения взаимодействия с веб-сервисами, позволяя разработчикам легко отправлять HTTP запросы и обрабатывать ответы. `RestTemplate` поддерживает различные типы HTTP запросов, такие как GET, POST, PUT, DELETE и другие, и может автоматически преобразовывать ответы в объекты Java.

Основные особенности `RestTemplate`:

1. **Удобство использования**:

`RestTemplate` предоставляет простой и понятный API для выполнения HTTP запросов.
2.
Поддержка различных типов запросов:
Можно
легко выполнять GET, POST, PUT, DELETE и другие типы запросов.
3.
Автоматическое преобразование:

`RestTemplate` может автоматически преобразовывать ответы в объекты Java, что упрощает работу с данными.
4.
Обработка ошибок:

`RestTemplate` предоставляет механизмы для обработки ошибок, такие как исключения и статусы ответов.
5.
Настраиваемость:

`RestTemplate` может быть настроен для изменения поведения, например, для добавления заголовков, изменения таймаутов и т.д.

- `getForObject(String url, Class<T> responseType)`: выполняет GET запрос и возвращает тело ответа, преобразованное в указанный тип.
- `getForEntity` возвращает `ResponseEntity`, который содержит не только тело ответа, но и статус ответа, заголовки и другую информацию.
- автоматически преобразует тело ответа в объекты Java, что упрощает работу с данными. Это достигается за счет использования библиотеки Jackson для JSON и XML, что позволяет легко работать с данными в этих форматах.

С версии Spring 5, `RestTemplate` находится в режиме поддержки, и для новых проектов рекомендуется использовать `WebClient`, который предлагает более современные и гибкие возможности.

▼ WebClient

`WebClient` в Spring WebFlux представляет собой мощный и гибкий HTTP-клиент, который предлагает ряд преимуществ по сравнению с традиционными синхронными HTTP-клиентами, такими как `RestTemplate`. Вот некоторые из ключевых преимуществ `WebClient`:

1. Асинхронность и Неблокирующий Исполнение

`WebClient` поддерживает асинхронное выполнение запросов, что позволяет выполнять другие задачи в то время как запрос обрабатывается.

2. Реактивное Программирование

`WebClient` полностью интегрирован с реактивным программированием, что делает его идеальным выбором для проектов, использующих Spring WebFlux. Реактивное программирование позволяет эффективно обрабатывать потоки данных и обеспечивает лучшую производительность и масштабируемость по сравнению с традиционными блокирующими операциями.

3. Гибкая Конфигурация

`WebClient` предлагает гибкую конфигурацию, позволяя легко настраивать параметры запроса, такие как базовый URL, заголовки, параметры запроса и кодировку. Это упрощает создание и использование клиента для выполнения различных типов HTTP-запросов.

4. Поддержка JSON и других Форматов Данных

`WebClient` поддерживает различные форматы данных, включая JSON, XML и другие, что делает его универсальным инструментом для работы с веб-сервисами. Он также предоставляет удобные методы для преобразования ответов в объекты Java.

5. Интеграция с Spring Ecosystem

`WebClient` тесно интегрирован с остальной частью Spring Ecosystem, включая Spring Security для аутентификации и авторизации, Spring Data для работы с базами данных и другими компонентами. Это обеспечивает согласованный и мощный набор инструментов для разработки веб-приложений.

6. Поддержка Тестирования

Spring WebFlux предоставляет поддержку тестирования для `WebClient`, что позволяет легко создавать и выполнять тесты для HTTP-клиентов. Это упрощает проверку корректности работы клиента и обеспечивает высокое качество кода.

В целом, `WebClient` предлагает ряд преимуществ, которые делают его предпочтительным выбором для разработки современных веб-приложений, требующих высокой производительности, масштабируемости и поддержки реактивного программирования.

▼ spring mvc test, но это тестирование

▼ Feign от Netflix

5. Java 11+ HttpClient

Начиная с Java 11, в стандартную библиотеку Java был добавлен новый HTTP-клиент, который предлагает асинхронные и синхронные методы для выполнения HTTP-запросов.

▼ Dependency injection

В Spring Framework существует несколько видов внедрения зависимостей (Dependency Injection, DI), каждый из которых имеет свои особенности, преимущества и недостатки. Давайте рассмотрим основные виды DI в Spring и их плюсы и минусы.

1. Конструкторное внедрение зависимостей (Constructor Injection)

Описание: Зависимости внедряются через конструктор класса. Это гарантирует, что объект будет иметь все необходимые зависимости сразу после его создания.

Преимущества:

- **Неизменность**: После создания объекта его состояние не может быть изменено, что улучшает безопасность и предсказуемость.
- **Ясность**: Все зависимости явно указываются в конструкторе, что делает код более понятным и легко читаемым.

Недостатки:

- **Сложность**: Может привести к большому количеству конструкторов для классов с множеством зависимостей.

Пример:

```
@Service
public class UserService {
    private final UserRepository userRepository;

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

2. Установка зависимостей через сеттеры (Setter Injection)

Описание: Зависимости устанавливаются через сеттеры класса. Это позволяет изменять зависимости после создания объекта.

Преимущества:

- **Гибкость**: Позволяет изменять зависимости после создания объекта.

Недостатки:

- **Модификация состояния**: Позволяет изменять состояние объекта после его создания, что может привести к непредсказуемому поведению.

- **Неявность**: Зависимости не явно указываются в конструкторе, что может сделать код менее понятным.

Пример:

```
@Service
public class UserService {
    private UserRepository userRepository;

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

3. Установка зависимостей через поля (Field Injection)

Описание: Зависимости устанавливаются напрямую через поля класса. Это самый простой способ внедрения зависимостей, но он не рекомендуется из-за проблем с тестированием и управлением состоянием.

Преимущества:

- **Простота**: Самый простой способ внедрения зависимостей.

Недостатки:

- **Неявность**: Зависимости не явно указываются в конструкторе или сеттерах, что может сделать код менее понятным.
- **Проблемы с тестированием**: Труднее тестировать классы с внедренными зависимостями через поля.
- **Управление состоянием**: Позволяет изменять состояние объекта после его создания, что может привести к непредсказуемому поведению.

Пример:

```
@Service
public class UserService {
    @Autowired
    private UserRepository userRepository;
}
```

Выбор подхода

Выбор подхода к внедрению зависимостей зависит от конкретных требований к приложению и предпочтений разработчика. Конструкторное внедрение зависимостей является наиболее предпочтительным, так как оно обеспечивает неизменность и ясность кода. Однако в некоторых случаях установка зависимостей через сеттеры или поля может быть удобна для упрощения кода или из-за специфических требований к приложению.

▼ Inversion of Control

Spring IoC (Inversion of Control) контейнер является сердцем Spring Framework, обеспечивая централизованное управление компонентами приложения. IoC контейнер отвечает за создание, настройку и управление объектами (или "bean" в терминологии Spring), что позволяет **разработчикам сосредоточиться на бизнес-логике, а не на управлении жизненным циклом объектов**.

Основные концепции IoC контейнера

- **Конфигурация**: Spring IoC контейнер настраивается с помощью XML-файлов, аннотаций или Java-конфигураций. Это позволяет разработчикам определять, какие компоненты должны быть созданы и как они должны быть связаны.
- **Создание объектов**: IoC контейнер отвечает за создание объектов, управляемых Spring. Это включает в себя создание экземпляров классов, настройку их свойств и внедрение зависимостей.
- **Внедрение зависимостей**: IoC контейнер автоматически внедряет зависимости в компоненты приложения. Это означает, что разработчики не должны явно создавать зависимые объекты; вместо этого они просто указывают, какие зависимости требуются, и Spring IoC контейнер заботится о их предоставлении.
- **Жизненный цикл объектов**: IoC контейнер управляет жизненным циклом объектов, включая их создание, настройку и уничтожение. Это позволяет разработчикам сосредоточиться на бизнес-логике, а не на управлении жизненным циклом объектов.

Пример использования IoC контейнера

Предположим, у вас есть класс `UserService`, который зависит от `UserRepository`:

```
@Service
public class UserService {
    private final UserRepository userRepository;

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

    // Методы для работы с пользователями
}
```

В этом примере, `UserService` зависит от `UserRepository`. Вместо того чтобы явно создавать экземпляр `UserRepository` в `UserService`, мы используем аннотацию `@Autowired` для автоматического внедрения зависимости. Spring IoC контейнер создает экземпляр `UserRepository` и внедряет его в `UserService` при его создании.

Преимущества использования IoC контейнера

- **Упрощение управления зависимостями**: IoC контейнер автоматически управляет зависимостями, что упрощает код и уменьшает количество ошибок.
- **Гибкость**: Разработчики могут легко изменять конфигурацию приложения без необходимости изменять код.
- **Тестирование**: IoC контейнер упрощает тестирование компонентов, позволяя легко заменять зависимости на моки или стабы.