**Конспект:**

**Spring:**

1. *Аннотации:*

**@Controller** – так же как и RestController – это аннотация, которой помечаются классы, внутри, которых обрабатывает HTTP запросы, которые помечаются соответсвенно(GetMapping, PostMapping и т.д), разница данных аннотаций в том что, функции внутри @Controller чаще всего возвращаются представления ну или view, чаще всего он используется с дополнительным шаблонизатором, таким как JSP или Thymeleaf

**@RestController** – аннотация, которой помечается класс, внутри, которого будут функции, которые будут принимать Rest запросы

**@Service** – данной аннотацией, помечается классы, которые содержат бизнес-логику, которая будет применяться в разных частях программы

**@Bean** - это **аннотация** используется в Spring для определения метода, который возвращает объект, управляемый контейнером Spring (Bean). @Bean используется в конфигурационных классах, помеченных аннотацией @Configuration, для создания и настройки бинов, которые потом можно инжектировать в другие части приложения.

**@RequestMapping** – данная аннотация в Spring помечается функция (или класс), чтобы настроить данные о принимаемых запросах, такие как End Point, тип HTTP-запроса и другие параметры

**@RequestBody** – аннотация, которая помечает объект внутри, которого будет помещено тело запроса, этот объект должен иметь название переменных одинаковые с теми переменными, которые передаются в теле запроса.

**@Entity –** данная аннотация помечает класс, которая используется в Spring (с JPA или Hibernate) для того, чтобы пометить класс как сущность, которая будет сопоставляться с таблицей в базе данных. Класс, помеченный аннотацией @Entity, представляет собой модель данных, где описаны поля, которые будут сохраняться в базу данных, и их настройки.

**@Id –** данной аннотацией помечается поле, которое будет является первичным ключом в базе данных

**@GenerateValue –** аннотация используется для того, чтобы пометить поле, значение которого будет автоматически генерироваться. Она задает стратегию генерации значений для поля

**@GeneratedValue**(strategy = GenerationType.IDENTITY) – задает тип генерации для унификатора(первичного ключа)

**@Column**(unique = true) – задает настройку уникальности для поля, для соответствующей колонки в базе данных

**@Column –** Аннотацией @Column в JPA помечается переменная (поле) класса, и внутри этой аннотации можно задать настройки для соответствующей колонки в базе данных. Эта аннотация позволяет указать различные параметры, такие как имя колонки, уникальность, обязательность, тип, длину и другие свойства

**@Repository** – этой аннотацией помечаются классы, которые описывают функции для взаимодействия с базой данных

**@Autowired -**Аннотация @Autowired в Spring используется для того, чтобы автоматически инжектировать зависимости в объект, который помечен этой аннотацией. Spring автоматически находит и создает необходимые зависимости, упрощая процесс связывания компонентов приложения.

**@Transactional** – эта аннотация в Spring используется для того, чтобы пометить метод (или класс) как транзакционный. Это значит, что все операции внутри метода будут выполнены в рамках одной транзакции. Если в ходе выполнения метода произойдет ошибка, все изменения, сделанные до этой ошибки, будут автоматически откатаны. Это помогает сохранить целостность данных в случае возникновения исключений или сбоев в процессе выполнения операций с базой данных.

**@Component –** данная аннотацией можно пометить класс, содержащий утилитарные методы, такие как шифрование паролей, работа с форматированием данных, логирование и другие вспомогательные операции. Аннотация @Component используется для указания на то, что этот класс является кандидатом для управления Spring, и Spring автоматически создает бин из этого класса.

**@Configuration -** аннотация в Spring используется для пометки классов, которые содержат конфигурацию компонентов Spring. Эти классы представляют собой замену XML-конфигурации и используются для создания и настройки бинов.

**@EnableWebSecurity –** аннотация @EnableWebSecurity служит сигналом для Spring, чтобы включить стандартные методы защиты, предоставляемые Spring Security. Более того, она позволяет вам настраивать эти методы защиты в конфигурационном классе.

**@Value() -** аннотация @Value позволяет присвоить внутрь переменной значение из переменной application.properties((или другого источника свойств)), но для начала нужно создать метаданные для этой переменной

**@ControllerAdvice -** это аннотация в фреймворке Spring, которая используется для глобальной обработки исключений, привязки данных и добавления общих атрибутов модели для всех контроллеров в приложении. Она позволяет централизовать логику, которая иначе потребовала бы повторения в каждом контроллере, делая код более чистым и управляемым.

1. *Spirng Core:*

Spring Core — это основополагающий модуль Spring Framework, предоставляющий базовые функциональные возможности, на которых строятся все остальные модули Spring. Он отвечает за управление компонентами приложения, их конфигурацию и взаимодействие.

**IoC** — принцип, при котором контроль над созданием и управлением объектами передается контейнеру (в данном случае Spring), а не самому приложению.

Это способствует снижению связности компонентов и облегчает их тестирование и повторное использование.

**Dependency Injection (DI) — Внедрение зависимостей:**

**DI** — механизм, позволяющий автоматически передавать зависимости (например, объекты) в компоненты приложения.

Spring поддерживает различные виды внедрения зависимостей: через конструктор, сеттеры или поля.

**Контейнер Spring (ApplicationContext):**

**ApplicationContext** — центральный интерфейс Spring для управления бинами (компонентами).

Он отвечает за создание, конфигурацию и объединение объектов, а также за управление их жизненным циклом.

**Аспектно-ориентированное программирование (AOP) в Spring**

**Aspect-Oriented Programming (AOP)** — это парадигма программирования, которая позволяет разделять сквозные (cross-cutting) заботы от основной бизнес-логики приложения. Сквозные заботы — это функциональности, которые затрагивают несколько модулей приложения, такие как логирование, управление транзакциями, безопасность и т.д. AOP помогает структурировать код таким образом, чтобы эти функциональности были вынесены в отдельные модули (аспекты), что повышает модульность и облегчает сопровождение кода.

### Основные понятия AOP

1. **Аспект (Aspect):**
   * Модуль, который инкапсулирует сквозную заботу.
   * Может содержать одну или несколько точек соединения (pointcuts) и советы (advice).
2. **Совет (Advice):**
   * Действие, выполняемое в определенное время при выполнении программы.
   * Существует несколько типов советов:
     + **Before:** Выполняется до метода.
     + **After:** Выполняется после метода (независимо от результата).
     + **After Returning:** Выполняется после успешного завершения метода.
     + **After Throwing:** Выполняется, если метод завершился исключением.
     + **Around:** Окружающий совет, который может контролировать выполнение метода и изменять его поведение.
3. **Точка соединения (Join Point):**
   * Момент выполнения программы, в который может быть применен совет.
   * В Spring AOP поддерживаются точки соединения на уровне вызова методов.
4. **Pointcut:**
   * Определяет, в какие точки соединения будет применен совет.
   * Обычно задается с помощью выражений, определяющих набор методов или классов.
5. **Weaving (Внедрение аспектов):**
   * Процесс объединения аспектов с основной логикой приложения.
   * В Spring AOP внедрение происходит во время выполнения (runtime) с помощью прокси-объектов.

Для того чтобы использовать парадигму AOP – необходимо настроить её в конфигурационном файле, а так же установить аннотацию @EnableAspectJAutoProxy – включение поддержки AOP.

**Join Point** могут быть переданы, как входные значение для функции помеченные советом, **Pointcut указываются в круглых скобочках после выбранного совета.**

**Внедрение Spring AOP происходит на этапе runtime и поддерживает точки соединения только методы**

**Пример:**

@Aspect

@Component

public class LoggingAspect {

@Before("execution(\* com.example.service.\*.\*(..))")

public void logBefore(JoinPoint joinPoint) {

System.out.println("Вызов метода: " + joinPoint.getSignature().getName()); }

@AfterReturning(pointcut = "execution(\* com.example.service.\*.\*(..))", returning = "result")

public void logAfter(JoinPoint joinPoint, Object result) {

System.out.println("Метод " + joinPoint.getSignature().getName() + " вернул " + result);

}

}

**Обработка событий (Event Handling)** в Spring позволяет компонентам приложения взаимодействовать друг с другом посредством публикации и прослушивания событий. Это способствует созданию более гибких, слабосвязанных и реактивных приложений

Обработка событий включает в себя 3 этапа:

1.Создание события: Это класс, в котором описываются логика события, например, регистрация пользователя, это класс должен наследоваться от ApplicationEvent

2. Создание служителя события, это метод, помеченный аннотацией @EventListener, он принимает на вход event, в этом методе реализуется логика, от

3. Публикация события происходит в месте, где он должен быть вызван, при публикации срабатывает функция, которая является слушателем, публикация происходит через ApplicationEventPublisher, который должен быть инжектирован спрингом.

### Встроенные события Spring

Spring предоставляет несколько встроенных событий, которые можно использовать для различных целей:

1. **ContextRefreshedEvent:** Срабатывает при инициализации или обновлении контекста приложения.
2. **ContextStartedEvent:** Срабатывает при запуске контекста приложения.
3. **ContextStoppedEvent:** Срабатывает при остановке контекста приложения.
4. **ContextClosedEvent:** Срабатывает при закрытии контекста приложения.
5. **RequestHandledEvent:** Для веб-приложений — срабатывает при обработке HTTP-запроса.
6. *Конфиги:*

Конфигурационные файлы, это классы, которые помечаются аннотацией @Configuration, в них происходит настройка создания бинов, помечанные аннотацией @Bean, в них мы так же можем создавать и настраивать внутренние бины spring, например, SecurityFilterChain,

1. *Корс:*

**CORS** (Cross-Origin Resource Sharing) — это механизм веб-безопасности, который позволяет контролировать доступ веб-страниц к ресурсам на другом домене. По умолчанию, браузеры ограничивают такие кросс-доменные запросы по соображениям безопасности. CORS предоставляет способ серверу указать браузеру, какие запросы с других доменов он разрешает.

Можно настраивать CORS локально с помощью аннотации @CrossOrigin, так и глобавльно внутри конфигурационный файл, который имплементирует интерфейс WebMvcConfigurer

@Configuration

public class GlobalCorsConfig implements WebMvcConfigurer {

@Override

public void addCorsMappings(CorsRegistry registry) { registry.addMapping("/\*\*").allowedOrigins("https://global.example.com") .allowedMethods("GET", "POST")

.allowedHeaders("\*") .allowCredentials(true);

}

}

Spring поддерживает **совмещение** локальной и глобальной настройкой CORS

1. *CSRF:*

CSRF (межсайтовая подделка запроса) — это тип атаки на веб-приложения, при котором злоумышленник заставляет аутентифицированного пользователя выполнить нежелательные действия на доверенном сайте. Это возможно, когда браузер пользователя автоматически добавляет аутентификационные данные (например, куки) к запросам.

Spring Security по умолчанию включает защиту от CSRF для веб-приложений, использующих сессионную аутентификацию. Эта защита основана на использовании специальных токенов, которые проверяются при каждом небезопасном запросе (POST, PUT, DELETE и т.д.).

**Как работает CSRF-защита в Spring**

1. **Генерация CSRF-токена**: При загрузке страницы сервер генерирует уникальный CSRF-токен и сохраняет его в сессии.
2. **Встраивание токена в запрос**: Клиентское приложение (браузер) должно отправлять этот токен вместе с каждым небезопасным запросом.
3. **Проверка токена на сервере**: При получении запроса сервер сравнивает токен из запроса с токеном в сессии. Если они совпадают, запрос считается легитимным.

Т.е механизм стандартного spring анологичен работы JWT токен

1. *JDBC и JPA + настройка*

**JDBC** – это низкоуровневый подход взаимодействия с базой данных, он основан на том, что внутри Java записываются SQL запросы.

Для того чтобы использовать JDBC, мы должны установить необходимые зависимости внутри Gradle

implementation 'org.springframework.boot:spring-boot-starter-jdbc'

Так же нужно установить нужный драйвер в зависимости от того,какой субд используется, пример:

implementation 'org.postgresql:postgresql:42.5.0

Далее в application.properties нужно настроить взаимодействие с субд:

# Настройки источника данных spring.datasource.url=jdbc:postgresql://localhost:5432/mydb spring.datasource.username=логин

spring.datasource.password=пароль

# установка driver

spring.datasource.driver-class-name=org.postgresql.Driver

# Инициализация схемы и данных

spring.sql.init.mode=always

spring.sql.init.schema-locations=classpath:schema.sql #Путь до создание таблицы

# Начальные значения если нужны

spring.sql.init.data-locations=classpath:data.sql

# Показ SQL-запросов в консоли

spring.jpa.show-sql=true

Далее sql запросы отправляются с помощью JdbcTemplate

**JPA** – это спецификация для ORM внутри Spring,“под капотом” использует Hibirnate. Чтобы его использовать необходимо установить внутри Gradle:

implementation 'org.springframework.boot:spring-boot-starter-data-jpa'

Далее настраивается она так же почти так же, как **JDBC,** за исключением установки диалекта для определённой субд:  
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.PostgreSQLDialect

Далее создается сущность, объект, который JPA будет отправлять в базу данных, он помечается аннотацией @Entety, далее внутри interface помеченного @Repository, описываются CRUD операций. Самое главное чтобы interface был наследован от JpaRepository <Сущность,тип id >

// Crud операции

1. *Spring Security*

**Spring Security** — это мощный и настраиваемый фреймворк для обеспечения безопасности в приложениях на базе Spring Framework. Он предоставляет широкий набор функций для реализации аутентификации, авторизации и других аспектов безопасности на уровне приложения.

**Для того чтобы работать со spring security необходимо установить аннотацию @EnableWebSecurity,** внутри конфигарциионого файла, и настраивать внутренние бины такие как AuthenticationManager, PasswordEncoder. А так же необходимо довить зависимость в Gradle:

implementation 'org.springframework.boot:spring-boot-starter-security'

**Ключевые компоненты Spring Security:**

* **AuthenticationManager**:
  + **Аутентификация пользователей:** AuthenticationManager — это интерфейс в Spring Security, который отвечает за проверку предоставленных пользователем учетных данных.
  + **Процесс проверки:** Он принимает объект Authentication, содержащий учетные данные пользователя, и пытается аутентифицировать его с помощью одного или нескольких AuthenticationProvider.
  + **Возврат результата:** Если аутентификация успешна, AuthenticationManager возвращает обновленный объект Authentication с информацией о пользователе и его правах доступа.
* **SecurityContext**: Хранит информацию о текущем аутентифицированном пользователе.Получить доступ до контекста мы можем через объект **SecurityContextHolder**
* **GrantedAuthority**: Представляет права и роли пользователя.
* **UserDetailsService**: Интерфейс для загрузки данных пользователя из базы данных или другого источника. Если мы собираемся проверять данных из базы данных, тогда нужно создать собственный сервис, который будет наследоваться от **UserDetailsService**
* **UserDetails:** хранит информацию о пользователе найденному с помощью **UserDetailsService**(логин, пароль, роль и т.д)
* **UsernamePasswordAuthenticationToken:** это класс в Spring Security, реализующий интерфейс **Authentication**. Он используется для представления учетных данных пользователя (имя пользователя и пароль) во время процесса аутентификации и после него.
  + Основные функции:
    - Представление учетных данных:
      * До аутентификации: Хранит введенные пользователем имя пользователя (**principal**) и пароль (**credentials**).
      * После аутентификации: Содержит информацию о пользователе и его правах доступа (**authorities**).
    - Хранение информации о правах доступа:
      * После успешной аутентификации включает коллекцию GrantedAuthority, которая определяет роли и права пользователя.
  + Как работает UsernamePasswordAuthenticationToken:
    - До аутентификации:
      * Создание экземпляра с введенными учетными данными:
      * Authentication authentication = new UsernamePasswordAuthenticationToken(username, password);
    - Состояние:
      * Поле authenticated установлено в false.
      * Используется для передачи учетных данных в AuthenticationManager.
  + Во время аутентификации:
    - Процесс:
      * AuthenticationManager принимает объект Authentication и передает его в соответствующий AuthenticationProvider.
      * AuthenticationProvider проверяет учетные данные (например, сравнивает пароль с сохраненным в базе данных хешем).
    - Результат:
      * Если аутентификация успешна, создается новый UsernamePasswordAuthenticationToken с заполненным списком прав доступа и флагом authenticated = true.
      * Если аутентификация не удалась, выбрасывается исключение AuthenticationException.
  + После аутентификации:
    - Сохранение в SecurityContext:
      * SecurityContextHolder**.**getContext().setAuthentication(authentication);

1. *ORM*

**ORM (Object-Relational Mapping)** — это технология, позволяющая разработчикам работать с реляционными базами данных, используя объектно-ориентированные подходы. Основная цель ORM — упростить процесс взаимодействия между объектами приложения и таблицами базы данных, автоматизируя преобразование данных между двумя различными моделями представления информации.

**ORM создает SQL запрос на основе предоставленных данных и возвращает ответ**

1. *Hibirnate*
2. *JWT*
3. Фильтры – это классы, которые наследуются от определенного класса фильтра, например, OncePerRuqestFilter или RequestContextFilter, они перехватывают запрос(или может срабатывать уже после обработки запроса) и производят какую-то логику. Фильтров может быть много. Чтобы добавить созданный фильтр, надо определить SecurityFilterChain бин внутри конфигурационного файла и добавить его в HttpSecurity
4. *Шаблонизаторы JSP или Thymeleaf*
5. *Глобальные обработки ошибок*
6. *Управление жизненным циклом*

**Ванила Java:**

1. CompletableFuture - это реализация интерфейса Future, которая добавляет возможность явно завершать вычисление (методом complete), а также связывать между собой последовательности асинхронных задач.

## Основные возможности CompletableFuture

1. **Асинхронное выполнение задач**: Вы можете запускать задачи в других потоках без явного управления потоками.
2. **Цепочки колбэков**: Позволяет определить, что должно происходить после завершения асинхронной операции.
3. **Обработка исключений**: Предоставляет методы для обработки возможных исключений, возникших в ходе асинхронных вычислений.

## Пример использования CompletableFuture

import java.util.concurrent.CompletableFuture;

public class Example {

public static void main(String[] args) {

CompletableFuture<Void> future = CompletableFuture.supplyAsync(() -> {

// Асинхронная задача

return compute();

}).thenAccept(result -> {

// Обработка результата

System.out.println("Результат: " + result);

}).exceptionally(ex -> {

// Обработка исключений

System.out.println("Ошибка: " + ex.getMessage());

return null;

});

// Ожидание завершения всех задач

future.join();

}

public static String compute() {

// Некоторая долгосрочная операция

return "Вычисление завершено";

}

}

## Объяснение примера

* **supplyAsync**: Запускает задачу в другом потоке и возвращает CompletableFuture с результатом.
* **thenAccept**: Принимает результат предыдущего вычисления и выполняет действие (в данном случае выводит результат).
* **exceptionally**: Обрабатывает возможные исключения, возникшие в цепочке.

## Цепочка асинхронных задач

Вы можете строить сложные цепочки операций:

CompletableFuture.supplyAsync(() -> {

return stepOne();

}).thenApply(result -> {

return stepTwo(result);

}).thenAccept(finalResult -> {

System.out.println("Итоговый результат: " + finalResult);

});

* **thenApply**: Применяет функцию к результату предыдущего вычисления и возвращает новый CompletableFuture.
* **thenAccept**: Потребляет результат без возвращения нового CompletableFuture.

## Комбинирование нескольких CompletableFuture

Можно объединять несколько асинхронных задач:

CompletableFuture<String> future1 = CompletableFuture.supplyAsync(() -> "Hello");

CompletableFuture<String> future2 = CompletableFuture.supplyAsync(() -> "World");

CompletableFuture<String> combinedFuture = future1.thenCombine(future2, (s1, s2) -> s1 + " " + s2);

combinedFuture.thenAccept(System.out::println);

## Обработка исключений

Методы для обработки исключений:

* **exceptionally**: Обрабатывает исключение и возвращает значение по умолчанию.
* **handle**: Позволяет обработать и результат, и исключение.
* **whenComplete**: Выполняет действие после завершения, но не изменяет результат.

Когда вы используете методы с суффиксом Async, такие как supplyAsync, runAsync, thenApplyAsync, thenAcceptAsync и т.д., задачи выполняются асинхронно в отдельных потоках. По умолчанию эти методы используют общий пул потоков ForkJoinPool.commonPool().

Методы без суффикса Async, такие как thenApply, thenAccept, выполняются в потоке, который завершил предыдущую стадию. Это может быть как основной поток, так и любой другой поток, в котором была выполнена предыдущая задача.

**Пример:**

import java.util.concurrent.CompletableFuture;

public class SyncExample {

public static void main(String[] args) {

CompletableFuture<Void> future = CompletableFuture.supplyAsync(() -> {

// Выполняется в отдельном потоке

return compute();

}).thenAccept(result -> {

// Выполняется в том же потоке, что и предыдущая задача

System.out.println("Результат: " + result);

});

future.join();

}

public static String compute() {

return "Вычисление завершено";

}

}

Вы можете передать свой собственный Executor, чтобы контролировать, в каких потоках будут выполняться задачи. Это полезно, если вы хотите использовать определенный пул потоков или ограничить количество одновременно выполняемых задач.

## Дополнительная информация

* **ForkJoinPool.commonPool()**: Общий пул потоков, используемый по умолчанию для асинхронных задач. Подходит для большинства случаев, но в высоконагруженных приложениях может потребоваться собственный пул потоков.
* **Executor**: Интерфейс для объектов, которые могут выполнять предоставленные им задачи. Позволяет управлять многопоточностью более гибко.

**Пример с кастомным Executor:**

import java.util.concurrent.\*;

public class CustomExecutorExample {

public static void main(String[] args) {

ExecutorService executor = Executors.newFixedThreadPool(2);

CompletableFuture<Void> future = CompletableFuture.supplyAsync(() -> {

// Выполняется в потоке из executor

return compute();

}, executor).thenAcceptAsync(result -> {

// Также выполняется в потоке из executor

System.out.println("Результат: " + result);

}, executor);

future.join();

executor.shutdown();

}

public static String compute() {

return "Вычисление завершено";

}

}

## Итоги

* **Асинхронные методы** (supplyAsync, runAsync, thenApplyAsync, thenAcceptAsync и т.д.) выполняют задачи в отдельных потоках, не блокируя основной поток.
* **Синхронные методы** (thenApply, thenAccept и т.д. без Async) выполняются в том потоке, который завершил предыдущую задачу.
* **Пользовательский Executor**: Передавая свой Executor, вы получаете контроль над тем, какие потоки используются для выполнения задач.

1. Thread

В Java поток (Thread) — это наименьшая единица выполнения в многопоточном программировании. Потоки позволяют выполнять несколько операций одновременно в рамках одного процесса, используя общую память и ресурсы. Это полезно для улучшения производительности и реагирования приложения, особенно когда необходимо выполнять длительные или независимые задачи параллельно.

### Асинхронность и многопоточность

**Асинхронность** в контексте программирования означает выполнение задач без блокирования основного потока управления. В Java асинхронность часто достигается с помощью многопоточности, где задачи выполняются в отдельных потоках, позволяя основной программе продолжать работу.

### Создание и запуск потоков

#### 1. Реализация интерфейса Runnable

Наиболее распространенный способ создать поток — реализовать интерфейс Runnable и передать его экземпляр в конструктор класса Thread.

public class MyRunnable implements Runnable {

@Override

public void run() {

// Код, который будет выполняться в новом потоке

System.out.println("Поток: " + Thread.currentThread().getName());

}

}

public class Main {

public static void main(String[] args) {

Thread thread = new Thread(new MyRunnable());

thread.start(); // Запуск потока

}

}

#### 2. Расширение класса Thread

Также можно создать подкласс Thread и переопределить метод run().

public class MyThread extends Thread {

@Override

public void run() {

System.out.println("Поток: " + Thread.currentThread().getName());

}

}

public class Main {

public static void main(String[] args) {

MyThread thread = new MyThread();

thread.start();

}

}

### Асинхронность внутри потока

Асинхронность внутри потока может быть достигнута путем запуска дополнительных потоков или использования других асинхронных механизмов, таких как CompletableFuture или ExecutorService.

#### Пример: Запуск дополнительных потоков внутри потока

public class NestedThreadsExample {

public static void main(String[] args) {

Thread parentThread = new Thread(() -> {

System.out.println("Родительский поток: " + Thread.currentThread().getName());

Thread childThread = new Thread(() -> {

System.out.println("Дочерний поток: " + Thread.currentThread().getName());

});

childThread.start();

try {

childThread.join(); // Ожидание завершения дочернего потока

} catch (InterruptedException e) {

e.printStackTrace();

}

});

parentThread.start();

}

}

### Использование ExecutorService для управления потоками

ExecutorService предоставляет более гибкий способ управления потоками и задачами.

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

public class ExecutorServiceExample {

public static void main(String[] args) {

ExecutorService executor = Executors.newFixedThreadPool(2);

executor.submit(() -> {

System.out.println("Задача 1 выполняется в потоке: " + Thread.currentThread().getName());

});

executor.submit(() -> {

System.out.println("Задача 2 выполняется в потоке: " + Thread.currentThread().getName());

});

executor.shutdown(); // Остановка приема новых задач и завершение текущих

}

}

### Асинхронные операции с CompletableFuture

1. CompletableFuture позволяет выполнять асинхронные операции без явного создания потоков.

import java.util.concurrent.CompletableFuture;

public class CompletableFutureInsideThread {

public static void main(String[] args) {

Thread thread = new Thread(() -> {

System.out.println("Основной поток: " + Thread.currentThread().getName());

CompletableFuture<Void> future = CompletableFuture.runAsync(() -> {

System.out.println("Асинхронная задача внутри потока: " + Thread.currentThread().getName());

});

future.join(); // Ожидание завершения асинхронной задачи

});

thread.start();

}

}

### Синхронизация при работе с потоками

При работе с общими ресурсами важно обеспечить синхронизацию, чтобы избежать состояния гонки.

public class SynchronizedExample {

private int counter = 0;

public synchronized void increment() {

counter++;

}

public int getCounter() {

return counter;

}

}

1. Thread Factory

ThreadFactory — это интерфейс в Java из пакета java.util.concurrent, который используется для создания новых потоков. Он предоставляет метод newThread(Runnable r), позволяющий настраивать параметры создаваемых потоков при использовании таких инструментов, как ExecutorService и пулы потоков.

**Зачем нужен ThreadFactory?**

По умолчанию при использовании пулов потоков создаются потоки с базовыми настройками. Однако иногда требуется изменить поведение создания потоков:

* **Установка пользовательских имен потоков** для удобства отладки и логирования.
* **Настройка приоритетов потоков**.
* **Определение, будут ли потоки демонами**.
* **Назначение потоков определенной группе потоков (ThreadGroup)**.
* **Обработка необработанных исключений** путем установки собственного UncaughtExceptionHandler.

**Пример использования ThreadFactory**

Создадим кастомный ThreadFactory, который присваивает потокам уникальные имена и устанавливает их как демоны.

java

Copy code

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

import java.util.concurrent.ThreadFactory;

public class CustomThreadFactoryExample {

public static void main(String[] args) {

// Создаем кастомный ThreadFactory

ThreadFactory customThreadFactory = new CustomThreadFactory();

// Создаем ExecutorService с использованием кастомного ThreadFactory

ExecutorService executorService = Executors.newFixedThreadPool(2, customThreadFactory);

// Отправляем задачи на выполнение

for (int i = 0; i < 5; i++) {

executorService.submit(new Task());

}

// Останавливаем ExecutorService

executorService.shutdown();

}

}

// Кастомный ThreadFactory

class CustomThreadFactory implements ThreadFactory {

private int threadId = 1;

private String namePrefix = "CustomThread-";

@Override

public Thread newThread(Runnable r) {

Thread thread = new Thread(r, namePrefix + threadId);

System.out.println("Создан поток: " + thread.getName());

thread.setDaemon(true); // Устанавливаем поток как демон

threadId++;

return thread;

}

}

// Пример задачи

class Task implements Runnable {

@Override

public void run() {

System.out.println("Выполняется задача в потоке: " + Thread.currentThread().getName());

try {

Thread.sleep(1000); // Имитация работы

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

**Объяснение примера**

* **CustomThreadFactory**: реализует интерфейс ThreadFactory и переопределяет метод newThread(Runnable r). В этом методе создается новый поток с уникальным именем, устанавливается как демон и возвращается для использования в пуле потоков.
* **ExecutorService**: создается с помощью Executors.newFixedThreadPool, куда передается наш кастомный ThreadFactory. Это обеспечивает использование настроенных потоков при выполнении задач.
* **Task**: простая реализация Runnable, которая выводит имя текущего потока и имитирует выполнение работы с помощью Thread.sleep(1000).

1. “Погоня за данными” – погоня за данными – это распростанённая ошибка для потоков, когда 2 потока взаимодействует с одной ячейкой и одновременно изменяют её, чтобы избежать при изменении ячеек памяти нужно помечать методы synchronized
2. Паттерны
3. Интерфейс

Интерфейс в Java — это специальный тип, который определяет набор абстрактных методов (методов без реализации), а начиная с Java 8, также может содержать методы с реализацией по умолчанию (default методы) и статические методы. Интерфейсы используются для определения контрактов, которые классы должны выполнять, то есть они указывают, какие методы должны быть реализованы в классах, которые этот интерфейс реализуют.

Интерфейсы могут содержать переменные, но они все public, final и static (являются константами)

**Основные характеристики интерфейсов в Java:**

1. **Абстрактность методов:** По умолчанию все методы в интерфейсе являются публичными и абстрактными (без реализации).
2. **Константы:** Интерфейсы могут содержать переменные, которые автоматически являются public, static и final, то есть это константы.
3. **Множественная реализация:** Класс может реализовывать несколько интерфейсов, что позволяет обходить ограничение одиночного наследования классов в Java.

А так же interface может быть наследован от других interface

1. **Default методы:** С версии Java 8 интерфейсы могут содержать методы с реализацией по умолчанию, обозначенные ключевым словом default.
2. **Статические методы:** Также с Java 8 интерфейсы могут содержать статические методы.
3. **Приватные методы:** Начиная с Java 9, интерфейсы могут иметь приватные методы, которые помогают избежать дублирования кода внутри интерфейса.

**Пример интерфейса:**

public interface Animal {

void eat();

void sleep();

default void breathe() {

System.out.println("Breathing...");

}

}

**Реализация интерфейса в классе:**

public class Dog implements Animal {

@Override

public void eat() {

System.out.println("Dog eats meat.");

}

@Override

public void sleep() {

System.out.println("Dog sleeps in kennel.");

}

}

1. super – вызов конструктора из родительского класса внутри дочернего класса, так же служит для вызова функций из родительского класса
2. Разница между extends и implements

implements озночает что класс имплементирует интерфейс, для него доступны абстрактные методы и необходимо переопределить метод по умолчанию, а наследование - получение методов и переменных от родительского класса. Резюмируя, Implements - для interface, extends для class

Класс может одновременно наследоваться от другого и иплементировать методы

**Git:**

1. Git Merge

**git merge** — это команда в Git, используемая для объединения изменений из одной ветки в другую. Обычно она применяется для интеграции изменений из одной ветки разработки (например, feature-ветки) в основную ветку (например, main или master). Процесс выглядит следующим образом:

Основные этапы работы с git merg:

Переключитесь на целевую ветку: Сначала убедитесь, что вы находитесь на ветке, в которую хотите влить изменения.

**git checkout main**

Запустите команду merge: Выполните git merge с указанием ветки, изменения которой хотите объединить в текущую.

**Основные виды git merge**

* **Fast-forward merge** — если в целевой ветке не было новых коммитов, Git просто сдвигает указатель ветки на последний коммит из объединяемой ветки.
* **Three-way merge** — используется, если обе ветки имели свои коммиты. В таком случае Git создаст новый коммит слияния, объединяющий изменения обеих веток.

### Конфликты при слиянии

При git merge могут возникать конфликты, если одни и те же строки были изменены в обеих ветках. В таком случае Git остановит процесс слияния и предложит вручную разрешить конфликтные изменения

**Определите файлы с конфликтами**: После выполнения команды git merge Git выведет список файлов, в которых обнаружены конфликты. Эти файлы также будут отмечены как "unmerged" при выполнении git status.

1. **Используйте временные файлы**: Когда возникает конфликт, Git создает временные файлы с изменениями каждой ветки. Эти файлы можно получить через команды:
   * **HEAD-версия** (ваша текущая ветка):

git show :1:<имя\_файла> > <имя\_файла>.ours

* + **Версия из сливаемой ветки** (ветка, в которую вы выполняли merge):

git show :2:<имя\_файла> > <имя\_файла>.theirs

1. Эти команды сохранят каждую версию конфликтного файла в виде отдельных копий (<имя\_файла>.ours и <имя\_файла>.theirs), что позволяет увидеть разницу и разрешить конфликт вручную.
2. **Используйте git checkout для выбора конкретной версии файла**: Если вы хотите полностью заменить файл на одну из версий (HEAD или сливаемой ветки), используйте:
   * Для сохранения версии из вашей текущей ветки:

git checkout --ours <имя\_файла>

* + Для сохранения версии из сливаемой ветки:

bash

Copy code

git checkout --theirs <имя\_файла>

1. **Используйте git diff для просмотра различий**: Чтобы увидеть различия между конфликтующими версиями в терминале, вы можете использовать:

git diff --ours --theirs <имя\_файла>

Эти методы помогут вам понять, какие изменения были в каждой из конфликтующих версий и сохранить их копии для дальнейшего анализа.

1. Pull request

Чтобы сделать pull request нам необходимо создать ветку, загрузить данные в неё:

git checkout -b feature-branch

**Слияние pull request**

Когда изменения одобрены, их можно интегрировать в основную ветку. Это можно сделать через интерфейс системы контроля версий, выбрав опцию «Merge pull request». Существует несколько стратегий слияния:

gh pr create --base main --head feature-branch --title "Название PR" --body "Описание изменений"

--base — ветка, в которую вы хотите влить изменения (обычно main или master). --head — ветка с вашими изменениями.

* **Merge commit**: создает отдельный коммит слияния, сохраняющий историю всех коммитов из PR.
* **Squash and merge**: объединяет все коммиты из PR в один.
* **Rebase and merge**: переписывает коммиты PR так, будто они созданы поверх основной ветки, делая историю линейной.

Во-первых, найдите номер или URL pull request, который вы хотите слить. Обычно номер PR отображается рядом с ним в веб-интерфейсе GitHub (например, #42).

Вы можете также увидеть все открытые pull requests в репозитории, используя:

gh pr list

### 2. Выполните слияние pull request

Для слияния pull request используйте команду gh pr merge, указав номер или URL pull request. Вот несколько вариантов слияния:

* **Стандартное слияние** (Merge commit):

gh pr merge <номер или URL PR>

* **Squash and merge** (объединение всех коммитов в один перед слиянием):

gh pr merge <номер или URL PR> --squash

* **Rebase and merge** (переписывает коммиты PR как отдельные коммиты в основной ветке):

gh pr merge <номер или URL PR> --rebase

### 3. Дополнительные параметры

* Чтобы автоматически удалить ветку после слияния, добавьте флаг --delete-branch:

gh pr merge <номер или URL PR> --delete-branch

* Вы можете также добавить описание для слияния с помощью флага --body:

gh pr merge <номер или URL PR> --body "Описание слияния"

### Пример команды для слияния с удалением ветки

gh pr merge 42 --squash --delete-branch

### Завершение

После выполнения команды gh pr merge GitHub CLI покажет статус слияния, и pull request будет закрыт.

**5. Закрытие pull request**

После успешного слияния pull request закрывается. В случае, если изменения больше не актуальны или отменены, pull request можно закрыть без слияния.

1. Git add . – добаление только изменных или удаленных файлов
2. Git release

Команда **gh release** в GitHub CLI предназначена для управления релизами в репозитории. С её помощью можно создавать, просматривать, редактировать и удалять релизы прямо из командной строки. Релизы в GitHub используются для публикации стабильных версий кода с дополнительной информацией, такими как заметки о выпуске и бинарные файлы.

### Основные команды gh release

#### 1. Создание релиза

Чтобы создать новый релиз, используйте команду **gh release create** с указанием тега, который будет использоваться как версия релиза (например, v1.0.0):

gh release create <tag> --title "Название релиза" --notes "Описание релиза"

* **<tag>** — тег релиза, который обычно представляет версию (например, v1.0.0). Если такого тега нет, GitHub CLI создаст его.
* **--title** — заголовок релиза.
* **--notes** — описание изменений, включенных в релиз.

#### 2. Добавление файлов к релизу

Вы можете также добавить файлы к релизу (например, сборки, архивы или исполняемые файлы):

gh release create <tag> path/to/file1 path/to/file2 --title "Релиз с файлами" --notes "Содержит бинарные файлы."

### 3. Просмотр списка релизов

Чтобы увидеть все релизы в репозитории, используйте:

**gh release list**

Эта команда отобразит список всех релизов с указанием тегов и названий.

#### 4. Просмотр конкретного релиза

Чтобы получить информацию о конкретном релизе, выполните команду:

**gh release view** <tag>

Здесь <tag> — это тег релиза, который вы хотите просмотреть. Вы также можете добавить --web, чтобы открыть релиз в браузере.

#### 5. Редактирование релиза

Чтобы внести изменения в существующий релиз, например, изменить заметки о выпуске, используйте команду:

**gh release edit** <tag> --title "Обновленный релиз" --notes "Обновленные заметки к релизу."

#### 6. Удаление релиза

**gh release delete** <tag>

### Полезные опции команды gh release create

* **--prerelease** — отмечает релиз как предварительный, если версия еще не готова для полной публикации.
* **--generate-notes** — автоматически генерирует заметки на основе последних коммитов и PR, связанных с релизом.

1. Git rebase – Это альтернатива, позволяющая "переписать" историю коммитов, чтобы объединить изменения более линейно, без создания дополнительного коммита слияния.
2. Работа с ветками

### 1. Создание новой ветки

Чтобы создать новую ветку, используется команда git branch:

### 2. Переключение на ветку

Чтобы начать работать в новой ветке, нужно переключиться на неё с помощью

git checkout:

Для более удобного создания и перехода в новую ветку можно использовать:

git checkout -b <имя\_ветки>

### 3. Просмотр существующих веток

Чтобы увидеть все ветки в репозитории, выполните:

git branch

### 5. Удаление ветки

Когда ветка больше не нужна, её можно удалить:

Для удаления локальной ветки:

git branch -d <имя\_ветки>

Флаг -d (от слова "delete") удалит ветку, если её изменения уже слиты в основную ветку. Если же изменения не были слиты, Git не позволит удалить ветку, чтобы случайно не потерять работу. Чтобы удалить ветку принудительно, используйте флаг -D:

git branch -D <имя\_ветки>

* Для удаления удалённой ветки:

git push origin --delete <имя\_ветки>

### Отправка ветки на удалённый репозиторий

Если вы хотите поделиться своей веткой с другими или сохранить её на удалённом сервере, отправьте её на удалённый репозиторий:

git push origin <имя\_ветки>

### 7. Слежение за удалёнными ветками

Чтобы создать локальную ветку, отслеживающую удалённую, используйте:

git checkout -b <локальная\_ветка> origin/<удалённая\_ветка>

Эта команда создаёт локальную ветку, которая будет синхронизироваться с указанной удалённой.

### 8. Ребейзинг (перенос изменений)

Команда git rebase позволяет перенести коммиты одной ветки на другую, сохраняя линейную историю. Например:

1. Переключитесь на ветку, которую хотите перенести:

git checkout feature-branch

git rebase main

Эта команда перенесёт все коммиты из feature-branch так, будто они были созданы после последних коммитов в main. После ребейза история становится линейной, без дополнительных коммитов слияния.

**SQL**

 **SQL (Structured Query Language):** Язык для управления и манипуляции реляционными базами данных.

 **Типы команд SQL:**

* **DDL (Data Definition Language):** CREATE, ALTER, DROP
* **DML (Data Manipulation Language):** SELECT, INSERT, UPDATE, DELETE
* **DCL (Data Control Language):** GRANT, REVOKE
* **TCL (Transaction Control Language):** COMMIT, ROLLBACK, SAVEPOINT

### ****2.1. Создание базовых таблиц****

**Синтаксис:**

CREATE TABLE table\_name (

column1 datatype constraints,

column2 datatype constraints,

...

);

### ****.2. Типы данных****

* **Числовые:** INT, DECIMAL, FLOAT, DOUBLE
* **Строковые:** VARCHAR, CHAR, TEXT
* **Дата и время:** DATE, TIME, DATETIME, TIMESTAMP
* **Логические:** BOOLEAN

## **3. Манипуляция данными (DML)**

### ****3.1. Вставка данных****

**Синтаксис:**

INSERT INTO table\_name (column1, column2, ...)

VALUES (value1, value2, ...);

**Пример:**

INSERT INTO Employees (EmployeeID, FirstName, LastName, DepartmentID, Salary)

VALUES (1, 'Иван', 'Иванов', 101, 50000.00);

### ****3.2. Обновление данных****

**Синтаксис:**

sql

Copy code

UPDATE table\_name

SET column1 = value1, column2 = value2, ...

WHERE condition;

**Пример:**

UPDATE Employees

SET Salary = Salary \* 1.1

WHERE DepartmentID = 101;

**Синтаксис:**

DELETE FROM table\_name

WHERE condition;

**Пример:**

DELETE FROM Employees

WHERE EmployeeID = 1;

## **4. Запросы и фильтрация данных**

### ****4.1. Основной SELECT запрос****

**Синтаксис:**

SELECT column1, column2, ...

FROM table\_name;

**Пример:**

sql

Copy code

SELECT FirstName, LastName FROM Employees;

### ****4.2. Фильтрация с помощью WHERE****

**Синтаксис:**

SELECT columns

FROM table\_name

WHERE condition;

**Пример:**

SELECT \* FROM Employees

WHERE Salary > 40000;

### ****4.3. Сортировка результатов****

**Синтаксис:**

SELECT columns

FROM table\_name

ORDER BY column1 [ASC|DESC], column2 [ASC|DESC], ...;

**Пример:**

SELECT FirstName, LastName, Salary

FROM Employees

ORDER BY Salary DESC;

**СУБД**

1. MYSQL

Система управления базами данных MySql содержит в себе 2 программы, MySQL Shell (командную строку) и MySQL WorkBench (UI интерфейс управления базой данных)

Базовый логин: root

Базовый port: 3306

1. PostgreSQL

Система управления базами данных PostgreSQL – содержит в себе программу pgAdmin, для управления базами данных

Базовый логин: postgres

Базовый порт:5432

**Kafka**

**Микро-Сервисы:**

**Оркестрирование**

**Тестирование:**

**Spring MVC**

**AspectJ**

**Spring Reactive Web**

**Docker**