1. Принципы проектирования и разработки

1.1. SOLID

Single Responsibility Principle (Принцип единственной ответственности): Каждый класс должен иметь одну причину для изменения. Это упрощает поддержку и тестирование. **Пример**: Класс UserService отвечает только за бизнес-логику пользователей, а UserRepository — за доступ к данным. **Нарушение**: Класс UserService обрабатывает и логику, и доступ к БД, и отправку email.

Open/Closed Principle (Принцип открытости/закрытости): Классы должны быть открыты для расширения (например, через наследование), но закрыты для модификации. **Пример**: Реализация новых стратегий через интерфейс PaymentStrategy (например, CreditCardPayment, PayPalPayment) без изменения базового класса PaymentProcessor. **Нарушение**: Изменение метода processPayment для каждого нового типа оплаты.

Liskov Substitution Principle (Принцип подстановки Барбары Лисков): Объекты подкласса должны быть полностью взаимозаменяемы с объектами родительского класса без изменения поведения программы. **Пример**: Если Square наследуется от Rectangle, метод setWidth не должен нарушать поведение Rectangle. **Нарушение**: Square изменяет setWidth, так что площадь вычисляется некорректно.

Interface Segregation Principle (Принцип разделения интерфейсов): Клиенты не должны зависеть от интерфейсов, которые они не используют. **Пример**: Разделение интерфейса Worker на Eatable и Workable, чтобы класс Robot реализовал только Workable. **Нарушение**: Один общий интерфейс Worker с методами eat() и work(), который заставляет Robot реализовать ненужный eat().

Dependency Inversion Principle (Принцип инверсии зависимостей): Модули верхнего уровня не должны зависеть от конкретных реализаций, а от абстракций. **Пример**: UserService зависит от интерфейса UserRepository, а не от MySQLUserRepository. **Код**:

java

```java
public interface UserRepository {
    User findById(Long id);
}

@Component
public class MySQLUserRepository implements UserRepository {
    public User findById(Long id) { /* Реализация */ }
}

@Component
public class UserService {
```

```
    private final UserRepository repository;

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

Вопросы с собеседования:

- Как нарушение SRP влияет на поддержку кода?
- Приведите пример реализации OCP в реальном проекте.
- Как проверить, нарушает ли класс LSP?

Совет: Нарисуйте UML-диаграмму для SOLID на бумаге, чтобы объяснить принципы визуально.

1.2. Принципы ООП

- **Инкапсуляция**: Скрытие внутренней реализации через модификаторы доступа (private) и предоставление интерфейса через геттеры/сеттеры. **Пример**: Поле private String password доступно только через getPassword() с проверкой. **Код**:

  java

  ```
  public class User {
      private String password;

      public String getPassword() {
          return password != null ? "****" : null; // Скрытие данных
      }

      public void setPassword(String password) {
          this.password = password;
      }
  }
  ```

- **Наследование**: Переиспользование кода через extends. Используйте с осторожностью, чтобы не нарушить LSP. **Пример**: Класс Dog наследуется от Animal и переопределяет метод speak().
- **Полиморфизм**: Вызов метода зависит от типа объекта во время выполнения. **Пример**:

  java

```java
public abstract class Animal {
    public abstract String speak();
}

public class Dog extends Animal {
    @Override
    public String speak() {
        return "Woof!";
    }
}

Animal animal = new Dog();
System.out.println(animal.speak()); // Вывод: Woof!
```

- **Абстракция**: Сокрытие деталей реализации через абстрактные классы или интерфейсы. **Пример**: Интерфейс Shape с методом calculateArea().

Вопросы с собеседования:

- Чем отличается инкапсуляция от абстракции?
- Когда использовать интерфейсы, а когда абстрактные классы?

Совет: Подготовьте пример кода с полиморфизмом, чтобы показать разницу между статическим и динамическим полиморфизмом.

1.3. Принципы разработки

- **DRY (Don't Repeat Yourself)**: Избегайте дублирования кода, вынося общую логику в методы или классы. **Пример**: Создание утилитного класса StringUtils для форматирования строк вместо повторения кода.
- **KISS (Keep It Simple, Stupid)**: Пишите понятный и минималистичный код. **Пример**: Используйте for-each вместо сложных циклов с индексами, если это не требуется.

- **YAGNI (You Ain't Gonna Need It)**: Реализуйте только необходимые функции. **Пример**: Не добавляйте сложную систему кэширования, если текущие требования этого не предусматривают.

Вопросы с собеседования:

- Как вы применяете DRY в своих проектах?
- Пример, когда KISS помог упростить архитектуру?

Совет: Приведите пример рефакторинга кода, где вы устранили дублирование (DRY).

1.4. Паттерны проектирования

- **Порождающие**:
  - **Singleton**: Гарантирует один экземпляр класса. **Код**:

    java

    ```
    public class Singleton {
        private static final Singleton INSTANCE = new Singleton();
        private Singleton() {}
        public static Singleton getInstance() {
            return INSTANCE;
        }
    }
    ```

  - **Factory Method**: Создание объектов через фабрику. **Пример**: Фабрика ConnectionFactory для создания соединений к разным БД.
  - **Builder**: Пошаговое создание сложных объектов. **Пример**: User.builder().setName("John").setAge(30).build().
- **Структурные**:
  - **Adapter**: Преобразование интерфейса одного класса в другой. **Пример**: Адаптер для работы со старой системой оплаты через новый интерфейс.
  - **Decorator**: Динамическое добавление функционала. **Пример**: Добавление логирования к сервису через декоратор.
  - **Proxy**: Контроль доступа или ленивая загрузка. **Пример**: Прокси для проверки прав доступа перед вызовом метода.
- **Поведенческие**:
  - **Observer**: Уведомление объектов об изменениях. **Пример**: Система подписки на события (например, PropertyChangeListener).
  - **Strategy**: Выбор алгоритма во время выполнения. **Пример**: Разные алгоритмы сортировки (QuickSort, MergeSort).

- **Command**: Инкапсуляция запросов в объекты. **Пример**: Команды "включить/выключить свет" в системе умного дома.

Вопросы с собеседования:

- Какой паттерн вы использовали в проекте и почему?
- Чем отличается Decorator от Proxy?

Совет: Напишите реализацию Singleton и Builder, чтобы показать на собеседовании.

1.5. Big O

Оценка сложности алгоритмов по времени и памяти:

- O(1): Доступ к элементу массива или HashMap.get().
- O(log n): Бинарный поиск.
- O(n): Линейный поиск, проход по списку.
- O(n log n): Эффективные алгоритмы сортировки (QuickSort, MergeSort).
- O(n^2): Пузырьковая сортировка, вложенные циклы.

Пример: Сравнение сложности для поиска:

java

```java
// O(n) - Линейный поиск
public int linearSearch(int[] arr, int target) {
    for (int i = 0; i < arr.length; i++) {
        if (arr[i] == target) return i;
    }
    return -1;
}

// O(log n) - Бинарный поиск
public int binarySearch(int[] arr, int target) {
    int left = 0, right = arr.length - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (arr[mid] == target) return mid;
        if (arr[mid] < target) left = mid + 1;
        else right = mid - 1;
    }
```

```
    return -1;
}
```

Вопросы с собеседования:

- Как выбрать структуру данных в зависимости от Big O?
- Почему HashMap.get() в среднем O(1)?

Совет: Разберите 2–3 задачи на LeetCode, объясняя сложность решения.

1.6. Микросервисная архитектура

Плюсы:

- Масштабируемость: Каждый сервис масштабируется независимо.
- Гибкость: Разные технологии для разных сервисов.
- Независимый деплой: Обновление одного сервиса без остановки других.

Минусы:

- Сложность мониторинга: Требуются инструменты (Prometheus, Grafana).
- Распределённые транзакции: Сложности с согласованностью.
- Сетевая задержка: Вызовы между сервисами увеличивают latency.

Паттерны микросервисов:

- **API Gateway**: Единая точка входа (например, Spring Cloud Gateway). **Пример**: Маршрутизация запросов /users к сервису пользователей.
- **Service Discovery**: Динамическое обнаружение сервисов (Eureka, Consul).
- **Circuit Breaker**: Защита от сбоев (Hystrix, Resilience4j). **Пример**: Откат на резервный сервис при сбое основного.
- **Event Sourcing**: Хранение событий вместо текущего состояния. **Пример**: Лог операций для восстановления состояния заказа.
- **Saga**: Управление транзакциями через последовательность локальных операций. **Пример**: Сага для заказа: создание → оплата → доставка.
- **CQRS**: Разделение операций чтения и записи. **Пример**: Отдельные модели для чтения (UserDTO) и записи (UserEntity).

Вопросы с собеседования:

- Как обеспечить согласованность данных в микросервисах?
- Когда использовать Saga вместо двухфазного коммита?

Совет: Изучите Netflix OSS (Eureka, Hystrix) и напишите пример REST API с Gateway.

2. Java Collections Framework

2.1. Общая структура

- **Интерфейсы**:
  - Collection: List (упорядоченный), Set (уникальные элементы), Queue (FIFO/LIFO).
  - Map: Хранение пар ключ-значение.
- **Реализации**:
  - List: ArrayList, LinkedList, Vector.
  - Set: HashSet, TreeSet, LinkedHashSet.
  - Queue: PriorityQueue, ArrayDeque.
  - Map: HashMap, TreeMap, LinkedHashMap.

2.2. ArrayList vs LinkedList

Коллекция	Структура	Доступ по индексу	Вставка/Удаление	Пример использования
ArrayList	Динамический массив	O(1)	O(n) (сдвиг элементов)	Списки с частым доступом
LinkedList	Двусвязный список	O(n)	O(1) (изменение ссылок)	Очереди, частые вставки

Код:

java

```java
List<String> arrayList = new ArrayList<>();
arrayList.add("Item"); // O(1) в конец, O(n) в середину
System.out.println(arrayList.get(0)); // O(1)

List<String> linkedList = new LinkedList<>();
linkedList.addFirst("Item"); // O(1)
```

2.3. HashMap

- **Структура**: Массив бакетов (Node<K,V>[]), где каждый бакет — список или дерево (при коллизиях).
- **Хэширование**: index = hash(key) & (n - 1), где n — размер массива.
- **loadFactor**: 0.75 — порог для увеличения массива (resize).
- **Коллизии**: Решаются цепочками (до Java 8) или красно-чёрными деревьями (Java 8+ при большом количестве коллизий).

Код:

java

```
Map<String, Integer> map = new HashMap<>();
map.put("key", 1); // O(1) в среднем
System.out.println(map.get("key")); // O(1) в среднем
```

2.4. Queue/Stack

- **Queue (FIFO)**: offer(), poll(), peek(). **Пример**: PriorityQueue для задач по приоритету.
- **Stack (LIFO)**: push(), pop(), peek(). **Пример**: ArrayDeque для реализации стека.

Код:

java

```
Deque<String> stack = new ArrayDeque<>();
stack.push("Item"); // O(1)
System.out.println(stack.pop()); // O(1)
```

Вопросы с собеседования:

- Чем отличается HashMap от ConcurrentHashMap?
- Когда использовать LinkedList вместо ArrayList?

Совет: Напишите код с использованием Stream API для обработки коллекций, например, фильтрация и группировка.

3. Исключения

3.1. Checked vs Unchecked

- **Checked**: Проверяются компилятором, должны быть обработаны (try-catch) или объявлены (throws). **Пример**: IOException, SQLException.
- **Unchecked**: Возникают во время выполнения, не требуют обработки. **Пример**: NullPointerException, IllegalArgumentException.

Код:

java

```
public void readFile(String path) throws IOException { // Checked
    throw new IOException("File not found");
}

public void divide(int a, int b) { // Unchecked
    if (b == 0) throw new ArithmeticException("Division by zero");
}
```

3.2. Иерархия

- Throwable: Базовый класс.
  - Error: Серьёзные сбои (OutOfMemoryError, StackOverflowError).
  - Exception: Проверяемые и непроверяемые исключения.

Вопросы с собеседования:

- Как создать кастомное исключение?
- Когда использовать throws против try-catch?

Совет: Напишите кастомное исключение и его обработку.

4. Spring / Spring Boot

4.1. Spring Core

- **IoC (Inversion of Control)**: Контейнер Spring управляет созданием и жизненным циклом объектов. **Пример**: @Component создаёт бин, который Spring инжектирует через @Autowired.
- **DI (Dependency Injection)**: Внедрение зависимостей через конструктор, сеттер или поля. **Код**:

  java

  ```java
  @Component
  public class UserService {
      private final UserRepository repository;

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

- **ApplicationContext**: Контейнер для бинов, поддерживает конфигурацию через XML, Java или аннотации.

4.2. Scope бинов

- singleton: Один экземпляр на приложение (по умолчанию).
- prototype: Новый экземпляр при каждом запросе.
- request, session, application: Для веб-приложений.

Код:

java

```java
@Bean
@Scope("prototype")
public MyBean myBean() {
    return new MyBean();
```

```
}
```

4.3. Создание и инжектирование бинов

- **Создание**: XML, Java Config (@Configuration), аннотации (@Component, @Service, @Repository).
- **Инжектирование**:
  - @Autowired: Автоматическое внедрение.
  - @Qualifier: Уточнение бина при конфликте.
  - @Primary: Приоритетный бин.

Код:

java

```java
@Configuration
public class AppConfig {
    @Bean
    @Primary
    public UserRepository userRepository() {
        return new MySQLUserRepository();
    }
}
```

4.4. Транзакции

- @Transactional: Обеспечивает атомарность операций. **Пример**: Сохранение пользователя и его профиля в одной транзакции.
- **Подводные камни**:
  - LazyInitializationException: Доступ к ленивым данным вне сессии.
  - Транзакции не работают на private методах или без прокси.

Код:

java

```java
@Service
public class UserService {
    @Transactional
    public void saveUser(User user) {
        userRepository.save(user);
```

```
    }
}
```

4.5. Spring Boot

- **Особенности**:
  - Автонастройка: Стартеры (spring-boot-starter-web, spring-boot-starter-data-jpa).
  - Встроенный сервер: Tomcat, Jetty.
  - Упрощённая конфигурация: application.properties.
- **@SpringBootApplication**:
  - @Configuration: Определение бинов.
  - @EnableAutoConfiguration: Автонастройка.
  - @ComponentScan: Сканирование компонентов.

Код (application.properties):

properties

```
spring.datasource.url=jdbc:mysql://localhost:3306/mydb
spring.datasource.username=root
spring.datasource.password=password
```

Вопросы с собеседования:

- Как работает @Autowired под капотом?
- Как избежать LazyInitializationException?

Совет: Настройте Spring Boot проект с REST API и базой данных.

5. JPA / Hibernate

5.1. Основы

- **JPA**: Спецификация для ORM.
- **Hibernate**: Реализация JPA, добавляет дополнительные возможности (HQL, кэширование).
- **Аннотации**:
  - @Entity: Объект, связанный с таблицей.
  - @Id: Первичный ключ.
  - @OneToMany, @ManyToOne: Связи между сущностями.

Код:

java

```java
@Entity
@Table(name = "users")
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(name = "username")
    private String name;

    @OneToMany(mappedBy = "user")
    private List<Order> orders;
}
```

5.2. Требования к Entity

- Публичный класс.
- Безаргументный конструктор.
- Поле с @Id.
- Рекомендуется: Реализация Serializable.

5.3. Состояния сущности

- Transient: Не связана с БД.
- Persistent: В сессии Hibernate, синхронизируется с БД.
- Detached: Отключена от сессии.
- Removed: Помечена для удаления.

5.4. Кэширование

- **1-й уровень**: В рамках сессии, автоматически.
- **2-й уровень**: На уровне SessionFactory, требует настройки (например, EhCache).

5.5. HQL vs SQL

- **HQL**: Запросы к объектам (from User where name = :name).
- **SQL**: Запросы к таблицам (SELECT * FROM users).

Код (HQL):

java

```
@Query("SELECT u FROM User u WHERE u.name = :name")
List<User> findByName(@Param("name") String name);
```

5.6. Подводные камни

- **N+1 проблема**: Многократные запросы при ленивой загрузке. **Решение**: Используйте JOIN FETCH или @EntityGraph.
- **LazyInitializationException**: Доступ к ленивым данным вне сессии. **Решение**: Используйте Eager загрузку или откройте сессию.

Вопросы с собеседования:

- Как настроить связи @OneToMany?
- Как оптимизировать запросы для избежания N+1?

Совет: Напишите пример репозитория с HQL и JOIN FETCH.

6. Базы данных

6.1. ACID

- **Atomicity**: Транзакция неделима.
- **Consistency**: Данные остаются в согласованном состоянии.
- **Isolation**: Транзакции изолированы.
- **Durability**: Зафиксированные изменения сохраняются.

6.2. Аномалии

- **Грязное чтение**: Чтение незафиксированных данных.
- **Неповторяемое чтение**: Изменение данных в другой транзакции.
- **Фантомное чтение**: Появление новых строк.

6.3. Уровни изоляции

- READ UNCOMMITTED: Возможны все аномалии.
- READ COMMITTED: Исключает грязное чтение.
- REPEATABLE READ: Исключает неповторяемое чтение.
- SERIALIZABLE: Полная изоляция, но низкая производительность.

6.4. JOIN

- INNER JOIN: Только совпадающие записи.
- LEFT JOIN: Все записи из левой таблицы.
- RIGHT JOIN: Все записи из правой таблицы.
- FULL JOIN: Все записи из обеих таблиц.
- CROSS JOIN: Все возможные комбинации.

Код (SQL):

sql

```sql
SELECT u.name, o.order_date
FROM users u
LEFT JOIN orders o ON u.id = o.user_id;
```

6.5. WHERE vs HAVING

- WHERE: Фильтрация строк до агрегации.
- HAVING: Фильтрация после GROUP BY.

Код:

sql

```
SELECT department, COUNT(*)
FROM employees
WHERE salary > 50000
GROUP BY department
HAVING COUNT(*) > 5;
```

Вопросы с собеседования:

- Какой уровень изоляции выбрать для высокой конкуренции?
- Как оптимизировать запрос с JOIN?

Совет: Напишите SQL-запросы с JOIN и агрегатными функциями.\

7. Java 8+

- **Stream API**: Функциональная обработка коллекций. **Код**:

  java

  ```
  List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
  List<Integer> evenSquares = numbers.stream()
      .filter(n -> n % 2 == 0)
      .map(n -> n * n)
      .collect(Collectors.toList()); // [4, 16]
  ```

- **Лямбда-выражения**: Анонимные функции. **Пример**: x -> x * 2.
- **Optional**: Избежание NullPointerException. **Код**:

  java

  ```
  Optional<String> name = Optional.ofNullable(user.getName());
  String result = name.orElse("Unknown");
  ```

Вопросы с собеседования:

- Как оптимизировать Stream API для больших данных?
- Чем Optional лучше проверки null?

Совет: Практикуйтесь с Stream API на задачах фильтрации и группировки.

8. JMS (Messaging)

- **RabbitMQ**: Очереди сообщений, протокол AMQP. **Пример**: Отправка сообщений о заказах в очередь.
- **Kafka**: Потоковая обработка, хранение логов. **Пример**: Обработка событий в реальном времени.
- **ActiveMQ**: Поддержка JMS API, очереди и топики.

Код (Spring + RabbitMQ):

java

```java
@Service
public class MessageProducer {
    @Autowired
    private RabbitTemplate rabbitTemplate;

    public void sendMessage(String message) {
        rabbitTemplate.convertAndSend("myQueue", message);
    }
}
```

Вопросы с собеседования:

- Чем Kafka отличается от RabbitMQ?
- Как обеспечить доставку сообщений?

Совет: Настройте простую очередь в RabbitMQ.

9. HTTP, REST, SOAP

- **HTTP**: Протокол с методами GET, POST, PUT, DELETE. **Пример**: GET /users/1 для получения пользователя.
- **REST**: Архитектурный стиль, ресурсы + HTTP + JSON. **Код**:

  java

  ```java
  @RestController
  @RequestMapping("/users")
  public class UserController {
      @GetMapping("/{id}")
      public User getUser(@PathVariable Long id) {
          return userService.findById(id);
      }
  }
  ```

- **SOAP**: XML + WSDL, строгий контракт.

Вопросы с собеседования:

- Чем REST отличается от SOAP?
- Как обеспечить идемпотентность в REST?

Совет: Напишите REST API с @RestController.

10. Тестирование

- **Unit**: Тестирование отдельных методов (JUnit, Mockito). **Код**:

  java

  ```java
  @Test
  public void testUserService() {
      UserRepository mockRepo = mock(UserRepository.class);
      when(mockRepo.findById(1L)).thenReturn(new User(1L, "John"));
      UserService service = new UserService(mockRepo);
      assertEquals("John", service.findById(1L).getName());
  }
  ```

- **Integration**: Проверка взаимодействия модулей.
- **E2E**: Тестирование пользовательских сценариев.

Вопросы с собеседования:

- Как мокать зависимости в тестах?
- Чем отличаются unit и integration тесты?

Совет: Напишите тесты для REST контроллера.

11. Git

- **merge**: Объединяет ветки, сохраняя историю.
- **rebase**: Переписывает историю для линейности.
- **reset**: Откат изменений (--hard, --soft).
- **revert**: Создаёт коммит-откат.
- **squash**: Объединяет коммиты в один.
- **cherry-pick**: Перенос конкретного коммита.

Команды:

bash

```
git merge feature-branch
git rebase main
git reset --hard HEAD~1
git revert <commit-id>
git cherry-pick <commit-id>
```

Вопросы с собеседования:

- Как разрешить конфликты при merge?
- Когда использовать rebase вместо merge?

Совет: Практикуйтесь с ветками и конфликтами в локальном репозитории.

12. DevOps

- **CI/CD**: Автоматизация сборки, тестирования, деплоя (Jenkins, GitHub Actions).
- **Docker**: Контейнеризация приложений. **Команда**:

  bash

  ```
  docker build -t my-app .
  docker run -p 8080:8080 my-app
  ```

- **Kubernetes**: Оркестрация контейнеров.
- **Мониторинг**: Prometheus, Grafana.
- **Логи**: ELK Stack (Elasticsearch, Logstash, Kibana).

Вопросы с собеседования:

- Как настроить CI/CD для Spring Boot?
- Чем Docker отличается от виртуальной машины?

Совет: Настройте Docker для Spring Boot приложения.

Рекомендации по подготовке

1. **Практика кода**:
   - Решайте задачи на LeetCode (алгоритмы, коллекции).
   - Напишите REST API с Spring Boot и JPA.
   - Реализуйте паттерны (Singleton, Factory, Strategy).
2. **Теория**:
   - Объясняйте термины простыми словами. Пример: "IoC — это когда Spring берёт на себя создание объектов, а я только указываю зависимости".
   - Подготовьте ответы на вопросы о подводных камнях (N+1, LazyInitializationException).
3. **SQL**: Напишите запросы с JOIN, GROUP BY, агрегатными функциями.
4. **Проект**: Создайте небольшой проект (REST API с базой данных и тестами).
5. **Мок-собеседование**: Попросите друга задать вопросы из методички.
6. **Документация**: Изучите Spring Docs, Hibernate Docs, Java API.