**Легенда для «Детали Торговли» (Март 2023 — настоящее время)**

**Суть проекта:** Интернет-магазин сложного торгового оборудования (стеллажи, витрины, кассовые боксы) с большим количеством модификаций и характеристик.

**Ваша роль:** Вы пришли в уже работающий проект. Основа была на jQuery, но руководство приняло решение постепенно модернизировать фронтенд на React.

**Ключевые детали для рассказа:**

* **Проект:** "Унаследованный проект (legacy). Изначально это был монолит на jQuery, который уже плохо масштабировался и поддерживался."
* **Команда:** "Мы работали в связке с одним бэкендером. Дизайнер присылал макеты в Figma, я занимался всей фронтенд-частью."
* **Зачем понадобился React?** "Главная боль — это каталог с сложными фильтрами (по размерам, материалу, нагрузке). На jQuery это была огромная, нечитаемая логика. Я предложил и реализовал переписывание этого функционала на React + Redux Toolkit. Это дало мгновенный эффект: код стал структурированным, а интерфейс отзывчивым."
* **Про "легаси":** "Моя ключевая задача была — не сломать старое. Новые компоненты на React встраивались в существующую jQuery-структуру. Например, страница товара осталась на jQuery, а блок с фильтрами и списком товаров рядом — уже был на React."
* **Про технологии (если спросят детально):**
  + **FancyBox:** "Использовали для галереи товаров, где нужно было показывать оборудование с разных ракурсов."
  + **Bootstrap:** "На нем была построена вся сетка и базовые компоненты. Я адаптировал и дорабатывал стили под дизайн."
  + **OWL Carousel:** "Использовался для блоков 'Похожие товары' и 'Вы недавно смотрели'."

**Пример ответа на вопрос "Расскажите о вашем последнем проекте":**  
"Мой текущий проект — это интернет-магазин торгового оборудования. Я пришел на этапе, когда нужно было модернизировать устаревший код на jQuery. Моей главной задачей была разработка новой, современной системы фильтрации для сложного каталога на React и Redux Toolkit. Я интегрировал ее в существующий проект, обеспечив полную совместимость, что значительно улучшило производительность и удобство поддержки кода. Параллельно занимался доработкой интерфейса, внедрением ленивой загрузки и работой с легаси-кодом."

### ****Часть 1: Вопросы от рекрутера****

**1. Почему решил поменять место работы?**  
(Для твоего текущего места в "Детали Торговли")  
**Ответ:** "Я благодарен своей текущей компании за опыт и возможности. За время моей работы мы успешно провели основную фазу модернизации проекта — перевели ключевые модули с jQuery на современный стек (React/TypeScript/RTK). Сейчас проект стабилен и развивается уже в рамках обновленной архитектуры. Я чувствую, что основные сложные и интересные задачи в рамках этого тех stack'а решены, и я ищу новый вызов. Мне хочется применить свой опыт работы с легаси и миграцией в более крупном или сложном продукте, возможно, с более современным изначально стеком или в более большой и опытной команде, где будет возможность углубиться в архитектурные вопросы и менторство."

* Что мы подчеркиваем: благодарность, завершенность этапа, профессиональный рост, желание новых вызовов. Никакого негатива к текущему работодателю.

**2. Расскажи про последний проект: задачи, команда**  
(Используем легенду для "Детали Торговли")  
**Ответ:** "Мой последний проект — интернет-магазин торгового оборудования (стеллажи, витрины). Это был унаследованный монолит на jQuery. **Команда** была небольшой: я (фронтенд), бэкендер-универсал и дизайнер на аутсорсе.  
Моей **ключевой задачей** была постепенная миграция проекта на современный стек. Я начал с самого проблемного места — системы фильтрации, которая была нечитаемой и медленной. Я переписал её на **React + Redux Toolkit**, что сразу дало прирост в производительности и удобстве поддержки.  
Другие **задачи** включали: интеграцию новых React-компонентов в старую jQuery-основу, внедрение ленивой загрузки изображений для повышения скорости загрузки сайта, доработку UI-кита на Bootstrap и рефакторинг старого кода для поддержания его работоспособности."

**3. Твои гринфлаги, редфлаги и триггеры у работодателя**  
(Будь честен, это ценится)  
**Гринфлаги (что ты ценишь):**

* **Четкие процессы:** Code review, CI/CD, понятный планирование (спринты/итерации).
* **Профессиональная команда:** Возможность перенимать опыт у более senior-разработчиков, конструктивное обсуждение решений.
* **Качество кода:** Понимание руководством, что поддержка качества кода и тестирование — это инвестиция, а не трата времени.
* **Современный (или обновляемый) стек:** Готовность компании развивать технологию, а не застревать в легаси навсегда.

**Редфлаги (чего ты избегаешь):**

* **Постоянный аврал и переработки** как стандартный режим работы.
* **Микроменеджмент:** Отсутствие доверия к специалисту.
* **Полное отсутствие процессов:** "Бросаемся на амбразуру", задачи ставятся в чатике, ничего не фиксируется.
* **Нежелание инвестировать в развитие:** "Мы на jQuery 1.7 и так сойдет, ничего менять не будем".

**Триггеры (что может заставить уйти):**

* Систематическое нарушение work-life balance.
* Полный застой в профессиональном развитии, технологический ступор.
* Токсичная атмосфера в коллективе, постоянный негатив.

**4. Уровень дохода**  
(Заранее посмотри вилки на HH на позицию Middle+/Senior Frontend-разработчик в Москве)  
**Ответ:** "Исходя из моего опыта (4.5+ года), сложности проектов и текущей рыночной ситуации, мои ожидания находятся в вилке **от 200 000 до 280 000 руб. на руки**. Для меня комфортной цифрой была бы **~240 000 руб. на руки**. Я открыт к обсуждению и готов рассмотреть интересное предложение с учетом всего пакета (ДМС, бонусы, отпуск)."

**5. Как вообще пришел в айти, что вдохновило?**  
**Ответ (нейтральный и правдоподобный):** "Мое образование связано с IT. Еще в университете я начал интересоваться веб-разработкой. Меня всегда вдохновляла возможность создавать что-то tangible — то, что можно увидеть, пощупать и чем могут пользоваться тысячи людей. Frontend оказался идеальным сочетанием логики, творчества и немедленного визуального результата от написанного кода. Я начал с простых сайтов, постепенно углубляясь в JavaScript и его экосистему, и понял, что это именно та область, в которой хочу развиваться."

### ****Часть 2: Технические вопросы****

**1. Замыкание в JS**  
**Ответ:** "Замыкание — это способность функции **«запоминать»** свое лексическое окружение (область видимости), в котором она была создана, даже после того, как это окружение перестало существовать. Проще говоря, функция имеет доступ к переменным из внешней функции, в которой она была объявлена. Это фундаментальный механизм JS, который используется повсеместно: для создания приватных переменных, каррирования, работы с событиями и колбэками."

**Пример:**  
function createCounter() { let count = 0; // эта переменная "замыкается" return function() { return ++count; }; } const counter = createCounter(); console.log(counter()); // 1 console.log(counter()); // 2 // Переменная count доступна только внутри counter и является по сути приватной.

**2. Что такое stateless приложение?**  
**Ответ:** "Stateless (бесссостоятельное) приложение — это приложение, которое **не хранит состояние (state) пользователя между запросами**. Каждый запрос от клиента к серверу содержит всю необходимую информацию для его обработки. Классический пример — сервер, отвечающий на REST API запросы. В контексте фронтенда так иногда называют компоненты, которые не имеют внутреннего состояния (например, функциональные компоненты, которые получают все данные через props). Противоположность — Stateful приложение (например, сервер с сессиями)."

**3. Что такое SQL инъекция? Методы защиты, в чем опасность?**  
**Ответ:** "SQL-инъекция — это уязвимость, возникающая, когда злоумышленник может **вмешаться в запрос**, который приложение отправляет к базе данных, подставив в него malicious код.  
**Опасность** колоссальна: кража, изменение или удаление данных, обход аутентификации.  
**Методы защиты:**

1. **Использование подготовленных выражений (Prepared Statements) с параметризованными запросами.** Это главный и самый надежный способ. Параметры передаются отдельно от запроса и не интерпретируются как SQL-код.
2. **Валидация и экранирование** всех пользовательских входных данных (менее надежно, чем prepared statements).
3. **Принцип наименьших привилегий:** Не подключаться к БД от имени root-пользователя. У аккаунта приложения должны быть только необходимые права.
4. **ORM** (как Sequelize из моего резюме) по умолчанию используют подготовленные выражения, что автоматически защищает от инъекций."

**4. Важен ли порядок столбцов в составном первичном ключе?**  
**Ответ:** "**Да, важен.** Составной первичный ключ (a, b) — это не то же самое, что ключ (b, a). Это важно с точки зрения **производительности индексов**. Индекс, построенный по (a, b), будет эффективен для поиска по a и по паре (a, b), но он будет неэффективен для поиска только по b. Если частым запросом будет поиск по b, то потребуется отдельный индекс на b."

**5. Как проанализировать производительность запросов в PostgreSQL?**  
**Ответ:** "Основной инструмент — команда EXPLAIN ANALYZE.

1. EXPLAIN показывает план выполнения запроса, который выбрал планировщик PostgreSQL.
2. EXPLAIN ANALYZE не только показывает план, но и выполняет запрос, показывая реальное время выполнения и другие статистические данные.  
   Я смотрю на:

* **Типы сканирования** (Seq Scan — плохо, Index Scan — хорошо).
* **Стоимость (cost)** операций.
* **Количество возвращаемых строк (rows)**.
* **Наличие операций, убивающих производительность** (например, Sort или Nested Loop при большом количестве строк).  
  Это помогает найти «узкие места» и понять, какие индексы нужно добавить или переписать запрос."

### 4. Приходилось ли писать юнит-тесты? Для чего они были и как Вы их писали?

**Тактика ответа:** Да, конечно приходилось! Это стандартная практика. Даже если писал немного — это уже опыт.

**Ваш ответ (используем легенду):**  
"Да, конечно. В рамках миграции легаси-кода на React и TypeScript мы с командой пришли к выводу о необходимости повысить надежность ключевых модулей. Мы начали писать юнит-тесты для самых важных и сложных частей приложения — **логики работы корзины, утилит для обработки данных с бэкенда и сложных хуков**.  
Мы использовали связку **Jest + React Testing Library**. Подход был такой: мы не стремились покрыть тестами весь старый jQuery-код, а фокусировались на новых, переписанных модулях. Например, я писал тесты для функции-редьюсера, которая управляет состоянием фильтров: проверял, что при dispatch определенного action состояние меняется предсказуемым образом. Это помогло нам избежать регрессий при дальнейшей разработке и рефакторинге."

* **Что мы говорим:** Конкретные технологии (Jest, RTL), конкретные объекты тестирования (редьюсеры, хуки, утилиты), понятную цель (избежать регрессии).
* **Чего не говорим:** "Нет, не писал" или "Писал всё подряд без цели".

### 5. Приходилось ли проводить рефакторинг кода? Каким образом Вы понимали, что это нужно сделать?

**Тактика ответа:** Весь твой опыт — это сплошной рефакторинг! Это твой главный козырь.

**Ваш ответ:**  
"Да, рефакторинг был постоянной и неотъемлемой частью моей работы на обоих проектах. Понимание о необходимости рефакторинга приходило по нескольким признакам:

1. **Проблемы с поддержкой:** Когда добавление новой простой фичи занимало непропорционально много времени из-за сложности и запутанности старого кода.
2. **Появление багов:** Когда в несвязанных частях приложения начинали возникать ошибки после небольших изменений — это явный признак сильной связанности кода (tight coupling).
3. **Низкая производительность:** Например, интерфейс «подтормаживал» при работе с большими списками товаров. Профилирование показывало, что проблема в неоптимальном алгоритме на jQuery.
4. **Плановая миграция:** Самым крупным рефакторингом была плановая миграция с jQuery на React. Мы принимали решение на основе грядущих бизнес-требований: старый стек не позволял эффективно реализовывать сложный интерактивный функционал.

**Мой подход** к рефакторингу всегда был итеративным: не переписать всё сразу, а делать это постепенно, по модулям, обязательно покрывая рефакторенный код тестами, чтобы быть уверенным, что мы ничего не сломали."

* **Что мы говорим:** Конкретные метрики и боли (время разработки, баги, производительность), системный подход.
* **Чего не говорим:** "Мне не нравился код, вот я его и переписал".

Эти ответы представят тебя как вдумчивого, опытного и целеустремленного разработчика, который не просто пишет код, а мыслит категориями качества и развития продукта.

**1. Что для тебя важно в будущей компании, где ты будешь работать?**

**Тактика ответа:** Сгруппируй свои ожидания по блокам: продукт, процессы, команда, развитие. Покажи, что ты не просто ищешь "место работы", а осознанно выбираешь среду для роста.

**Ваш ответ:**

"Для меня важно несколько аспектов, которые в совокупности создают идеальную среду для работы:

1. **Интересный продукт:** Мне хочется работать над сервисом, который реально полезен пользователям и решает сложные задачи. Это мотивирует гораздо сильнее, чем просто "делать таски".
2. **Качественные процессы:** Я ценю там, где есть **code review**, продуманный CI/CD, понятное планирование (спринты, итерации) и тестирование. Это не бюрократия, а инструменты, которые в долгосрочной перспективе экономят время и нервы всей команды.
3. **Сильная команда:** Для меня важно окружение, у которого можно учиться. Я хочу работать с опытными коллегами, с которыми можно советоваться по архитектурным решениям и которые дают конструктивную обратную связь.
4. **Современный стек и подходы:** Меня привлекает возможность работать с актуальными технологиями и практиками (например, FSD-архитектура, о которой я читал). Я не хочу "консервироваться" на устаревших решениях.
5. **Баланс и адекватность:** Понимание, что аврал — это исключение, а не правило. Уважение к личному времени и focus на результате, а не на отсидке часов."

**2. Как ты видишь свое развитие в этой профессии? Какую бы роль хотелось бы занять в команде?**

**Тактика ответа:** Покажи амбиции, но оставайся адекватным. Не заявляй "хочу быть тимлидом через месяц", а покажи путь постепенного роста ответственности и экспертизы.

**Ваш ответ:**

"В краткосрочной перспективе (1-2 года) я хочу углубиться как **эксперт по фронтенд-архитектуре**. Мне интересно разбираться в том, как строить сложные, производительные и масштабируемые приложения: оптимизация бандлов, управление состоянием на уровне приложения, производительность.

Что касается роли в команде, то мне близка роль **ведущего разработчика (Senior Developer) или наставника**. Мне нравится не только самому решать сложные задачи, но и помогать расти другим, делиться знаниями, участвовать в принятии архитектурных решений и быть тем человеком, к которому можно обратиться за советом по сложным вопросам по коду.

В более долгосрочной перспективе я открыт к тому, чтобы взять на себя больше архитектурной ответственности или менторской функции, возможно, в роли **техлида**."

**3. Почему тебя это привлекает? (вопрос, вытекающий из предыдущего)**

**Тактика ответа:** Объясни мотивацию, стоящую за твоими карьерными целями. Покажи, что движет тобой как инженером.

**Ваш ответ:**

"Меня это привлекает, потому что я по натуре **решатель проблем** ('problem solver').

* **Архитектура** — это самый высокий уровень решения проблем. Это не просто "починить баг" или "сверстать кнопку", а создать систему, которая будет надежно работать, легко масштабироваться и в которую будет приятно вносить изменения год и два спустя. Это сложный пазл, решение которого приносит огромное удовлетворение.
* **Менторство и обмен знаниями** — это, по сути, **сила умножения**. Помогая расти другим членам команды, я могу положительно повлиять не на одну задачу, а на качество всего продукта в целом. Видеть, как коллега, которому ты помог, начинает сам решать сложные задачи — это круто. Кроме того, сам процесс объяснения и обсуждения помогает глубже понять subject matter самому.

Мне нравится не просто писать код, а **создавать что-то ценное и долговечное**, и рост в сторону архитектуры и менторства — это прямой путь к этому."

**Итог:** Эти ответы рисуют портрет не просто исполнителя, а мыслящего инженера, который заботится о качестве продукта, команды и своего профессионального роста. Это именно тот тип разработчиков, которых хотят видеть в сильных компаниях.

**1. Как в команде договариваетесь, какую библиотеку использовать?**

**Ответ:** Это процесс принятия взвешенного решения, а не голосование.

1. **Выявление потребности:** Сначала четко формулируем проблему, которую нужно решить.
2. **Поиск вариантов:** Ищем несколько подходящих библиотек (на GitHub, [npmtrends.com](https://npmtrends.com/)).
3. **Критерии оценки:**
   * **Размер бандла:** Как повлияет на скорость загрузки?
   * **Популярность и сообщество:** Много ли звезд на GitHub, часто ли обновляется, есть ли ответы на Stack Overflow?
   * **Соответствие задаче:** Насколько хорошо библиотека решает именно нашу проблему?
   * **Простота интеграции и API:** Будет ли легко использовать и поддерживать?
   * **Типизация:** Есть ли поддержка TypeScript?
4. **Принятие решения:** Разработчик, отвечающий за задачу, готовит краткий обзор по критериям и предлагает вариант. Решение принимается после обсуждения на команде (тимлид, техлид, архитектор). Важно зафиксировать выбор в документации.

**2. Что такое семантические теги?**

**Ответ:** Это HTML-теги, которые **несут смысловое значение** (семантику), а не просто задают внешний вид.

* **Примеры:** <header>, <footer>, <nav>, <article>, <section>, <aside>, <main>.
* **Зачем нужны:**
  1. **Доступность (a11y):** Скринридеры используют их для навигации.
  2. **SEO:** Поисковые системы лучше понимают структуру и содержание страницы.
  3. **Читаемость кода:** По тегу сразу понятно, какую роль играет этот блок.

**3. Что такое специфичность в CSS?**

**Ответ:** Это **алгоритм**, который браузер использует для определения того, какое CSS-правило применить к элементу, если на него претендует несколько правил. Это "вес" селектора.

**Порядок приоритета (от низкого к высокому):**

1. Селекторы тегов (p) и псевдоэлементов (::before) -> **(0,0,1)**
2. Селекторы классов (.class), атрибутов ([type="text"]) и псевдоклассов (:hover) -> **(0,1,0)**
3. Селекторы ID (#id) -> **(1,0,0)**
4. **Inline-стили** (атрибут style) -> **(1,0,0,0)**
5. Ключевое слово !important — ломает всю специфичность и имеет наивысший приоритет (использовать только в крайних случаях).

**Как считать:** Представь себе число (a, b, c), где:

* a — количество ID,
* b — количество классов,
* c — количество тегов.  
  Селектор #header .menu li.item имеет специфичность (1,2,1).

**4. Что такое ссылочный тип данных?**

**Ответ:** В JavaScript это тип данных, значение которого **не хранится непосредственно в переменной**, а хранится **по ссылке** в памяти. Переменная содержит лишь адрес (ссылку) на это место в памяти.

* **Какие типы являются ссылочными?** **Объекты** (включая **массивы** и **функции**).
* **Особенность:** При присваивании одной переменной другой копируется **ссылка**, а не само значение. Поэтому изменение данных по одной переменной повлияет на другую.

javascript

let a = { name: 'John' };

let b = a; *// скопировалась ссылка, а не объект*

b.name = 'Pete';

console.log(a.name); *// 'Pete' - изменение затронуло оба "объекта"*

**5. Для чего нужен TypeScript?**

**Ответ:** TypeScript — это **статически типизированное надмножество JavaScript**, которое компилируется в обычный JS.  
**Основные цели:**

1. **Обнаружение ошибок на этапе разработки.** TS найдет опечатки в свойствах, несоответствие типов и многие другие ошибки еще до запуска кода.
2. **Улучшение читаемости и самодокументирования кода.** Типы служат документацией и делают ожидания от функций и компонентов явными.
3. **Улучшение возможностей IDE** (автодополнение, навигация по коду, рефакторинг).

**6. Что такое Omit?**

**Ответ:** Omit<Type, Keys> — это **утилита типа в TypeScript**, которая создает новый тип путем исключения указанных свойств Keys (строковый литерал или их union) из типа Type.

* **Пример:**

typescript

interface User {

name: string;

age: number;

email: string;

password: string;

}

type UserWithoutPassword = Omit<User, 'password'>;

*// { name: string; age: number; email: string; }*

* **Антоним:** Pick<Type, Keys> — выбирает только указанные свойства.

**7. Разница между интерфейсом и типом?**

| Критерий | interface | type (Type Alias) |
| --- | --- | --- |
| **Объявление** | interface User { ... } | type User = { ... } |
| **Расширение** | Через extends: interface Admin extends User {} | Через & (intersection): type Admin = User & { ... } |
| **Объединение** | Через extends нескольких интерфейсов | Через ` | (union):<br>type Id = number | string` |
| **Декларативное слияние** | **Можно** объявить интерфейс с одним именем дважды, и они сольются. | **Нельзя** — ошибка повторного объявления. |  |  |
| **Использование** | В основном для описания **объектов** и **классов**. | Может описывать **что угодно**: объекты, примитивы, union-типы, кортежи. |  |  |

**Общее правило:** Используй interface для описания форм объектов, особенно если они будут публичным API (например, библиотеки компонентов). Используй type для более сложных конструкций (union, кортежи) или когда нужно дать имя примитиву.

**8. Разница между useMemo и useCallback?**

* useMemo — **мемоизирует результат вычислений (значение).** Кэширует результат тяжелой функции и возвращает его, пока зависимости не изменились.

javascript

const expensiveValue = useMemo(() => computeExpensiveValue(a, b), [a, b]);

* useCallback — **мемоизирует саму функцию.** Возвращает ту же самую функцию, пока зависимости не изменились. Это нужно, чтобы не ломать мемоизацию дочерних компонентов (например, обернутых в React.memo), куда эта функция передается пропсом.

javascript

const memoizedCallback = useCallback(() => { doSomething(a, b); }, [a, b]);

**Проще говоря:** useMemo запоминает *что вернула функция*, useCallback запоминает *саму функцию*.

**9. HOC?**

**Ответ:** **Higher-Order Component (Компонент высшего порядка)** — это **функция**, которая принимает компонент и возвращает новый, обернутый компонент с дополнительной логикой или пропсами.

* **Аналогия из функционального программирования:** HOC = connect(mapStateToProps)(MyComponent) из React-Redux.
* **Зачем нужны?** Для **переиспользования логики** (например, логика подписки на данные, проверка прав доступа, инжекция пропсов).
* **Сейчас используются реже,** так как во многих случаях их заменяют **кастомные хуки.**

**10. Для чего нужен Redux?**

**Ответ:** Redux — это **менеджер состояния** для JavaScript-приложений. Он предоставляет **глобальный store (хранилище)** для состояния, которое должно быть доступно из разных частей приложения.  
**Основные причины использования:**

1. **Управление сложным, глобальным состоянием** (данные пользователя, корзина, уведомления).
2. **Предиктивная работа с состоянием:** Состояние изменяется только через **actions** и **reducers**, что делает изменения предсказуемыми и легче отлаживаемыми.
3. **Централизация состояния:** Правда о состоянии приложения живет в одном месте.
4. **Инструменты разработчика (DevTools)** для отладки ("машина времени" по действиям).

**Минус:** Добавляет много шаблонного кода (boilerplate), поэтому сейчас часто используют **Redux Toolkit (RTK)** — официальный упрощенный подход к написанию Redux-логики.

**11. Какие еще знаешь стейт менеджеры и в чем отличия?**

1. **Context API (+ useReducer):**
   * **Встроен в React.** Прост в использовании для простых случаев.
   * **Не оптимизирован для частых обновлений.** Перерисовывает всех потребителей контекста при изменении значения, даже если они используют неизменившуюся часть состояния.
2. **MobX:**
   * **Использует реактивное программирование.** Ты помечаешь данные как observable, а компоненты как observer, и они автоматически обновляются при изменении данных.
   * **Меньше boilerplate-кода**, более "магический" и императивный по сравнению с Redux.
3. **Zustand:**
   * **Очень легкий и простой.** Минимум кода для создания store.
   * Не требует оборачивать приложение в провайдеры, как Context или Redux.
   * Позволяет напрямую изменять состояние (как в MobX), но без "магии".
4. **Recoil (от Meta):**
   * Решает проблемы производительности Context API.
   * Работает с концепцией **атомов** (мелкие части состояния) и **селекторов** (производные состояния).

**Главное отличие Redux от остальных:** Строгая архитектура и иммутабельные обновления (состояние не меняется, а создается заново).

**12. На твой взгляд, React — это библиотека или фреймворк?**

**Ответ:** **Библиотека.**

* **Библиотека** предоставляет набор готовых функций и компонентов, которые ты **вызываешь сам**, чтобы строить приложение так, как хочешь. React отвечает только за рендеринг UI и управление его состоянием. Маршрутизацию, HTTP-запросы, управление состоянием всего приложения ты выбираешь сам (React Router, Axios, Redux).
* **Фреймворк** (как Angular) — это уже **готовая архитектура**, которая диктует, как должно быть организовано приложение, и предоставляет все необходимое "из коробки".

React часто называют "библиотекой для построения пользовательских интерфейсов".

**13. HTTP методы?**

**Ответ:** Это действия, которые указывают, какую операцию мы хотим выполнить с ресурсом на сервере.  
Основные методы (глаголы) REST:

* **GET** — запросить данные (получить ресурс). Без побочных эффектов.
* **POST** — создать новый ресурс.
* **PUT** — полностью обновить (заменить) существующий ресурс.
* **PATCH** — частично обновить существующий ресурс.
* **DELETE** — удалить ресурс.
* **HEAD** — запросить только заголовки ответа (без тела).
* **OPTIONS** — запросить допустимые методы для ресурса (используется для CORS).

**14. Разница между PUT и PATCH?**

* **PUT** — **идемпотентный** метод. Он полностью **заменяет** весь ресурс на тот, который передан в теле запроса. Если какого-то поля нет в теле запроса, на сервере оно должно быть установлено в null или значение по умолчанию. "Положи этот ресурс вот с такими данными по этому URL."
* **PATCH** — **неидемпотентный** (чаще всего). Он применяет **частичные модификации** к ресурсу. В теле запроса передаются только те поля, которые нужно изменить. "Внеси точечные изменения в ресурс по этому URL согласно инструкции в теле."

**Аналогия с документом:**

* **PUT** — это отправить весь документ заново.
* **PATCH** — это отправить инструкцию "замени в первом абзаце слово 'хорошо' на 'отлично'".

Что ответить на вопрос рекрутеа: "Как проходили код ревью?"

Отличный вопрос! Ответ на него показывает вашу зрелость как разработчика и опыт работы в команде. Рекрутер хочет понять, насколько вы знакомы с современными процессами разработки, умеете ли вы принимать обратную связь и работать коллаборативно.

Вот структурированный план, как можно построить ответ, с вариантами для разного уровня опыта.

**Структура идеального ответа (по пунктам):**

1. **Начать с контекста:** Кратко описать процесс в команде/компании.
2. **Описать инструменты и механику:** Где и как это происходило.
3. **Привести конкретный пример:** Кратко рассказать кейс из своей практики (что проверяли, какие были комментарии, как вы действовали).
4. **Сделать акцент на своей роли:** Показать себя как ответственного и проактивного участника процесса (и как автора, и как ревьюера).
5. **Завершить выводом о пользе:** Обозначить ваше понимание ценности процесса.

**Пример ответа для Middle/Senior разработчика (универсальный и сильный):**

"В моих предыдущих командах код-ревью было обязательным этапом перед любым мержем в основную ветку. Это был неформальный, но очень важный процесс обеспечения качества.

**Процесс был таким:** я завершал задачу, пушил код в feature-ветку и создавал Pull Request в GitHub/GitLab. К PR я всегда старался прикладывать понятное описание: что сделано, как тестировать. Система автоматически запускала CI/CD (прогоняла тесты, линтеры). После этого ревью запрашивалось у одного или двух коллег.

**Что проверяли:** смотрели не только на очевидные ошибки, но и на архитектурные решения, читаемость кода, покрытие тестами и соответствие нашему гайдлайну. Мы использовали ESLint и Prettier для автоматической проверки стиля, так что ревьюеры могли сконцентрироваться на логике.

**Например,** как-то раз я реализовывал сложный фильтр для каталога. В ревью мне указали на потенциальную уязвимость — я не экранировал один из параметров, который мог прийти с фронтенда, что могло привести к SQL-инъекции. Я поблагодарил коллегу, исправил код, и мы вместе убедились, что проблема решена. Это здорово меня выручило.

**Я участвовал в процессе и как автор, и как ревьюер:** я всегда старался оперативно отвечать на комментарии, а когда проверял чужой код, делал это максимально конструктивно, не просто указывая на ошибку, а предлагая варианты решения или задавая наводящие вопросы.

**В итоге,** я считаю код-ревью бесценной практикой. Это не только про поиск багов, но и про обмен знаниями внутри команды и поддержание единого стиля кода."

**Ключевые моменты, которые стоит упомянуть (выберите те, что вам ближе):**

* **Инструменты:** GitHub Pull Requests, GitLab Merge Requests, Phabricator, Crucible.
* **Автоматизация:** Упоминание CI (например, GitHub Actions, Jenkins), линтеров (ESLint), форматтеров (Prettier). Это большой плюс.
* **Культура:** "Мы стремились к конструктивному диалогу", "ревью не для поиска виноватых, а для улучшения кода", "комментарии всегда были вежливыми и по делу".
* **Ваша проактивность:**
  + *Как автор:* "Всегда описывал изменения в PR", "исправлял замечания оперативно", "отвечал на вопросы".
  + *Как ревьюер:* "Старался проверять PR коллег быстро", "задавал уточняющие вопросы", "предлагал альтернативные решения".
* **Пример:** Приведите короткий, но конкретный пример. Это делает ответ живым и правдоподобным.

**Что говорить, если опыта мало или он неформальный?**

Не страшно! Можно честно сказать об этом, но показать понимание процесса.

"В моем предыдущем проекте не было строгого обязательного код-ревью, но мы практиковали **парное программирование** на сложных задачах, что, по сути, является непрерывным ревью. Кроме того, мы часто добровольно показывали друг другу код по сложным модулям, чтобы получить фидбэк. Я сам всегда интересовался лучшими практиками, смотрел на код коллег и изучал гайдлайны (например, по React или Python). Я понимаю важность этого процесса для качества продукта и очень хотел бы работать в команде, где он хорошо налажен."

**Чего стоит избегать:**

* **Негатива:** Не говорите "Ревью было ужасно, меня постоянно критиковали" или "Ревьюер всегда придирался к мелочам". Это красный флаг для рекрутера.
* **Общих фраз:** Избегайте фраз "Всё было хорошо" без деталей.
* **Лжи:** Не придумывайте сложных процессов, если их не было. Рекрутер или техинтервьюер легко это проверит.

Главное — показать, что вы **командный игрок**, который понимает **ценность качества кода** и умеет **конструктивно коммуницировать**.

**Опиши свой самый большой факап в работе. Какой вывод был сделан?**

На одном из проектов, связанном с миграцией части старого jQuery-кода на React + TypeScript, я столкнулся с ситуацией, когда из-за спешки и желания быстрее показать результат не до конца проработал архитектуру взаимодействия между legacy-компонентами и новыми.

В результате после деплоя на прод обнаружилась проблема: при определённых условиях фильтры товаров в каталоге переставали работать, если пользователь переходил из старой части интерфейса в новую и обратно. Это приводило к некорректному отображению товаров и путанице.

Ситуация была неприятная, но мы быстро откатили изменения, чтобы восстановить работоспособность. После этого я провёл детальный разбор проблемы, изолировал причину — оказалось, что не был учтён сценарий совместного управления состоянием между старой и новой системами.

**Выводы, которые я сделал:**

1. Всегда тестировать интеграцию новых модулей с legacy-кодом в максимально приближенных к боевым условиям, включая пользовательские сценарии.
2. Не пренебрегать написанием хотя бы минимальных интеграционных тестов для критически важных функциональностей.
3. Заранее проектировать чёткий контракт взаимодействия между старой и новой системами, даже если это замедляет разработку на начальном этапе.

С тех пор я стал гораздо внимательнее относиться к постепенному и безопасному внедрению изменений в существующую кодовую базу, особенно когда речь идёт о гибридных решениях.

Этот ответ показывает, что ты умеешь признавать ошибки, анализировать их и главное — извлекать полезные уроки, которые применяешь в дальнейшей работе.

Добрый день, меня зовут Денис Попов, я frontend-разработчик с 4-летним опытом создания адаптивных и производительных интерфейсов для e-commerce проектов. Изучив вакнсию, я уверен, что мой опыт работы с современным стеком технологий (React, TypeScript, Redux), а также умение эффективно модернизировать legacy-код могут быть полезными вашему продукту.

Да, конечно. Весь мой опыт коммерческой разработки (и в «Детали Торговли», и в «21Компани») был построен в рамках SCRUM.

Мы работали в итерациях (спринтах), длительностью обычно 2 недели. Стандартный процесс включал:

* **Планирование спринта:** где мы вместе с командой (Product Owner, разработчики, QA) оценивали задачи из бэклога продукта и формировали план на следующий спринт.
* **Ежедневные стендапы (daily stand-up):** где каждый рассказывал, что сделал вчера, что планирует сделать сегодня и с какими препятствиями столкнулся.
* **Ретроспективу спринта:** после завершения каждого спринта мы анализировали, что прошло хорошо, а что можно улучшить в наших процессах. Это позволяло постоянно адаптироваться и становиться эффективнее.
* **Обзор спринта (demo):** где мы показывали готовый функционал продукт-оунеру и стейкхолдерам.

На практике это означало, что я постоянно работал с таск-трекером (чаще всего Jira), брал задачи в спринт, оценивал их по стори поинтам и был ответственным за их завершение в срок. Такой подход очень дисциплинирует, учит разбивать большие задачи на мелкие подзадачи и эффективно коммуницировать с командой о прогрессе и возможных рисках.

Мне нравится структурированность и прозрачность, которые даёт SCRUM, а также возможность быстро получать обратную связь и гибко менять приоритеты.

**Легенда для «21Компани» (Апрель 2021 — Февраль 2023)**

**Суть проекта:** Крупный интернет-магазин одежды (аналог Lamoda или Brandshop). Очень высокая посещаемость, поэтому важна каждое миллисекунда скорости.

**Ваша роль:** Вы начинали как разработчик, поддерживающий старый код, и стали двигателем его постепенной модернизации.

**Ключевые детали для рассказа:**

* **Проект:** "Высоконагруженный проект с большой кодовой базой на jQuery и Bootstrap 3. С течением времени код стал очень сложным для поддержки."
* **Ваша инициатива:** "Я лично проявил инициативу по внедрению TypeScript и React. Начал с небольших, изолированных компонентов, чтобы доказать их эффективность (например, модальное окно выбора размера на странице товара). После успеха мне дали добро на более крупные модули."
* **Миграция:** "Ключевым достижением была постепенная миграция корзины покупок и системы фильтров с jQuery на TypeScript. Это позволило почти полностью исключить ошибки, связанные с типами данных, и упростить разработку новых фич."
* **Оптимизация:** "Скорость загрузки была критически важна. Я внедрил ленивую загрузку изображений (LazySizes), что dramatically сократило время первоначальной отрисовки страниц. Также поработал над оптимизацией Redux-стора, чтобы избежать лишних ре-рендеров при обновлении большого количества товаров."
* **Легаси:** "jQuery 1.11 — это очень старая версия. Приходилось писать код, который бы не конфликтовал с ней, и быть очень осторожным с плагинами."

**Пример ответа на вопрос "Каким был самый сложный проект и почему?":**  
"Наиболее сложным был интернет-магазин [21-shop.ru](https://21-shop.ru/). Это был большой легаси-проект на jQuery с огромным количеством кода. Сложность заключалась в том, чтобы вносить изменения, не ломая существующую функциональность, и при этом постепенно modernizing стек. Я начал с внедрения TypeScript для типизации самых 'больных' мест, а затем, шаг за шагом, перевел ключевые модули, такие как корзина и фильтры, на React. Это был длительный процесс, но он дал огромный выигрыш в поддерживаемости и надежности кода."

**Что подчеркнуть на собеседовании:**

1. **Опыт работы с легаси:** Это ваш козырь. Все умеют писать с нуля на современном стеке, но немногие могут аккуратно и эффективно работать со старым кодом.
2. **Постепенная миграция:** Вы не сносили все подряд, а действовали поэтапно, что ценится в бизнесе (меньше рисков, можно быстро получить результат).
3. **Бизнес-ориентированность:** Ваши решения (ленивая загрузка, оптимизация) были направлены на улучшение реальных метрик (скорость загрузки, конверсия в покупки).
4. **Инициативность:** Вы не просто выполняли задачи, а предлагали решения (внедрить TS, React).

**Слабое место в резюме:** Уровень английского A1. Будьте готовы к вопросу. Ответ может быть таким: "Сейчас мой уровень — базовый, я могу читать техническую документацию со словарем. Но я понимаю его важность для карьеры в разработке и активно занимаюсь его повышением с помощью курсов/приложений."

Удачи! Ваше резюме выглядит как у сильного middle-разработчика с нужным опытом. Осталось так же уверенно его подать.

Отличная вакансия! Компания выглядит адекватной, а проект — интересным. Ваш опыт идеально подходит под их требования. Давайте подготовимся.

Исходя из вакансии, они ищут **Fullstack-разработчика с уклоном во фронтенд (React)**, но с обязательным знанием бэкенда (.NET или Node.js). Упоминание Next.js говорит о том, что они, скорее всего, работают с SSR (Server-Side Rendering) или хотят начать.

Вот какие вопросы вам почти гарантированно зададут, сгруппированные по темам, и как на них лучше отвечать, **опираясь на ваш опыт**.

### 1. Проекты и опыт (Самый важный блок)

Здесь будут оценивать, насколько ваш опыт релевантен.

**Вопросы:**

1. **«Расскажите о самом сложном/complex проекте, который вы разрабатывали (особенно в контексте интернет-магазинов)».**
   * **Что отвечать:** Используйте метод STAR. Пример: «На проекте [*Standes.ru*](https://standes.ru/) (S) мне нужно было провести миграцию с jQuery на современный стек (T). Я разработал план, внедрял React+TypeScript компоненты постепенно, обеспечивая обратную совместимость (A). В результате мы получили более поддерживаемую codebase, ускорили разработку новых фич и улучшили UX за счет более отзывчивого интерфейса (R)».
2. **«Приходилось ли вам участвовать в обсуждении требований или планировании?»** (Это прямо в вакансии)
   * **Что отвечать:** «Да, активно участвовал. Например, когда нужно было реализовать новую систему фильтрации, я предлагал технические решения, оценивал сложность, советовал, как можно упростить реализацию без потери качества для пользователя. Всегда стараюсь понять бизнес-задачу за требованием».
3. **«Был ли у вас опыт работы в команде? Как вы выстраивали процесс?»**
   * **Что отвечать:** «Да, работал в команде с бэкенд-разработчиками и дизайнерами. Мы использовали GitFlow, код-ревью через Merge Request, CI/CD для запуска тестов. Я всегда стараюсь писать чистый и понятный код, чтобы его легко могли поддерживать коллеги».

### 2. Frontend (React / Next.js)

**Вопросы:**

1. **«Расскажите о вашем опыте с React. Какие хуки используете чаще всего?»**
   * **Что отвечать:** *useState*, *useEffect*, *useCallback*, *useMemo*. Обязательно упомяните, что используете *useCallback* и *useMemo* для оптимизации и предотвращения лишних рендеров. Упомяните **Redux Toolkit** для управления состоянием.
2. **«Что такое Virtual DOM и как работает реконсиляция (reconciliation) в React?»** (На Middle+ спросят)
   * **Что отвечать:** «Virtual DOM — это легковесное представление реального DOM в памяти. Когда состояние компонента меняется, React создает новый VDOM, сравнивает его со старым (это и есть reconciliation) и вычисляет минимальный набор изменений для применения к реальному DOM. Это делает обновления эффективными».
3. **«Работали с Next.js? Зачем он нужен?»** (Это будет ключевой вопрос)
   * **Что отвечать (если прямого опыта нет):** «Прямого коммерческого опыта с Next.js пока нет, но я очень внимательно его изучаю и понимаю его ключевые преимущества: Server-Side Rendering (SSR) для SEO и быстрой первоначальной загрузки, статическая генерация (SSG), упрощенная маршрутизация и встроенный API. Я уверен, что смогу быстро в него погрузиться, учитывая мой опыт с React и Node.js». (Это честный и сильный ответ, показывающий интерес).
4. **«Как вы решаете проблему повторного использования кода?»**
   * **Что отвечать:** «Через создание переиспользуемых React-компонентов, кастомных хуков, утилитарных функций. Слежу за принципами DRY (Don't Repeat Yourself). Сейчас интересуюсь архитектурой FSD для более правильной организации кода».

### 3. Backend (Node.js / .NET / Базы данных)

**Вопросы:**

1. **«Какой у вас опыт с Node.js? Что вы на нем делали?»** (Важно!)
   * **Что отвечать:** «Я использовал Node.js в связке с Express.js и Sequelize для создания серверной логики учебных и pet-проектов: настройка REST API, работа с аутентификацией (JWT), взаимодействие с PostgreSQL. Понимаю основы асинхронности, Event Loop. Готов углублять эти знания». (Это показывает инициативу).
2. **«Что такое REST API?»**
   * **Что отвечать:** «Архитектурный стиль для построения веб-сервисов. Использует HTTP-методы (GET, POST, PUT, DELETE) для операций с ресурсами, которые представлены в виде URL-адресов. Данные обычно передаются в формате JSON».
3. **«Опыт работы с SQL и ORM?»**
   * **Что отвечать:** «Да, работал с PostgreSQL через ORM Sequelize. Создавал модели, прописывал связи между ними, выполнял запросы. Понимаю важность индексов для производительности. Для сложных запросов использовал raw SQL».
4. **«Что такое JWT и как работает аутентификация с его помощью?»**
   * **Что отвечать:** «JSON Web Token — это стандарт для безопасной передачи данных между сторонами в виде JSON-объекта. После успешного логина сервер генерирует access и refresh токены (подписанные секретным ключом) и отдает их клиенту. Клиент затем прикладывает access-токен в заголовке запроса к защищенным API. Сервер проверяет подпись токена и его валидность».

### 4. Инструменты и процессы

**Вопросы:**

1. **«Расскажите о вашем workflow с Git».**
   * **Что отвечать:** «Я использую функциональные ветки. Создаю ветку от *main*/*develop*, делаю коммиты, пушаю, создаю Pull/Merge Request, прохожу код-ревью, после апрува мерджу. Использую *rebase* для поддержания чистой истории. Коммиты стараюсь делать атомарными».
2. **«Что такое CI/CD? Сталкивались?»**
   * **Что отвечать:** «Да, на проектах был настроен CI-пайплайн в GitLab CI. При пуше в MR автоматически запускались линтеры (ESLint), проверка типов TypeScript и unit-тесты. Это помогало отлавливать ошибки до мержа в основную ветку».
3. **«Писали ли вы тесты?»**
   * **Что отвечать:** «Да, писал unit-тесты с использованием Jest. Понимаю их важность для надежности кода, особенно при рефакторинге».

### Вопросы, которые ВЫ должны задать им

Это покажет вашу заинтересованность и зрелость.

1. **«На каком стеке (конкретные технологии и версии) построены текущие проекты?»** (Уточните про бэкенд: это .NET или Node.js?)
2. **«Можете подробнее описать типичные задачи на первые пару месяцев?» Как проходит онбординг(адаптация).**
3. **«Как устроен процесс разработки в команде? (Спринты, ежедневные стендапы, планирование, ретроспективы)»**
4. **«Какой командой предстоит работать? (количество человек, роли, есть ли тимлид/проджект-менеджер)»**
5. **«Какие есть возможности для профессионального роста и обучения в компании?»** (Это очень важно для них, судя по вакансии).

**Итог:** Ваш главный козырь — это огромный практический опыт работы с **легаси-кодом, миграцией и поддержкой больших коммерческих проектов**. Это бесценно. Делайте на этом акцент. Говорите, что вам интересен **Next.js** и вы готовы развиваться в сторону **полного стека**.

У вас очень сильное резюме для этой вакансии. Удачи! Вы все сможете.

1. **Технологический стек и интересные задачи.** Мне важно, чтобы работа позволяла применять и углублять современные практики (например, **Next.js, углубленный TypeScript, бэкенд**). Меня мотивируют сложные вызовы, а не рутина по поддержке устаревшего кода. Очень привлекательна возможность участия в **полном цикле разработки** — от обсуждения архитектуры до запуска.
2. **Команда и культура.** Я хочу работать в сильной команде с опытными коллегами, у которых можно учиться. Для меня важна адекватная коммуникация, code review и возможность влиять на технологический выбор.
3. **Продукт и его ценность.** Мне интересно работать над продуктом, который решает реальные проблемы пользователей и имеет понятную ценность на рынке. Ваше описание задач по автоматизации бизнеса и производств звучит именно так.
4. **Условия для роста.** Это включает в себя не только финансовую составляющую, но и возможность профессионального развития: конференции, курсы, карьерный трек внутри компании, mentorship.