Q1. Расскажите, чем, на ваш взгляд, отличается хорошее клиентское приложение от плохого с точки зрения...

1. С точки зрения пользователя хорошему клиентскому приложению свойственны:
   - *Отказоустойчивость.* В случае возникновения ошибки, данные, которые пользователь ввел/добавил, должны быть сохранены. Пользователю нужно показать возможные несложные действия по исправлению ошибки, вместо страшного ярко красного блокирующего окна с сообщением об ошибке.

   - *Необходимый функционал.* Если приложение не обладает функционалом, необходимым пользователю, то тут уже не поможет ни красивый дизайн, ни бренд, ни прочие приятные, но не критически важные бонусы. Хорошее приложение учитывает пользовательские потребности – это в том числе касается содержания Feature Request площадки, где пользователи могу описать важные для их работы «фичи» и привлечь к ним внимание разработчиков.

   - *Высокая скорость загрузки приложения.* Доступ к самому ценному контенту должен быть предоставлен как можно раньше, с последующей дозагрузкой требующейся информации (тут можно использовать такие техники как *lazy loading images, code splitting по маршрутам, Server side rendering, оптимизацию ресурсов и т.д.*)

   - *Доступность*. Есть разные категории пользователей, некоторые обладают нарушением зрения, некоторые – физическими ограничениями. Хорошее приложение должно учитывать эти категории пользователей. WAI-ARIA и, собственно, aria-атрибуты отлично в этом помогают.

   - *Адаптивность*. Приложение должно учитывать различия между 12 дюймовым ноутбуком с возможным зумом по умолчанию, 21 дюймовым настольным дисплеем, и старой Nokia Lumia с экраном в ~320 пикселей.

   - *Оптимистичный UI.* Если приложение демонстрирует пользователю результат операции, до завершения её реального выполнения, то возникает ощущение высокой скорости работ, свойственной native

приложениям. Это однозначно улучшает опыт использования пользователя - вероятность того, что пользователь продолжит работать с таким приложением выше, как мне кажется, чем в случае с приложением, которое постоянно блокирует возможность взаимодействия с ним, чтобы подгрузить или отправить что-либо на сервер.

2. С точки зрения менеджера проекта:

   - *Соблюдение сроков*. Меньше стресса для членов команды, не возникают авралы, которые сложно разрешаются. В таких условиях приложение можно эффективно развивать.

   - *Высокий уровень коммуникации в команде*. Позволяет оперативно (и качественно) разбираться с возникшими проблемами, поскольку все члены команды предлагают своё решение. Не происходит параллельной работы двух членов команды над одной задачей без их координации (т.е. исключены ситуации, когда два разработчика разрабатываю каждый свою «ветку», не зная про другого).

   К тому же, возрастает заинтересованность в рабочем процессе всех участников команды, поскольку они понимают, что именно происходит как на проекте в целом, так и в отдельных командах.

   - *Использование методологии разработки.* Увеличивает продуктивность работы над проектом (например, Agile). А все, что позволяет оптимизировать рабочие процессы, увеличивает количество времени доступное на развитие существующего продукта. К тому же, методологии позволяют делать это [развивать] системно.

3. С точки зрения дизайнера:

   - *Наличие дизайн системы*. Имеется набор цветов, шрифтов, типовых компонентов, которые представляют проект. В итоге, дизайн новых составляющих интерфейса производится значительно быстрее, поскольку можно комбинировать уже существующие компоненты. В конце концов, желание гармонии никто не отменял 😉

- *Точность реализации дизайн-макета за счет взаимодействия с верстальщиком.* Порой случается, что некоторые отступы, цвета и т.д. несколько отличаются от уже имеющихся. В хорошей команде верстальщик начнет обсуждение с дизайнером, чтобы выяснить были ли небольшие расхождения в значениях умышленны (и стоит ли вообще вводить новые компоненты с 2 маленькими отличиями) или это ошибки при генерации макета, опечатки, т.д.
Также это увеличивает соответствие приложения заранее определенным требованиям.

4. С точки зрения верстальщика:

   - *Грамотная CSS архитектура*. Использование модулярного подхода к разработке стилей значительно упрощает затраты на поддержку и развитие существующих элементов интерфейса.

   - *Семантическое использование элементов.* Семантика, которая обычно важна для машин, не менее важна и для человека. Код семантически-грамотно размеченного интерфейса значительно легче читать за счет разделения разных сущностей в коде, чем «полный диватоз» (термин Вадима Макеева, означает неоправданное употребление обилия элементов `<div>` в интерфейсе,).

   - *Доступность*. Позволяет предоставить возможность использования приложения людям, с ограниченными возможностями.

   Также, доступность можно рассматривать, как возможность использования интерфейса в несколько нестандартных условиях: например, чтение ночью на новостном сайте – желательно иметь «ночную тему»; использование приложения на мобильном устройстве – размеры интерактивных элементов должны быть достаточно большими, чтобы было удобно нажимать их, т.д.

   - *Использование инструментов для оптимизации приложения.* Множество задач может быть автоматизировано, позволяя тем самым улучшить жизнь верстальщика и, как следствие, обеспечивать более высокое качество кода. К тому же это гарантирует, что важные стадии (сжатие CSS/JS кода, оптимизация ресурсов, вроде PNG или SVG),

которые вручную выполнять тяжело, не будут пропущены.

5. С точки зрения серверного программиста:

   - *Оптимальная структура БД).* В случае реляционной БД, имеется в виду что отношения между таблицами должны быть нормализованы.

   Это увеличивает скорость работы базы данных (за счет индексации данных), а также уменьшает её объем (за счет устранения дублирования) и надежность информации (т.е. вероятность того, что при обновлении записи где-то в других таблицах останутся не актуальные данные, которые могут привести к ошибке, значительно ниже).

   - *Регулярные бекапы данных.* В случае какого-либо сбоя приложения данные могут быть восстановлены из бекапа. Также это может быть ценно с точки зрения клиентской части, поскольку если там возникла ошибка и пользователь столкнулся с аномальным изменением данных, он может обратиться в саппорт и попросить восстановить утерянную или измененную информацию.
   Примечание: Этот пункт в большей степени относится к DevOps специалисту, нежели к серверному разработчику, оставил для полноты.

   - *Использование инструментов для оптимизации процессов разработки.* Достаточно обширный пункт, сюда я отношу инструменты, которые повышают итоговое качество/скорость выполненных разработчиком работ.
   Например, использование систем версионирования позволяет гибче реагировать на изменения в требованиях к приложению, проще переключаться между разными контекстами (например, быстро перейти с разработки новой «фичи», на исправление критического бага, который ухудшает или блокирует пользовательское взаимодействие с приложением).

   - *Использование инструментов для оптимизации деплоя.*
   Инструменты вроде *Docker* позволяют быстро развёртывать проект на удаленных серверах, нивелируя необходимость рутинной настройки. Плюсом еще и можно считать изоляцию разрабатываемого

приложения от внешнего окружения. Сюда же можно отнести *CI* системы, которые могут по коммиту в репозиторий приложения проводить сборку проекта и отправлять на прод. сервер (смело) или на sandbox-сервер для ручного тестирования обновления.

Многие из приведенных пунктов могут перемешаться между направлениями, хотя бы потому, что хорошие практики, так или иначе схожи между разными специализациями в разработке (в общем случае, не только в разработке).

Так же, я рассматривал пункты, которые прямо и *косвенно* влияют на итоговое качество продукта, а также скорость его разработки.

Q2. Опишите основные особенности разработки крупных многостраничных сайтов, функциональность которых может меняться в процессе реализации и поддержки.

К особенностям я бы отнес:

1. Необходимость учитывать композиционный характер компонентов интерфейса в архитектуре CSS стилей:

   Множество составляющих интерфейса имеют общие составляющие (цвета, геометрию, т.д.), а так как это большое количество страниц, то элементы могут повторяться на разных страницах.

   Вместо дублирования этих частей рекомендуется использовать известную в сообществе методологию, например, BEM или SMACSS, чтобы добиться логического разделения разных компонентов интерфейса, уменьшить дублирование кода и уровень сложности разработки (гораздо проще писать код, когда не приходится анализировать особенности специфичности сотни селекторов на проекте).

2. Загрузка ресурсов:

   На многостраничном сайте между страницами может иметься различие в составляющих интерфейса (например, на одной странице есть блок с регистрацией, на другой – нет). А учитывая, что

сайт крупный, то некоторые страницы могут гораздо больше отличаться, чем вообще быть похожими.

Вместо загрузки всего CSS/JS кода при первой загрузки страницы, который может не потребоваться пользователю, если он не посетит другие страницы, стоит рассмотреть возможность разделения кода по маршрутам – одна сборка для `/news`, другая для `/account`, и т.д.

3. Подход к разработке:

   Если сайт ориентирован на долго срочное развитие и расширение, то разумно использовать компонентную структуру для проекта: части сайта разбивать на отдельный модули, разрабатываемые и тестируемые изолированно. Все страницы собираются позднее из отдельных компонентов с контекстными дополнениями (если, конечно, контекстно-зависимые стили были разрешены к использованию).

   В таком подходе не сложно ввести новый функционал – можно разработать и протестировать новый модуль, а позже простым импортом загрузить его в нужном месте – или удалить не корректно функционирующий – так как модуль не связан (*decoupled*) с другими частями интерфейса/кода (он лишь предоставляет *интерфейс* [в программном смысле] для работы с ним), то достаточно удалить его папку и места импорта.

Расскажите о своем опыте работы над подобными сайтами...

В качестве frontend разработчика мне не доводилось работать над подобными проектами.

Ранее я удаленно занимался backend разработкой в небольшой веб-студии. Я выполнял преимущественно задачи интеграции CMS и верстки, развития и поддержки внутреннего движка, создания новых компонентов на PHP для него. Также я старался брать на себя задачи разработки логики взаимодействия между сервером и клиентом (сборка данных, отправка на сервер, обработка ответов), добавления обработчиков для переключения

состояний интерфейса и по мере возможности проектирования этих состояний.

Q3. При разработке интерфейсов с использованием компонентной архитектуры часто используются термины *Presentational Components* и *Container Components*...

С таким разделением сущностей я познакомился недавно, в ходе выполнения тестового задания на React. Мне кажется, подобная классификация – следствие, вытекающее из декларативного характера разработки на React.

Для себя я трактую эти термины так:

— *Presentational Components* – компоненты, представляющие части пользовательского интерфейса. Такие компоненты преимущественно содержат разметку и стили, а также инкапсулируют базовый контроль своего состояния (переключение цвета по клику, открытие в полный размер, и т.д.), предоставляя программный интерфейс для передачи им данных, необходимых компоненту для отображения (обычно это передача каких-либо данных через `props` или `props.children`).

— *Container Components* – контейнеры, реализующие основную логику приложения. Обычно они реализуют представление лишь за счет отображения существующих *presentational components.* Эти части приложения описывают процессы, которые должны выполняться в ходе взаимодействия с приложением (обновление store, отправка/загрузка данных).

Такое разделение оправдано в условиях React (как минимум), поскольку библиотека старается предоставить как можно более декларативный способ взаимодействия разных частей программы.

Но, учитывая, что декларативный подход плохо описывает действия, то выделяются – или естественным образом хочется выделить – компоненты, имеющие минимум разметки и все больше чистого JS кода – в итоге, такое разделение распределяет обязанности (*concerns*) за которые отвечают разные типы компонентов и увеличивает возможность переиспользования

кода (поскольку UI компоненты просто принимают данные на вход и как-то их отображают, то можно использовать любой компонент, предоставляющий данные, лишь бы сигнатура данных была подходящей).

Подобная практика от части схожа с разделением на View и Controller в рамках классического MVC подхода. Только в этом случае сохраняется сильная сторона изоморфной (т.е. использующееся для всего) технологии (или языка) – можно все время работать в рамках одного контекста, без необходимости переключаться на разные технологии (или языки) для решения проблем одного проекта.

В некоторых случаях недостатком можно назвать сложность определения назначения компонента (он должен относиться к *Presentational* или *Container'ам*?).

Также иногда высокий уровень вложенности презентационных компонентов вынуждает указывать props у верхних компонентов для данных, которые необходимо просто передать глубоко вниз по дереву (частично эта проблема может быть решена при помощи React Context API).

Q4. Как устроено наследование в JS? Расскажите о своем опыте реализации JS-наследования без использования Фреймворков.

В основе JS лежит прототипное наследование: каждый объект в JS имеет ссылку (в спецификации обозначающуюся как `[[Prototype]]`, по факту доступную в браузере как свойство `__proto__`), которая указывает на его объект-прототип. В классическом ООП, аналогом термина прототип можно считать суперкласс.

У одного объекта может быть только один прототип (или ни одного), однако подобная цепочка не ограничена (т.е. у прототипа может быть свой прототип и так далее), завершается она значением `null`.

Идея прототипного наследования в вебе была использована, в целях оптимизации хранения общих свойств и методов за счет вынесения их в отдельный объект. При этом, если в обычном объекте свойство или метод не найдены, исполнить JS-кода пройдет по цепочке, вплоть до `null` (конец цепочки прототипов) в попытке найти их.

На практике объект с необходимым прототипом (или цепочкой прототипов) создают либо при помощи `Object.create(proto)`, либо с помощью функции конструктора (любая функция, которая вызывается с помощью ключевого слова `new`).
Первый способ сохраняет ссылку на объект-прототип в скрытом свойстве `[[Prototype]]` объекта, который создается.

Второй способ создает новый экземпляр объекта, в качестве прототипа которого указывается объект `Func.prototype`, где `Func` – функция-конструктор (еще используется термин *функциональный класс*). По умолчанию, `Func.prototype` – просто чистый JS-объект со свойствами `__proto__` и `constructor`.

При использовании второго способа можно легко добиться динамического расширения всех ранее созданных с помощью `new Func` объектов, так как в JS передача объектов происходит по ссылке, а не по значению.

Мне не приходилось сталкиваться с необходимостью создания сложных цепочек наследования. И все же из задач, с которыми я сталкивался, была реализация класса `Modal`, который был расширение простого класса `Popup`.

`Popup` отвечал за отображение по клику на управляющий контр**о**л окна поверх основного интерфейса и скрытие этого окна при клике на закрывающий контр**о**л.

`Modal` расширял интерфейс `Popup` возможностью назначения событий дополнительным управляющим контролам, внутри этого окна, а также позволял изменять какие-либо связанные с `Modal` состояния (например, сохранять изменения произведенные над компонентами внутри него).

При этом `Popup` продолжал имплементировать базовое поведение открытия/закрытия и назначение обработчиков для этих событий.

Сейчас я предпочитаю использовать *композиционный* подход, вместо наследования, поскольку он гибче (позволяет комбинировать поведения независимо от иерархии) и доступней другим разработчикам для понимания.

```
const MaleWorker = (state) => Object.assign(
  {}, Male(state), Human(state), Mammal(state),
);
const theMan = MaleWorker();
```

Q5. Какие библиотеки можно использовать для написания тестов end-to-end во фронтенде?

К сожалению, в практическом плане я совсем слабо знаком с E2E тестами.

Из инструментов читал вводные статьи про *Selenium* и *PhantomJs*.
Однако, в условиях существования Chrome с headless режимом, я не уверен, что данные технологии еще активно используются на практике (поддержка *PhantomJs* была приостановлена кажется еще в 2017).

Также я неоднократно слышал про *Puppeteer*, но использовать его на практике мне тоже не доводилось.

Мне кажется, что use-кейсы E2E тестов в большей степени могут быть покрыты при помощи использования *Jsdom* (имплементация стандарта DOM в *NodeJs* окружении) и техник вроде *snapshot*-тестов.

Однако проверить реальное поведение интерфейса при взаимодействии пользователя вне реального браузера действительно остается не возможным без E2E тестирования.

Q6. Вам нужно реализовать форму для отправки данных на сервер, состоящую из нескольких шагов. В вашем распоряжении дизайн формы и статичная верстка, в которой не показано, как форма должна работать в динамике.

Думаю, в этом случае я запрошу пояснительную записку к техническому заданию, более подробно описывающую вопросы динамического взаимодействия.

В то время, пока информация будет подготавливаться, реализую

абстрактный механизм перехода между состояниями формы, сборку данных с полей и механизм валидации.

Эти составляющие являются логикой формы и не зависят от её представления (хотя и могут изменять состояние представления при необходимости, это нужно учитывать при разработке программного интерфейса).

Q7. Расскажите, какие инструменты помогают вам экономить время в процессе написания, проверки и отладки кода.

Собственный boilerplate репозиторий позволяет быстро развернуть новый проект, с настроенными линтерами (*ESLint*, *Stylelint*) и базовыми инструментами (вроде *browser-sync*, *webpack* или *gulp*).

Снипеты кода в VSCode также помогают быстрее писать типичные конструкции (тесты, создание компонентов, подключение модулей).

Для тестирования кода я использую unit и интеграционные тесты (*jest/mocha*, *chai*, *sinon*, *enzyme*).

В отладке кода неплохо помогает Console API (`.dir()`, `.time()`, `.group()` и т.д.), а также расширение *Chrome Debugger* для VSCode.

Q8. Какие ресурсы вы используете для развития в профессиональной сфере? Приведите несколько конкретных примеров (сайты, блоги и так далее).

Поскольку реальный список достаточно большой приведу самые ценные:

- Блог Axel Rauschmayer – помимо потрясающей серии книг Exploring ES, он публикует статьи по разным составляющим JS в блоге. Регулярно публикует обзор обсуждаемых TC39 нововведений для EcmaScript.

- Egghead.io – отличный сайт с большим количеством интересных курсов по разным аспектам веб-разработки, к тому же много бесплатных.

- CSS Tricks – блог Криса Койера, где часто публикуются крайне полезные статьи по CSS/HTML и JS. Я до сих признателен за отличную статью Styling Cross-Browser Compatible Range Inputs with CSS (хоть это и был гостевой пост, но все же с поддержкой Криса).

- Javascript.info – интерактивная книга (или курс), написанный Ильей Кантором, существует на английском и русском языках. Если судить по коммитам в репозиторий, английская версия гораздо интенсивней развивается.

- Google Web Fundamentals – множество отличных материалов и статей по разным технологиям, API и важным понятиям в вебе. Такие вещи, как Intersection Observer, Promises, Layout Thrashing я изучал именно там.

- Mozilla Developer Network – все самое важное про веб. Недавно открыли новый раздел CSS Layout Cookbook, который со временем должен превратится в ценный источник практических примеров верстки UI для начинающих фронтенд разработчиков.

Из не относящихся к работе дисциплин, я больше всего интересуюсь математикой, экономикой и психологией (страшный сет, это точно).

Последнее в значительной степени сказалось на желании заниматься менторством и обучением новичков в разработке.

Q9. Расскажите нам немного о себе и предоставьте несколько ссылок на последние работы, выполненные вами.

Меня зовут Виталий. Мне 20 лет. На текущий момент я занимаюсь сменой специализации с backend разработчика на frontend – меня привлекает динамичность развития фронтенда.

К тому же, даже занимаясь серверной разработкой, мне всегда нравилось решать задачи проектирования интерфейсов с помощью HTML и CSS, если была возможность взять их на себя.

Я однозначно фанат изучение всего нового. Я даже готов сказать, что высокая скорость обучения – один из самых важных навыков в моей жизни.

За последние несколько месяцев я быстро прогрессировал (и продолжаю это уверенно делать) в направлении фронтенда, используя свой опыт разработки в соседних направлениях программной инженерии.

Я привил себе практики тестирования JS-кода, погрузился в React, познакомился с системами mocking'а (вроде Sinon). В тоже время, я стараюсь не терять в понимании исполнения кода в условиях реального окружения (в частности в браузерах) - Event Loop, Tasks Queue, Layout (Reflow), Layout Thrashing – с данными концепциями я познакомился подробно не так давно, но они уже сказались на моих привычках писать код.

И, что как мне кажется важным, я не считаю HTML и CSS каким-то придатком к <новому популярному JS-фреймоворку> – я большой фанат интерфейсов, разработанных на чисто Vanill'ных технологиях в вебе.

За последние пару месяцев мне удалось поучаствовать в отборе Яндекс.ШРИ (хоть которой я и не прошел, alas), посоревноваться в конкурсе по фронтенду Блиц (проводимый также при Яндексе), а также уделить огромное количество своего времени самостоятельной практике и прохождению разнообразных онлайн-курсов.

Я регулярно нахожу и решаю практические «задачки» по фронтенду и алгоритмам.

В свободное время я занимаюсь спорт (особенно calisthenics), танцую в стиле hip-hop, люблю петь. Обожаю читать книги в оригинале по вечерам и поздней ночью.

Фанат классического крепкого американо. 😎

Тестовое задание компании FunBox - *Редактор маршрутов*:
https://github.com/VitalyKrenel/route-editor

Лого на чистом CSS и одним тегом:
https://codepen.io/VitalyKrenel/pen/WayWzQ

Вычисление максимального шрифта для контейнера без переноса текста (использование DOM только для получения начальных данных): https://codepen.io/VitalyKrenel/pen/mzavZa

Задание к ШРИ - *Яндекс.Дом*: https://github.com/VitalyKrenel/yandex-entrance-task-2 (пока не завершенное до конца, но я продолжаю работать над ним)

Задание к ШРИ - Алгоритм для вычисления расписания устройств умного дома: https://github.com/VitalyKrenel/yandex-entrance-task-3