14)React JS. Архитектура и основные принципы разработки приложений.

React — это инструмент для создания пользовательских интерфейсов. Его главная задача — обеспечение вывода на экран того, что можно видеть на веб-страницах. React значительно облегчает создание интерфейсов благодаря разбиению каждой страницы на небольшие фрагменты. Мы называем эти фрагменты компонентами.

Компонент React — это, если по-простому, участок кода, который представляет часть веб-страницы.

React использует язык программирования, называемый JSX, который похож на HTML, но работает внутри JavaScript, что отличает его от HTML.

15)Компоненты React. State & props. "Умные" и "глупые" компоненты.

До сих пор мы писали компоненты в виде функций. Их называют функциональными компонентами. Однако, компоненты можно писать и иначе, в виде классов JavaScript. Их называют классами компонентов. ???Классы компонентов должны содержать функцию, называемую render(). ???

Состояние(state) — это инструмент, позволяющий обновлять пользовательский интерфейс, основываясь на событиях.

Когда меняется состояние компонента, он снова вызовет функцию render. Мы можем изменить состояние с помощью this.setState(), если передадим этой функции объект, представляющий новое состояние. Компонент на странице всегда будет представлять своё текущее состояние. React самостоятельно обеспечивать такое поведение компонентов.

При вызове setState изменение состояния не производится мгновенно. React ждёт немного для того, чтобы увидеть, не нужно ли внести ещё какие-то изменения, и только потом производит изменение состояния. Это означает, что нельзя точно знать, каким будет состояние компонента после вызова setState.

Компоненты могут «общаться» друг с другом. Посмотрим, как это работает. Мы можем сообщить PlayButton, проигрывается музыка или нет, используя так называемые свойства (props). Свойства — это информация, коллективно используемая родительским компонентом и компонентами-потомками.

16) Разметка страниц в React-приложениях. JSX.

Монолитный код, не разделённый на части, это проблема, которая, как и многие другие, нарушает принцип единственной ответственности. Хорошо написанный код должен читаться как поэма, а я готов поспорить, что код метода render нашего компонента выглядит куда хуже. Займёмся решением этой проблемы.  
  
Поля ввода, используемые в форме, практически идентичны, и тому и другому нужны некие механизмы проверки данных. Применим к нашему компоненту некоторые приёмы рефакторинга и создадим компонент TextField, подходящий для повторного использования.

Обычно вы разделяете представления (HTML) и функциональность (JavsScript). Это приводит к монолитному JavaScript файлу, содержащему всю функциональность для одной страницы, и вы должны следить за сложным потоком JS->HTML->JS->неприятная ситуация.  
  
Связывание функциональности напрямую с разметкой и упаковка этого в портативный, автономный «компонент», сделает вас счастливее, а ваш код в целом лучше. Ваш Javasacript «хорошо знаком» с вашим HTML, так что смешивать их имеет смысл.

17) Навигация в React-приложениях. ReactRouter.

В любом реальном веб-приложении нужны маршруты, и приложение React не исключение. Пользователь должен видеть, где он находится в приложении в любой момент времени. А видит он свое текущее местоположение в адресной строке браузера. Следовательно приложение должно уметь сопоставлять определённый URL с соответствующей ему страницей. То есть, если мы введём в адресную строку например <https://health-imperium/appointmens>, то приложение должно направить нас на страницу списка приёмов, но не на какую-либо другую.

Также должна работать история. То есть когда пользователь кликает на стрелку "Назад" в браузере, приложение должно направить нас на предыдущую страницу.

ReactRouter популярна, довольна проста в использовании и обладает [**хорошей документацией**](https://reacttraining.com/react-router/web/guides/quick-start). Она предоставляет такие возможности как:

* Навигация по клику (компонент <Link>)
* Перенаправление (компонент <Redirect>)
* Маршрутизация (компонент Route)
* История (свойство history)

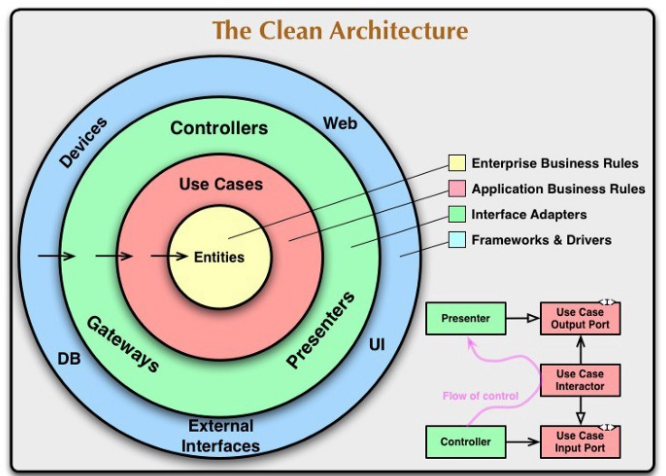
18) Управление состоянием интерфейса. Redux.

Redux — это контейнер предсказуемых состояний для JavaScript приложений.Он помогает вам писать приложения, которые ведут себя последовательно, работают в разных средах (клиентских, серверных и нативных), и которые легко тестировать. Кроме того, он предоставляет отличные возможности для разработчиков, такие как [редактирование кода в сочетании с временным отладчиком кода](https://github.com/reduxjs/redux-devtools).

Короче говоря, Redux предоставляет вам один объект, который хранит состояние всего вашего приложения в одном месте, который может включать данные из вашего серверного API или внешнего API, состояния навигации, информацию о пользователе, переключенное состояние кнопки и т.д.

19) Angular: архитектура и основные принципы разработки приложений.

Архитектура:



1.Основные сущности (Core entities): Это простые модели данных, которые по существу необходимы для представления нашей основной логики, построения потока данных и обеспечения работы нашего бизнес-правила.

2.Случаи использования (Usecases): Они построены на основе основных сущностей и реализуют всю бизнес-логику приложения. Случаи использования должны «жить в своем собственном мире» и делать только то, что они должны делать.

3.Правила зависимости (Dependency rule):Каждый уровень должен иметь доступ только к нижестоящему уровню. Так что уровень usecase должен использовать только entities которые определены в уровне entity, и контроллер должен использовать только usecase-ы из уровня usecase который находится ниже. Вы получите лучшее понимание того, как это работает, если вы посмотрите на практическую статью.

4.Адаптеры интерфейсов (Interface adapters): Чтобы получить рабочую архитектуру за границы каждого уровня, лучше всего работать с интерфейсными адаптерами из ядра (уровня сущностей и сценариев использования), чтобы обеспечить однородную структуру во всех проектах, что в конечном итоге выглядит как головоломка.

Основные принципы:  
**1.Независимость от фреймворков (Independent of Frameworks):** Наша бизнес-логика и правила не должны изменяться любой функциональность или структурой Angular. За счет этого достигается большая гибкость когда обновляемся до новой версии (которая может внести новую “архитектуру”/структуру проекта) или переключаемся на совершенно другой фреймворк —потому что нам не нужно пологаться на рамки какого либо фреймворка. А так же переход к следующему пункту: тестируемость!

**2.Тестируемость (Testable):** Наивно полагать, что нам нужно только проверить нашу бизнес-логику и правила, что значительно облегчит тестирование. Таким образом что бы протестить основную функциональность нашего приложения, ненужны специфичные тесты на подобии Karma.

**3.Независимость от UI (Independent of UI):** Бизнес -правила не заботит как они запускаются или контроллируются. Так что The business rules don’t care about how they are triggered or controlled. Поэтому ни какая бизнес логика не должна находиться внутри контроллера графического интерфейса.

**4.Не зависисмость от базы данных (Independent of Database):** В основном, веб-приложения не сильно заботятся об использовании базы данных потому что они берут данные в основном через API. Но смотря на шаблон репозиторием, большая часть приложений делают нейкий вид CRUD-операций к некому адресу. Это так же может быть абстрагированно к репозиторию.Дальше в этой статье, вы увидите как репозитории и API интегрируются.

**5.Независимость от сторонних агентов (Independent of any external agency):** Как упоминалось ранее, ядро нашего логики-приложения не должно зависить от функциональности Angular.

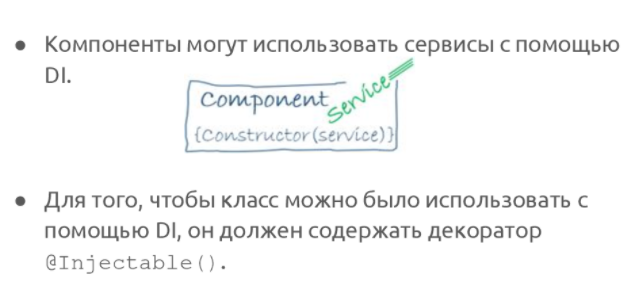
20) Angular: модули, компоненты, сервисы и DI.

На самом деле, главная цель модуля — группирование компонентов и/или сервисов, связанных друг с другом. Приложение Angular состоит из отдельных модулей. Как правило, приложения состоят из нескольких модулей. И каждое приложение Angular как минимум имеет один корневой модуль (root module), который, согласно условностям, называется **AppModule**.

Одним из ключевых элементов приложения являются компоненты. Компонент управляет отображением представления на экране. Для создания компонента необходимо импортировать функцию декоратора @Component из библиотеки @angular/core. Декоратор @Component позволяет идентифицировать класс как компонент.

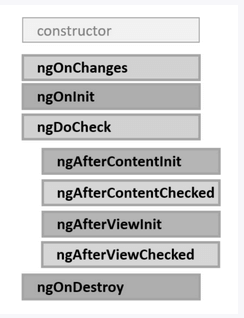
Сервисы в Angular представляют довольно широкий спектр классов, которые выполняют некоторые специфические задачи, например, логгирование, работу с данными и т.д. Стандартные задачи сервисов:

* Предоставление данных приложению. Сервис может сам хранить данные в памяти, либо для получения данных может обращаться к какому-нибудь источнику данных, например, к серверу.
* Сервис может представлять канал взаимодействия между отдельными компонентами приложения
* Сервис может инкапсулировать бизнес-логику, различные вычислительные задачи, задачи по логгированию, которые лучше выносить из компонентов. Тем самым код компонентов будет сосредоточен непосредственно на работе с представлением. Кроме того, тем самым мы также можем решить проблему повторения кода, если нам потребуется выполнить одну и ту же задачу в разных компонентах и классах



21) Angular: шаблоны страниц, жизненный цикл компонентов, подключение CSS.

???Шаблон страниц???



Стилизация компонента может производиться как с помощью установки стилей в самом компоненте, так и с помощью подключения внешних css-файлов. Для установки стилей в директиве @Component определено свойство styles:

При использовании стилей следует учитывать, что они применяются локально только к разметке, управляемой компонентом.

Параметр styleUrls позволяет указать набор файлов css, которые применяются для стилизации. В данном случае предполагается, что файл css располагается в проекте в папке app.

22) Angular: клиент-серверное взаимодействие, создание, отправка и валидация данных форм.