Questions for Interview

Web технологии

Протокол передачи гипертекста (Hypertext Transfer Protocol - HTTP) - это прикладной протокол для передачи гипертекстовых документов, таких как HTML. Он создан для связи между веб-браузерами и веб-серверами, хотя в принципе HTTP может использоваться и для других целей. Протокол следует классической клиент-серверной модели, когда клиент открывает соединение для создания запроса, а затем ждет ответа. HTTP - это протокол без сохранения состояния, то есть сервер не сохраняет никаких данных (состояние) между двумя парами "запрос-ответ".

- HTTP-метод, обычно глагол подобно GET, POST или существительное, как OPTIONS или HEAD, определяющее операцию, которую клиент хочет выполнить. Обычно, клиент хочет получить ресурс (используя GET) или передать значения HTML-формы (используя POST), хотя другие операция могут быть необходимы в других случаях.
- Путь к ресурсу: URL ресурсы лишены элементов, которые очевидны из контекста, например без protocol (http://), domain (здесь developer.mozilla.org), или TCP port (здесь 80).
- Версию HTTP-протокола.
- Заголовки (опционально), предоставляюшие дополнительную информацию для сервера.
- Или тело, для некоторых методов, таких как POST, которое содержит отправленный ресурс.

- **GET** запрашивает представление ресурса. Запросы с использованием этого метода могут только извлекать данные.
- **HEAD** запрашивает ресурс так же, как и метод GET, но без тела ответа.
- **POST** используется для отправки сущностей к определённому ресурсу. Часто вызывает изменение состояния или какие-то побочные эффекты на сервере.
- **PUT** заменяет все текущие представления ресурса данными запроса.
- **DELETE** удаляет указанный ресурс.
- **CONNECT** устанавливает "туннель" к серверу, определённому по ресурсу.
- **OPTIONS** используется для описания параметров соединения с ресурсом.
- **TRACE** выполняет вызов возвращаемого тестового сообщения с ресурса.
- **PATCH** используется для частичного изменения ресурса.

Cross-Origin Resource Sharing (CORS) — механизм, использующий дополнительные HTTP-заголовки, чтобы дать возможность агенту пользователя получать разрешения на доступ к выбранным ресурсам с сервера на источнике (домене), отличном от того, что сайт использует в данный момент. Говорят, что агент пользователя делает запрос с другого источника (cross-origin HTTP request), если источник текущего документа отличается от запрашиваемого ресурса доменом, протоколом или портом.

В целях безопасности браузеры ограничивают cross-origin запросы, инициируемые скриптами. Например, XMLHttpRequest и Fetch API следуют политике одного источника (same-origin policy). Это значит, что web-приложения, использующие такие API, могут запрашивать HTTP-ресурсы только с того домена, с которого были загружены, пока не будут использованы CORS-заголовки.

HTTP cookie (web cookie, cookie браузера) - это небольшой фрагмент данных, отправляемый сервером на браузер пользователя, который тот может сохранить и отсылать обратно с новым запросом к данному серверу. Это, в частности, позволяет узнать, с одного ли браузера пришли оба запроса (например, для аутентификации пользователя). Они запоминают информацию о состоянии для протокола HTTP, который сам по себе этого делать не умеет.

Cookie используются, главным образом, для:

- Управления сеансом (логины, корзины для виртуальных покупок)
- Персонализации (пользовательские предпочтения)
- Мониторинга (отслеживания поведения пользователя)

Babel.JS – это транспайлер, переписывающий код на ES-2015+ в код на предыдущем стандарте ES5.

Обычно Babel.JS работает на сервере в составе системы сборки JS-кода (например webpack или brunch) и автоматически переписывает весь код в ES5.

Настройка такой конвертации тривиальна, единственно – нужно поднять саму систему сборки, а добавить к ней Babel легко, плагины есть к любой из них.

Конфигурация Babel прописывается в файле babel.config.js, либо в .babelrc для настроек одного пакета, а также в package.json или .babelrc.js

Протокол WebSocket («веб-сокет»), описанный в спецификации RFC 6455, обеспечивает возможность обмена данными между браузером и сервером через постоянное соединение. Данные передаются по нему в обоих направлениях в виде «пакетов», без разрыва соединения и дополнительных HTTP-запросов.

...

JavaScript

Движок браузера выполняет JavaScript в одном потоке. Для потока выделяется область памяти —стэк, где хранятся фреймы (аргументы, локальные переменные) вызываемых функций.

Список событий, подлежащих обработке формируют очередь событий. Когда стек освобождается, движок может обрабатывать событие из очереди. Координирование этого процесса и происходит в event loop.

Это по сути бесконечный цикл, в котором выполняются многочисленные обработчики событий. Если очередь пустая—движок браузера ждет, когда поступит событие. Если непустая —первое в ней событие извлекается и его обработчик начинает выполняться. И так до бесконечности.
Детально описанно тут

Замыкание — это комбинация функции и лексического окружения, в котором эта функция была объявлена. Это окружение состоит из произвольного количества локальных переменных, которые были в области действия функции во время создания замыкания.

В глобальном контексте выполнения (за пределами каких-либо функций), this ссылается на глобальный объект вне зависимости от использования в строгом или нестрогом режиме.

В пределах функции значение this зависит от того, каким образом вызвана функция:

- Простой вызов - В этом случае значение this не устанавливается вызовом. Так как этот код написан не в строгом режиме, значением this всегда должен быть объект, по умолчанию - глобальный объект. В строгом режиме, значение this остается тем значением, которое было установлено в контексте исполнения. Если такое значение не определено, оно остается undefined. Для того что бы передать значение this от одного контекста другому необходимо использовать call или apply
- В стрелочных функциях, this привязан к окружению, в котором была создана функция. В глобальной области видимости, this будет указывать на глобальный объект.
- Когда функция вызывается как метод объекта, используемое в этой функции ключевое слово this принимает значение объекта, по отношению к которому вызван метод.

    - Решает проблему с this
    - etc.

Детально тут

Map – коллекция для хранения записей вида ключ:значение.

Set – коллекция для хранения множества значений, причём каждое значение может встречаться лишь один раз.

WeakSet – особый вид Set, не препятствующий сборщику мусора удалять свои элементы. То же самое – WeakMap для Map.

Детально тут

Детально тут

Классы в JavaScript были введены в ECMAScript 2015 и представляют собой синтаксический сахар над существующим в JavaScript механизмом прототипного наследования. Синтаксис классов **не вводит** новую объектно-ориентированную модель, а предоставляет более простой и понятный способ создания объектов и организации наследования.
Детально тут

Реализуют наследование через прототипы.
Детально тут

Promise – это специальный объект, который содержит своё состояние. Вначале pending («ожидание»), затем – одно из: fulfilled («выполнено успешно») или rejected («выполнено с ошибкой»).

Синтаксис создания Promise:

```
var promise = new Promise(function(resolve, reject) {
  //

  //       ,
  //     –    :
  // resolve()
  // reject()
})

//
//   promise
let promise = new Promise((resolve, reject) => {

  setTimeout(() => {
    //     fulfilled   "result"
    resolve("result");
  }, 1000);

});

// promise.then
promise
  .then(
    result => {
      //  - -    resolve
      alert("Fulfilled: " + result); // result -  resolve
    },
    error => {
      //   -    reject
      alert("Rejected: " + error); // error -  reject
    }
  );
```

Spread

```
var log = function(a, b, c) {
  console.log(a, b, c);
};

log(...['Spread', 'Rest', 'Operator']); // Spread Rest Operator
```

Использование оператора spread не ограничивается передачей параметров функции.
Несколько примеров его полезного использования

```
//
var arr = ['will', 'love'];
var data = ['You', ...arr, 'spread', 'operator'];
console.log(data); // ['You', 'will', 'love', 'spread', 'operator']

//
var arr = [1, 2, 3, 4, 5];
var data = [...arr];
console.log(data); // [1, 2, 3, 4, 5]

//  ,      ...     ,     .
var arr = [1, 2, 3, 4, 5];
var data = [...arr];
var copy = arr;
arr === data; // false
arr === copy; // true
```

Rest

В самом начале статьи я уже упоминал, что оператор `...` интерпретируется по-разному, в зависимости от контекста применения. *Spread* используется для разделения коллекций на отдельные элементы, а *rest*, наоборот, для соединения отдельных значений в массив.

```
var log = function(a, b, ...rest) {
  console.log(a, b, rest);
};

log('Basic', 'rest', 'operator', 'usage'); // Basic rest ['operator',
usage]
```

Детально тут

ES6 позволяет использовать метод `super` в (безклассовых) объектах с прототипами. Вот простой пример:

```
var parent = {
    foo() {
        console.log("  !");
    }
}

var child = {
    foo() {
        super.foo();
        console.log("  !");
    }
}

Object.setPrototypeOf(child, parent);
child.foo(); //   !
             //   !
```

`for...of` используется для перебора в цикле итерируемых объектов, например, массивов.

```
let nicknames = ['di', 'boo', 'punkeye'];
nicknames.size = 3;
for (let nickname of nicknames) {
    console.log(nickname);
}
// di
// boo
// punkeye
```

`for...in` используется для перебора в цикле всех доступных для перебора (enumerable) свойств объекта.

```
let nicknames = ['di', 'boo', 'punkeye'];
nicknames.size = 3;
for (let nickname in nicknames) {
    console.log(nickname);
}
// 0
// 1
// 2
// size
```

Деструктуризация помогает избежать использования вспомогательных переменных при взаимодействии с объектами и массивами.

```
function foo() {
    return [1, 2, 3];
}
let arr = foo(); // [1,2,3]

let [a, b, c] = foo();
console.log(a, b, c); // 1 2 3

function bar() {
    return {
        x: 4,
        y: 5,
        z: 6
    };
}
let { x: a, y: b, z: c } = bar();
console.log(a, b, c); // 4 5 6
```

TypeScript

В скобках, после номера вопроса, указана его сложность, оцениваемая по пятибалльной шкале.

TypeScript (TS) – это надмножество JavaScript (JS), среди основных особенностей которого можно отметить возможность явного статического назначения типов, поддержку классов и интерфейсов. Одним из серьёзных преимуществ TS перед JS является возможность создания, в различных IDE, такой среды разработки, которая позволяет, прямо в процессе ввода кода, выявлять распространённые ошибки. Применение TypeScript в больших проектах может вести к повышению надёжности программ, которые, при этом, можно разворачивать в тех же средах, где работают обычные JS-приложения.

Вот некоторые подробности о TypeScript:

- TypeScript поддерживает современные редакции стандартов ECMAScript, код, написанный с использованием которых, компилируется с учётом возможности его выполнения на платформах, поддерживающих более старые версии стандартов. Это означает, что TS-программист может использовать возможности ES2015 и более новых стандартов, наподобие модулей, стрелочных функций, классов, оператора spread, деструктурирования, и выполнять то, что у него получается, в существующих средах, которые пока этих стандартов не поддерживают.
- TypeScript – это надстройка над JavaScript. Код, написанный на чистом JavaScript, является действительным TypeScript-кодом.
- TypeScript расширяет JavaScript возможностью статического назначения типов. Система типов TS отличается довольно обширными возможностями. А именно, она включает в себя интерфейсы, перечисления, гибридные типы, обобщённые типы (generics), типы-объединения и типы-пересечения, модификаторы доступа и многое другое. Применение TypeScript, кроме того, немного упрощает работу за счёт использования вывода типов.
- Применение TypeScript, в сравнении с JavaScript, значительно улучшает процесс разработки. Дело в том, что IDE, в реальном времени, получает сведения о типах от TS-компилятора.
- При использовании режима строгой проверки на `null` (для этого применяется флаг компилятора `--strictNullChecks`), компилятор TypeScript не разрешает присвоение `null` и `undefined` переменным тех типов, в которых, в таком режиме, использование этих значений не допускается.
- Для использования TypeScript нужно организовать процесс сборки проекта, включающий в себя этап компиляции TS-кода в JavaScript. Компилятор может встроить карту кода (source map) в сгенерированные им JS-файлы, или создавать

отдельные .map-файлы. Это позволяет устанавливать точки останова и исследовать значения переменных во время выполнения программ, работая непосредственно с TypeScript-кодом.
- TypeScript — это опенсорсный проект Microsoft, выпущенный под лицензией Apache 2. Инициатором разработки TypeScript является Андерс Хейлсберг. Он причастен к созданию Turbo Pascal, Delphi и C#.

Делально тут

Декоратор — способ добавления метаданных к объявлению класса. Это специальный вид объявления, который может быть присоединен к объявлению класса, методу, методу доступа, свойству или параметру.

Декораторы используют форму @expression, где expression - функция, которая будет вызываться во время выполнения с информацией о декорированном объявлении.

И, чтобы написать собственный декоратор, нам нужно сделать его factory и определить тип:

- ClassDecorator
- PropertyDecorator
- MethodDecorator
- ParameterDecorator

Обобщённые типы (generics) позволяют создавать компоненты или функции, которые могут работать с различными типами, а не с каким-то одним.
Рассмотрим пример:

```
/**       */
class Queue<t> {
  private data = [];
  push = (item: T) => this.data.push(item);
  pop = (): T => this.data.shift();
}

const queue = new Queue<number>();
queue.push(0);
queue.push("1"); //  :      ,
```

Да, поддерживает. Существуют четыре основных принципа объектно-ориентированного программирования:

- Инкапсуляция
- Наследование
- Абстракция
- Полиморфизм

Пользуясь простыми и понятными средствами TypeScript, можно реализовать все эти принципы.

Файлы с расширением .map хранят карты кода (source map), которые содержат данные о соответствии кода, написанного на TypeScript, JavaScript-коду, созданному на его основе. С этим файлами могут работать многие отладчики (например — Visual Studio и инструменты разработчика Chrome). Это позволяет, в ходе отладки, работать с исходным кодом программ на TypeScript, а не с их JS-эквивалентами.

TypeScript поддерживает геттеры и сеттеры, которые позволяют управлять доступом к членам объектов. Они дают разработчику средства контроля над чтением и записью свойств объектов.

```
class foo {
  private _bar:boolean = false;

  get bar():boolean {
    return this._bar;
  }
  set bar(theBar:boolean) {
    this._bar = theBar;
  }
}

var myBar = myFoo.bar;  //
myFoo.bar = true;  //
```

Да, можно.

```
type NumberCallback = (n: number) => any;

class Foo {
    //
    save(callback: NumberCallback): void {
        console.log(1)
        callback(42);
    }
}

var numCallback: NumberCallback = (result: number) : void => {
    console.log("numCallback: ", result.toString());
}

var foo = new Foo();
foo.save(numCallback)
```

Вот примеры использования этих ключевых слов:

```
interface X {
    a: number
    b: string
}

type X = {
    a: number
    b: string
};
```

В отличие от объявления интерфейса, которое всегда представляет именованный тип объекта, применение ключевого слова `type` позволяет задать псевдоним для любой разновидности типа, включая примитивные типы, типы-объединения и типы-пересечения.

При использовании ключевого слова `type` вместо ключевого слова `interface` теряются следующие возможности:

- Интерфейс может быть использован в выражении `extends` или `implements`, а псевдоним для литерала объектного типа — нет.
- Интерфейс может иметь несколько объединённых объявлений, а при использовании ключевого слова `type` эта возможность не доступна.

Angular

Фреймворк - этого хватит

> Это платформа для разработки мобильных и десктопных веб-приложений. Наши приложения теперь представляют из себя «толстый клиент», где управление отображением и часть логики перенесены на сторону браузера. Так сервер уделяет больше времени доставке данных, плюс пропадает необходимость в постоянной перерисовке. С Angular мы описываем структуру приложения декларативно, а с TypeScript начинаем допускать меньше ошибок, благодаря статической типизации. В Angular присутствует огромное количество возможностей из коробки. Это может быть одновременно и хорошо и плохо, в зависимости от того, что вам необходимо.

...

...

MVVM - шаблон проектирования архитектуры приложения. Состоит из 3 ключевых блоков: Model, View, ViewModel.
Отличие от MVC заключаются в:

- View реагирует на действия пользователя и передает их во View Model через Data Binding.
- View Model, в отличие от контроллера в MVC, имеет особый механизм, автоматизирующий связь между View и связанными свойствами в ViewModel.

Привязка данных между View и ViewModel может быть односторонней или двусторонней (one-way, two-way data-binding).

- Структурные директивы влияют на DOM и могут добавлять/удалять элементы (ng-template, NgIf, NgFor, NgSwitch, etc)
- Атрибутные директивы меняют внешний вид или поведение элементов, компонентов или других директив NgStyle, NgClass, etc).

ng-template

`<template>` — это механизм для отложенного рендера клиентского контента, который не отображается во время загрузки, но может быть инициализирован при помощи JavaScript.

Template можно представить себе как фрагмент контента, сохранённый для последующего использования в документе. Хотя парсер и обрабатывает содержимое элемента `template` во время загрузки страницы, он делает это только чтобы убедиться в валидности содержимого; само содержимое при этом не отображается.

`<ng-template>` - является имплементацией стандартного элемента template, данный элемент появился с четвертой версии Angular, это было сделано с точки зрения совместимости со встраиваемыми на страницу template элементами, которые могли попасть в шаблон ваших компонентов по тем или иным причинам.

ng-container

`<ng-container>` - это логический контейнер, который может использоваться для группировки узлов, но не отображается в дереве DOM как узел (node).

На самом деле структурные директивы (*ngIf, *ngFor, …) являются синтаксическим сахаром для наших шаблонов. В реальности, данные шаблоны трансформируются в такие конструкции

ng-content

`<ng-content>` - позволяет внедрять родительским компонентам html-код в дочерние компоненты.

Здесь на самом деле, немного сложнее уже чем с ng-template, ng-container. Так как ng-content решает задачу проецирования контента в ваши веб-компоненты. Веб-компоненты состоят из нескольких отдельных технологий. Вы можете думать о Веб-компонентах как о переиспользуемых виджетах пользовательского интерфейса, которые создаются с помощью открытых веб-технологий. Они являются частью браузера и поэтому не нуждаются во внешних библиотеках, таких как jQuery или Dojo. Существующий Веб-компонент может быть использован без написания кода, просто путем импорта выражения на HTML-страницу. Веб-компоненты используют новые или разрабатываемые стандартные возможности браузера.

Давайте представим ситуацию от обратного, нам нужно параметризовать наш компонент. Мы хотим сделать так, чтобы на вход в компонент мы могли передать какие-либо статичные данные. Это можно сделать несколькими способами.

- Компонент контролирует участок экрана, т.н. view.
- Сервис это класс с узкой, четко определенной целью. Это может быть значение, функция, запрос, etc. Главное в них то, что они повторно используются, отделяя чистую функциональность компонента.
- Пайп преобразует отображение значений в шаблоне, к примеру отображение дат в разных локалях или изменяют в отображении регистр строк.

Запуск Angular приложения начинается с файла **main.ts**. Этот файл содержит в себе примерно следующее:

```
import { platformBrowserDynamic } from "@angular/platform-browser-
dynamic";
import { AppModule } from "./app/app.module";

const platform = platformBrowserDynamic();

platform.bootstrapModule(AppModule);
```

platformBrowserDynamic запускает AppModule. После этого, начинает работать логика в AppModule.

В AppModule обычно задается компонент, который будет использоваться для отображения при загрузке. Компонент находится в параметре **bootstrap**

```
@NgModule({
  imports: [BrowserModule, FormsModule],
  declarations: [AppComponent],
  bootstrap: [AppComponent],
})
export class AppModule {}
```

Можно сказать ок, если скажет хотябы 4

После создания компонента или директивы через вызов конструктора, Angular вызывает методы жизненного цикла в следующей последовательности в строго определенные моменты:

- ngOnChanges() - вызывается когда Angular переприсваивает привязанные данные к input properties. Метод получает объект SimpleChanges, со старыми и новыми значениями. Вызывается перед NgOnInit и каждый раз, когда изменяется одно или несколько связанных свойств.
- ngOnInit() - инициализирует директиву/компонент после того, как Angular впервые отобразит связанные свойства и устанавливает входящие параметры.
- ngDoCheck() - при обнаружении изменений, которые Angular не может самостоятельно обнаружить, реагирует на них.
- ngAfterContentInit() - вызывается после того, как Angular спроецирует внешний контент в отображение компонента или отображение с директивой. Вызывается единожды, после первого ngDoCheck().

- ngAfterContentChecked() - реагирует на проверку Angular-ом проецируемого содержимого. Вызывается после ngAfterContentInit() и каждый последующий ngDoCheck().
- ngAfterViewInit() - вызывается после инициализации отображения компонента и дочерних/директив. Вызывается единожды, после первого ngAfterContentChecked().
- ngAfterViewChecked() - реагирует на проверку отображения компонента и дочерних /директив. Вызывается после ngAfterViewInit() и каждый последующий ngAfterContentChecked().
- ngOnDestroy() - после уничтожения директивы/компонента выполняется очистка. Отписывает Observables и отключает обработчики событий, чтобы избежать утечек памяти.

Angular поддерживает одностороннюю и двустороннюю Data Binding. Это механизм координации частей шаблона с частями компонента.
Добавление специальной разметки сообщает Angular как соединять обе стороны. Следующая диаграмма показывает четыре формы привязки данных.
Односторонние:

- От компонента к DOM с привязкой значения: {{hero.name}}
- От компонента к DOM с привязкой свойства и присвоением значения: [hero]=" selectedHero"
- От DOM к компоненту с привязкой на ивент: (click)="selectHero(hero)"

Двусторонняя в основном используется в template-driven forms, сочетает в себе параметр и ивент.

NgZone - это сервис, который является обёрткой над zone.js, для выполнения кода внутри или вне зоны Angular. Этот сервис создаёт зону с именем angular для автоматического запуска обнаружения изменений, когда выполняются следующие условия:

- Когда выполняется синхронная или асинхронная функция
- Когда нет запланированной микро-задачи в очереди

Наиболее распространённое применение NgZone — это оптимизация производительности посредством выполнения асинхронной логики вне зоны Angular (метод `runOutsideAngular`), тем самым не вызывая обнаружение изменений или обработку ошибок. Или наоборот, данный сервис может использоваться для выполнения логики внутри зоны (метод `run`), что в конечном итоге приведёт к тому, что Angular снова вызовет обнаружение изменений и при необходимости перерисует представление.

Используется в директивах и компонентах для подписки на пользовательские ивенты синхронно или асинхронно, и регистрации обработчиков для этих ивентов.

Change Detection

Change Detection - процесс синхронизации модели с представлением. В Angular поток информации однонаправленный, даже при использовании `ngModel` для реализации двустороннего связывания, которая является синтаксическим сахаром поверх однонаправленного потока.

Change Detection Mechanism

Change Detection Mechanism - продвигается только вперед и никогда не оглядывается назад, начиная с корневого (рут) компонента до последнего. В этом и есть смысл одностороннего потока данных. Архитектура Angular приложения очень проста — дерево компонентов. Каждый компонент указывает на дочерний, но дочерний не указывает на родительский. Односторонний поток устраняет необходимость `$digest` цикла.

Всего есть две стратегии - `Default` и `OnPush`. Если все компоненты используют первую стратегию, то `Zone` проверяет все дерево независимо от того, где произошло изменение. Чтобы сообщить Angular, что мы будем соблюдать условия повышения производительности нужно использовать стратегию обнаружения изменений `OnPush`. Это сообщит Angular, что наш компонент зависит только от входных данных и любой объект, который передается ему должен считаться immutable. Это все построено на принципе автомата Мили, где текущее состояние зависит только от входных значений.

Конструктор сам по себе является фичей самого класса, а не Angular. Основная разница в том, что Angular запустит `ngOnInit`, после того, как закончит настройку компонента, то есть, это сигнал, благодаря которому свойства `@Input()` и другие байндинги, и декорируемые свойства доступны в `ngOnInit`, но не определены внутри конструктора, по дизайну.

Это важный паттерн шаблон проектирования приложений. В Angular внедрение зависимостей реализовано из-под капота.

Зависимости - это сервисы или объекты, которые нужны классу для выполнения своих функций. DI -позволяет запрашивать зависимости от внешних источников.

Это сервисы, объявленные в приложении и имеющие один экземпляр на все приложение. Его можно объявить двумя способами:

- Объявить его @Injectable(root)
- Включить его в AppModule в providers.

Interceptor (перехватчик) - просто причудливое слово для функции, которая получает запросы / ответы до того, как они будут обработаны / отправлены на сервер. Нужно использовать перехватчики, если имеет смысл предварительно обрабатывать многие типы запросов одним способом. Например нужно для всех запросов устанавливать хедер авторизации `Bearer`:

```
import { Injectable } from "@angular/core";
import {
  HttpInterceptor,
  HttpRequest,
  HttpHandler,
  HttpEvent,
} from "@angular/common/http";
import { Observable } from "rxjs/Observable";

@Injectable()
export class TokenInterceptor implements HttpInterceptor {
  public intercept(
    req: HttpRequest<any>,
    next: HttpHandler
  ): Observable<HttpEvent<any>> {
    const token = localStorage.getItem("token") as string;

    if (token) {
      req = req.clone({
        setHeaders: {
          Authorization: `Bearer ${token}`,
        },
      });
    }

    return next.handle(req);
  }
}
```

И регистрируем перехватчик как синглтон в провайдерах модуля:

```
import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { HTTP_INTERCEPTORS } from "@angular/common/http";
import { AppComponent } from "./app.component";
import { TokenInterceptor } from "./token.interceptor";

@NgModule({
  imports: [BrowserModule],
  declarations: [AppComponent],
  bootstrap: [AppComponent],
  providers: [
    {
      provide: HTTP_INTERCEPTORS,
      useClass: TokenInterceptor,
      multi: true, // <---
    },
  ],
})
export class AppModule {}
```

1. **NavigationStart** - начало навигации. Возникает во время нажатия на директиву **router link**, вызове функций **navigate** и **navigateByUrl**
2. **RoutesRecognized** - сопоставление URL-адресов и редиректы. Роутер сопоставляет URL-адрес навигации из первого события с одним из свойств path в конфигурации, применяя любые редиректы по-пути.
3. **GuardsCheckStart, GuardsCheckEnd** - функции, которые использует роутер для определения может ли он выполнить навигацию. Пример:

   ```
   const config = {
     path: "users",
     /* ... */
     canActivate: [CanActivateGuard],
   };

   class Guard {
     // router guard implementation
     canActivate(
       route: ActivatedRouteSnapshot,
       state: RouterStateSnapshot
     ): boolean {
       return this.auth.isAuthorized(route.queryParams.login);
     }
   }
   ```

   Если вызов `isAuthorized(route.queryParams.login)` возвращает true, guard завершится успехом. В противном случае guard упадет, роутер сгенерирует событие `NavigationCancel` и отменит всю дальнейшую навигацию.

   Другие guard включают `canLoad` (должен ли модуль быть лениво загружен или нет). `canActivateChild` и `canDeactivate` (которые полезны для предотвращения ухода пользователя со страницы, например, при заполнении формы).
4. **ResolveStart, ResolveEnd** - функции, которые мы можем использовать для подгрузки данных во время навигации. Например:

```
// router configuration
const config = {
  path: "users",
  /* ... */
  resolve: { users: UserResolver },
};

// router resolver implementation
export class UserResolver implements Resolve<Observable<any>> {
  constructor(private userService: MockUserDataService) {}
  resolve(): Observable<any> {
    return this.userService.getUsers();
  }
}
```

Результат, то есть данные, будет положен в `data` объект сервиса `ActivatedRoute` с ключом `users`. Данная информация может быть прочитаны с помощью подписки на d`ata observable`.

```
export class UsersComponent implements OnInit {
  public users = [];
  constructor(private route: ActivatedRoute) {}
  ngOnInit() {
    this.route.data
      .pipe(first())
      .subscribe((data) => (this.users = data.users));
  }
}
```

5. **ActivationStart, ActivationEnd, ChildActivationStart, ChildActivationEnd** - события, во время которых активируются компоненты и отображаются их с помощью. Роутер может извлечь необходимую информацию о компоненте из дерева ActivatedRouteSnapshots. Он был построен в предыдущие шаги навигационного цикла.
6. **Updating the URL** - последний шаг в навигационном цикле — это обновление URL-адреса в address bar

Роутинг позволяет реализовать навигацию от одного view приложения к другому при работе пользователя с приложением.
Это реализовано через взаимодействие с адресной строкой, Angular Router интерпретирует ее как инструкцию по переходу между view. Возможна передача параметров вспомогательному компоненту для конкретизирования предоставляемого контента.

- FormControl - отслеживает значение и статус валидации отдельного элемента формы.
- FormGroup - отслеживает состояние и статус валидации группы FormControl

Они используются для создания и работы с формами.

Реактивные формы обеспечивают управляемый моделями подход к обработке входных данных форм, значения которых могут меняться со временем. Они строятся вокруг наблюдаемых потоков, где входные данные и значения форм предоставляются в виде потоков входных значений, к которым можно получить синхронный доступ.

В реактивных формах валидация реализуется в компоненте. Есть два типа валидаторов: синхронные и асинхронные.
Можно использовать встроенные валидаторы, либо создать свои. Валидаторы добавляются

Angular приложение можно скомпилировать с помощью команд **ng serve** и **ng build**. При этом, можно работать с разными видами компиляции:

- **JIT** - (Just-In-Time compilation) - компиляция "на лету", динамическая компиляция. В Angular используется по умолчанию.
- **AOT** - (Ahead-Of-Time compilation) - компиляции перед исполнением.

Основные различия:

Параметры	JIT	AOT
Когда компилируется	в момент запуска приложения в браузере	в момент сборки приложения
Рекомендуется использовать для	локальной разработки	создания продуктовой сборки
Как включить	Не нужно выставлять дополнительных флагов	Нужно добавить флаг --aot или --prod
Скорость	Скорость компиляции быстрее, загрузка в браузере дольше	Скорость компиляции дольше, загрузка в браузере быстрее
Размер бандла	Бандл имеет большой размер из-за включенного в бандл компилятора.	Бандл имеет небольшой размер, тк содержит полностью скомпилированный и оптимизированный код.
Выявление ошибок	Ошибки отобразятся во время запуска приложения в браузере	Выявление ошибок во время компиляции
Работа с файлами	Может компилировать только измененные файлы отдельно	Компилирует сразу все файлы приложения