**component** - отвечает за часть web-страницы и включает в себя HTML-шаблон, CSS-стили и логику поведения;

**service** - поставщик данных для component;

**directive** - преобразует определенную часть DOM заданным образом.

Все вышеперечисленное собирается в корневой модуль, который общепринято называется **AppModule.**

**exports** - компоненты, сервисы, директивы и фильтры, доступные для использования разработчикам, которые будут использовать ваш модуль в своих разработках;

**bootstrap** - имя главного компонента приложения (как правило, называется AppComponent).

Сервисы нужны для предоставления данных компонентам. Это могут быть не только запросы к серверу, но и **функции**, преобразующие исходные данные по заданному алгоритму. Считается хорошей практикой все обращения к серверу и **функции, возвращающие данные**, выносить в сервисы.

**Контроллер** - экземпляр класса компонента.

**Некоторые DOM свойства:**

x.innerHTML - текстовое значение x

x.nodeName - название (имя) узла x

x.nodeValue - значение x

x.parentNode - родительский узел x

x.childNodes - дочерние узлы x

x.attributes - атрибутивные узлы x

Название свойства элемента DOM должно быть заключено в **квадратные скобки** или иметь **префикс bind-**, иначе значением будет строка с названием свойства компонента.

<p [class.red-text]="isRed">Some text.</p>

Здесь **red-text** - название класса, а **isRed** - переменная компонента типа boolean, значение которой определяет, будет ли в Angular шаблоне установлен класс или нет.

Каждое возникающее событие передает о себе всю информацию в объекте **$event**, который может быть передан методу класса в качестве аргумента.

<button on-click="showContacts($event)">Show Contacts List</button>

**Свойства объекта $event**:

**target** - элемент DOM, выступивший инициатором;

**target.value** - значение элемента DOM (справедливо для полей формы);

**keyCode** - код клавиши (справедливо для keyup).

Перечисленные выше свойства имеются только у стандартных событий браузера.

Для удобства в Angular предусмотрено так называемое псевдо событие **keyup.{keyCode}**, которое будет инициировано только в том случае, если будет нажата определенная клавиша. Например, для отслеживания нажатия Enter используется keyup.enter.

Пишем «new Event Emitter<any>», когда создаем свои события. Для стандартных событий типа “click” этого писать не нужно.

Метод **emit()** инициирует возникновение события.

Класс становится Angular компонентом, если его объявлению предшествует **декоратор @Component()** с объектом конфигурации.

**OnChanges** - устанавливаются или изменяются значения входных свойств класса компонента;

**OnInit** - устанавливаются "обычные" свойства; вызывается единожды вслед за первым вызовом OnChanges();

**DoCheck** - происходит изменения свойства или вызывается какое-либо событие;

**AfterContentInit** - в шаблон включается контент, заключенный между тегами компонента;

**AfterContentChecked** - аналогичен DoCheck(), только используется для контента, заключенного между тегами компонента;

**AfterViewInit** - инициализируются компоненты, которые входят в шаблон текущего компонента;

**AfterViewChecked** - аналогичен DoCheck(), только используется для дочерних компонентов;

**OnDestroy** - компонент "умирает", т.е. удаляется из DOM-дерева.

**<ng-content></ng-content>** отображает контент родительского компонента в дочернем.

**по поводу синхронности и асинхронности для наглядности примеры:**

==============

Синхронное выполнение:

console(1);

console(2);

console(3);

Выведет в консоль:

1

2

3

То есть код выполняется последовательно, строка за строкой.

==============

Асинхронное выполнение:

console.log(1);

setTimeout(() => console.log(2));

console.log(3);

Выведет:

1

3

2

То есть вывод цифры 2 происходит с задержкой, но это не остановило вывод цифры 3.

==============

Таймаут это самая простая имитация асинхронности. В жизни хороший пример это данные с сервера. Ты можешь сделать несколько запросов на сервер, а они могут выполняться за разное время и ты не ждёшь выполнения их по очереди, они происходят асинхронно и не зависят друг от друга.

**ControlValueAccessor** используется, когда мы создаем свой элемент формы, возможно, какой –нибудь необычный (например, светофор).

Назначение свойства **pathMatch**:

**const routes**:Routes=[

{**path**: **'login'**, **component**:LoginRouteComponent},

{**path**: **'home'**, **component**:HomeRouteComponent, **children**:[

{**path**: **'profile'**, **component**:ProfileRouteComponent}

]},

{**path**: **'contacts'**, **redirectTo**: **'/home'**, **pathMatch**: **'full'**, **children**:[

{**path**: **'director'**, **component**:DirectorContactsRouteComponent}

]},

{**path**: **'\*\*'**, **component**:LoginRouteComponent}

];

**@NgModule**({

**imports**:[RouterModule.**forRoot**(**routes**)],

**exports**:[RouterModule]

})

**export class** AppRoutingModule{}

У нас имеется маршрут, например, «contacts», прямой переход на который вызвал бы ошибку, если бы не было определено свойство redirectTo, поскольку для «contacts» не определен компонент.

Теперь представим, что redirectTo указан, а pathMatch - нет. Переход по маршруту /contacts/director отобразит компонент, указанный в redirectTo для contacts. Это происходит потому что при навигации на дочерний маршрут, Router проходит по дереву иерархии все его родительские URL и при попадании на /contacts сработает редирект.

Чтобы избежать этого в Angular routing указывается свойство pathMatch со значением full. И тогда перенаправление сработает только когда точно будет запрошен URL /contacts**.**

**LoadChildren:**

{path: 'order', loadChildren: './modules/order/order.module#OrderModule'}

Когда будет запрошен маршрут в состав которого входит order, тогда и загрузится OrderModule. Но в момент старта приложения он не включается в загрузку.

Использование **@ViewChild()** в родительском Angular component позволяет получить все свойства указанного дочернего компонента.

**Чтобы передавать доп. информацию при рутинге на другой стейт**:

import { Router, NavigationExtras } from '@angular/router';

private \_router: Router

const navigationExtras: NavigationExtras = {

state: {

isProfileFinished: true

}

};

this.\_router.navigate(['main/launch-plan'], navigationExtras).then();

получать:

this.\_router.getCurrentNavigation().extras.state.isProfileFinished

Подробнее на эту тему: <https://www.thepolyglotdeveloper.com/2016/10/passing-complex-data-angular-2-router-nativescript/>

Если **не** устанавливается **priming**:

npm update @angular/cli @angular/cdk rxjs

npm install -S @angular/material @angular/cdk @angular/animations

npm uninstall @angular/core

npm install -S @angular/core

**Как создать Resolver**:

<https://www.youtube.com/watch?v=fBJ-ziQdPuM>

**Методы и операторы Observable**:

1. **next()** - принимает значение, которое будет возвращено функции-обработчику (т. У. используется для передачи значений). Иными словами – *«Observable, вызови эту функцию, когда у тебя есть новое значение для меня»*;
2. **error()** - принимает значение, возвращаемое функции-обработчику ошибок. Иными словами – *«Observable, вызови эту функцию, когда у тебя есть новая error для меня»*;
3. **complete()** - вызывается для уведомления "подписчиков" об окончании рассылки. Иными словами – *«Observable, вызови эту функцию, когда закончишь свою работу»*;
4. Все операторы преобразования данных объявляются в методе **pipe()** через запятую. Сам метод pipe() импортировать не надо и он должен быть вызван перед методом subscribe().

**Операторы**— это функции, которыми можно преобразовывать данные между моментом, когда Observable их отправил, и моментом, когда подписчик их получил. Т.е. преобразовывать данные во время движения.

Oператор «**of»** используется для преобразования примитивных типов в объект Observable.

Если необходимо, чтобы обработчик вместо всего массива сразу получал каждый его элемент в отдельности, используется оператор «**from**».

**Для того, чтобы задать свой стиль элементам primeng:**

:host ::ng-deep { Your styles }

**Суффикс доллара** может быть использован для обозначения переменной являющейся экзмепляром Observable.

**Subjects** allow triggering the emission of a new value manually, but observables **don’t**.

На все контролы формы Angular динамически добавляет парные **CSS классы** в зависимости от определенных условий:

**ng-invalid/ng-valid** — меняется в зависимости от валидности контрола;

**ng-pristine/ng-dirty** — контрол считается dirty, если в нем хотя бы раз менялось значение;

**ng-untouched/ng-touched** — контрол считается touched при первой потере фокуса.

**“Чистые” функции** (pure functions), термин, описывающий функции, удовлетворяющие следующим условиям:

1. Они не должны делать внешних вызовов по сети или базе данных.
2. Они возвращают значение, зависящее только от переданных параметров.
3. Их аргументы являются неизменяемыми, т.е. функции не должны их изменять.
4. Вызов чистой функции с теми же аргументами всегда возвращает одинаковый результат.

Эти функции называют “чистыми”, потому что они не делают ничего, только возвращают значение, зависящее от параметров. Они не зависят от какой-либо из частей системы.

Подключение **ngx-translate**:

<https://www.youtube.com/watch?v=Ara2cQ4IQxA>

<https://medium.com/letsboot/translate-angular-4-apps-with-ngx-translate-83302fb6c10d>

**@Output**('last-price') lastPrice;

**lastPrice** – название, которое было дано в компоненте, где событие было создано.

**last-price** – новое название, которое можно присвоить к компоненте получателя.

**@ViewChild('commentRef', {static: false}) commentRef: ElementRef;**

Use { **static: true** } when you want to access the ViewChild in ngOnInit.

Use { **static: false** } will be accessible only in ngAfterViewInit. This is also what you want to go for when you have a structural directive (\*ngIf etc.) on your template.

In most cases { static: false } will work.

In most cases the @ViewChild/@ContentChild gets initialized after the main component initialization. So it is more safe to put the component initialization code that uses the references in the ngAfterViewInit Lyfecycle Method (for @ContentChild -> ngAfterContentInit).

**Renderer:**

<https://habr.com/ru/company/infowatch/blog/330030/>

Чтобы указать, что сервис сам может использовать другие сервисы, к классу сервиса применяется декоратор **Injectable**. Если класс не будет иметь подобного декоратора, то встроенный механизм внедрения зависимостей не сможет создать объект этого класса и выдаст ошибку.

Сервис по типу **Синглтона** – когда один объект сервиса доступен для всего приложения.

Если сервис определен в корневом модуле приложения, то во время работы приложения будет создаваться один объект данного сервиса для всего приложения.

**NGX strorage:**

Позволяет записывать данные в сторидж и потом подписываться на них.

<https://www.npmjs.com/package/ngx-webstorage>

**CSS transitions**

**-** предоставляют способ контролировать скорость анимации, при изменении CSS свойств.

**transition-property**

Указывает имя или имена свойств, чьи переходы должны анимироваться. Только свойства, указанные здесь, анимируются в переходах; изменение других свойств будет происходить обычным образом.

**transition-duration**

Определяет время происхождения перехода. Можно указать время анимирования всех свойств перехода сразу или для каждого свойства в отдельности.

**transition-timing-function**

Определяет функцию, указывающую, как вычисляются промежуточные значения свойств. Вы также можете выбрать ослабления от функции замедления Шпаргалка.Большинство [функций времени](https://developer.mozilla.org/en-US/docs/CSS/timing-function) может быть определено графиком соответствующей функции, образующей четырьмя точками кривую Безье.

Подобрать нужную функцию скорости можно здесь: <https://easings.net/ru>

**transition-delay**

Определяет как много должно пройти времени, перед тем как начнётся переход.

div {

transition: <property> <duration> <timing-function> <delay>;

}

padding: 10px(**top**) 20px(**right**) 10px(**bottom**) 20px(**left**);

When the position of an element is set to **relative**, it allows you to specify how CSS should move it relative to its current position in the normal flow of the page. It pairs with the CSS offset properties of *left* or *right*, and *top* or *bottom*.

Changing an element's position to relative does **not** remove it from the normal flow - other elements around it still behave as if that item were in its default position.

The next layout scheme that CSS offers is the **fixed** position, which is a type of absolute positioning that locks an element relative to the browser window (not to the parent element as in case of absolute positioning).

One key difference between the fixed and absolute positions is that an element with a fixed position won't move when the user scrolls.

**Сайт, который генерирует палитры цветов для создания дизайна:**

<https://coolors.co/3c91e6-9fd356-fafffd-342e37-fa824c>

**Схлопывание отступов (марджинов):**

Схлопываются только верхние и нижние отступы, отступы справа или слева не схлопываются никогда.

*Схлопывание* ***не*** *срабатывает:*

* для элементов, у которых установлено свойство padding.
* для элементов, у которых на стороне схлопывания задана граница;
* на элементах с абсолютным позиционированием, т.е. таких, у которых position установлено как absolute;
* на плавающих элементах (для них свойство float задано как left или right);
* для строчных элементов;
* для <html>.

*Также схлопывание* ***не*** *срабатывает при соблюдении некоторых условий:*

* для элементов, у которых значение overflow задано как auto, hidden или scroll схлопывание не действует для их дочерних элементов;
* у элементов, к которым применяется свойство clear, не схлопывается верхний отступ с нижним отступом родительского элемента.

**Cookie:**

**Куки** – это небольшие строки данных, которые хранятся непосредственно в браузере. Куки обычно устанавливаются веб-сервером при помощи заголовка **Set-Cookie**. Затем браузер будет автоматически добавлять их в (почти) каждый запрос на тот же домен при помощи заголовка Cookie. Один из наиболее частых случаев использования куки – это аутентификация:

1. При входе на сайт сервер отсылает в ответ HTTP-заголовок Set-Cookie для того, чтобы установить куки со специальным уникальным идентификатором сессии («session identifier»).
2. Во время следующего запроса к этому же домену браузер посылает на сервер HTTP-заголовок Cookie.
3. Таким образом, сервер понимает, кто сделал запрос.

Для правильного форматирования следует использовать встроенную функцию **encodeURIComponent**.

* **path=/mypath**

URL-префикс пути, куки будут доступны для страниц под этим путём. Как правило, указывают в качестве пути корень path=/, чтобы наше куки было доступно на всех страницах сайта.

* **domain=site.com**

Домен, на котором доступны наши куки.

* **e**[**xpires, max-age**](https://learn.javascript.ru/cookie#expires-max-age)

По умолчанию, если куки не имеют ни одного из этих параметров, то они удалятся при закрытии браузера. Такие куки называются сессионными («session cookies»).

Дата должна быть точно в этом формате, во временной зоне GMT. Мы можем использовать date.toUTCString, чтобы получить правильную дату.

* **max-age=3600**

Альтернатива expires, определяет срок действия куки в секундах с текущего момента.

* **secure**

Куки следует передавать только по HTTPS-протоколу. По умолчанию куки, установленные сайтом http://site.com, также будут доступны на сайтеhttps://site.comи наоборот. С настройкойsecure, если куки будет установлено на сайте https://site.com, то оно не будет доступно на том же сайте с протоколом HTTP, как <http://site.com>.

* **[samesite](https://learn.javascript.ru/cookie" \l "samesite)**

Это ещё одна настройка безопасности, применяется для защиты от так называемой XSRF-атаки (межсайтовая подделка запроса). Куки с **samesite=strict**никогда не отправятся, если пользователь пришёл не с этого же сайта. **Samesite=lax** -pежим Lax так же, как и strict, запрещает браузеру отправлять куки, когда запрос происходит не с сайта, но добавляет одно исключение.

Куки с samesite=lax отправляется, если два этих условия верны:

1. *Используются безопасные HTTP-методы* (например, GET, но не POST).

По сути, безопасными считаются методы, которые обычно используются для чтения, но не для записи данных. Они не должны выполнять никаких операций на изменение данных. Переход по ссылке является всегда GET-методом, то есть безопасным.

1. *Операция осуществляет навигацию верхнего уровня* (изменяет URL в адресной строке браузера).

Обычно это так, но если навигация выполняется в <iframe>, то это не верхний уровень.

* **httpOnly**

Эта настройка запрещает любой доступ к куки из JavaScript. Мы не можем видеть такое куки или манипулировать им с помощью document.cookie.

Куки называются **сторонними**, если они размещены с домена, отличающегося от страницы, которую посещает пользователь.

**Token:**

*«Разные способы аутентификации - на основе куки и на основе токенов(чаще всего jwt-токен имеется в виду, так что для подробностей гуглите именно это).  
В первом случае(на основе куки) в общих чертах происходит следующее: после отправки пользователем своих данных(логина и пароля) на сервере создается id сессии, этот айдишник записывается в базу данных и помещается в куки браузера пользователя. Затем при каждом обращении пользователя к серверу айдишник сессии из куки сверяется с айдишником сессии в базе данных, если совпадают - все ок.  
Во втором случае(на основе токенов) происходит следующее: пользователь отправляет логин и пароль, если все норм то в ответ на это сервер возвращает подписанный jwt-токен. На клиенте этот токен куда-нибудь сохраняется(например в локальную базу) после чего в каждый запрос отправленный клиентом добавляется этот самый jwt-токен. Сервер декодит полученный токен и если все норм то запрос обрабатывается. Ну и отмечу, что и в этом случае можно хранить jwt-токен в тех же куках и брать на проверку на оттуда, это уже не суть важно.  
Способы похожи, но все же разные. В первом случае есть так называемое состояние - то есть и сервер и клиент должны хранить айдишник сессии, сервер мониторит/ведет эти сессии в базе данных и т.д. В случае же с jwt-токеном от понятия состояние по сути избавляемся - серверу не важно кто вошел в систему, с помощью какого именно токена и т.д. То есть в данном случае токен самодостаточен и кроме этого токена серверу ничего больше не надо чтобы определить все ок или нет».*

**JSON Web Token (JWT)** — это открытый стандарт для создания токенов доступа, основанный на формате [JSON](https://ru.wikipedia.org/wiki/JSON). Как правило, используется для передачи данных для [аутентификации](https://ru.wikipedia.org/wiki/%D0%90%D1%83%D1%82%D0%B5%D0%BD%D1%82%D0%B8%D1%84%D0%B8%D0%BA%D0%B0%D1%86%D0%B8%D1%8F) в клиент-серверных приложениях. Токены создаются сервером, подписываются секретным ключом и передаются клиенту, который в дальнейшем использует данный токен для подтверждения своей личности.

Токен JWT состоит из трех частей: заголовок (header), полезная нагрузка (payload) и подпись или данные шифрования(signature). Первые два элемента — это JSON объекты определенной структуры. Третий элемент вычисляется на основании первых и зависит от выбранного алгоритма (в случае использования не подписанного JWT может быть опущен). Токены могут быть перекодированы в компактное представление.

* **Access-токен** — это токен, который предоставляет доступ его владельцу к защищенным ресурсам сервера. Обычно он имеет короткий срок жизни и может нести в себе дополнительную информацию, такую как [IP-адрес](https://ru.wikipedia.org/wiki/IP-%D0%B0%D0%B4%D1%80%D0%B5%D1%81) стороны, запрашивающей данный токен.
* **Refresh-токен** — это токен, позволяющий клиентам запрашивать новые access-токены по истечении их времени жизни. Данные токены обычно выдаются на длительный срок.

Как правило, при использовании JSON токенов в клиент-серверных приложениях реализована следующая схема:

1. Клиент проходит [аутентификацию](https://ru.wikipedia.org/wiki/%D0%90%D1%83%D1%82%D0%B5%D0%BD%D1%82%D0%B8%D1%84%D0%B8%D0%BA%D0%B0%D1%86%D0%B8%D1%8F) в приложении (к примеру, с использованием логина и пароля)
2. В случае успешной аутентификации, сервер отправляет клиенту access- и refresh-токены.
3. При дальнейшем обращении к серверу, клиент использует access-токен. Сервер проверяет токен на валидность и предоставляет клиенту доступ к ресурсам
4. В случае, если access-токен становится не валидным, клиент отправляет refresh-токен, в ответ на который сервер предоставляет два обновленных токена.
5. В случае, если refresh-токен становится не валидным, клиент опять должен пройти процесс аутентификации (п. 1).

JSON токены могут храниться в браузере двумя способами: в [DOM-хранилище](https://ru.wikipedia.org/wiki/Web_Storage) или в [куки](https://ru.wikipedia.org/wiki/Cookie" \o "Cookie).

*JWT имеет ряд преимуществ над [куки](https://ru.wikipedia.org/wiki/Cookie" \o "Cookie):*

* При использовании куки сервер должен хранить информацию о выданных сессиях, в то время как использование JWT не требует хранения дополнительных данных о выданных токенах: все, что должен сделать сервер — это проверить подпись.
* Сервер может не заниматься созданием токенов, а предоставить это внешним сервисам.
* В JSON токенах можно хранить дополнительную полезную информацию о пользователях. Как следствие — более высокая производительность. В случае c куки иногда необходимо осуществлять запросы для получения дополнительной информации. При использовании JWT эта информация может быть передана в самом токене.[[5]](https://ru.wikipedia.org/wiki/JSON_Web_Token#cite_note-5)
* JWT делает возможным предоставление одновременного доступа к различным [доменам](https://ru.wikipedia.org/wiki/%D0%94%D0%BE%D0%BC%D0%B5%D0%BD%D0%BD%D0%BE%D0%B5_%D0%B8%D0%BC%D1%8F) и сервисам.