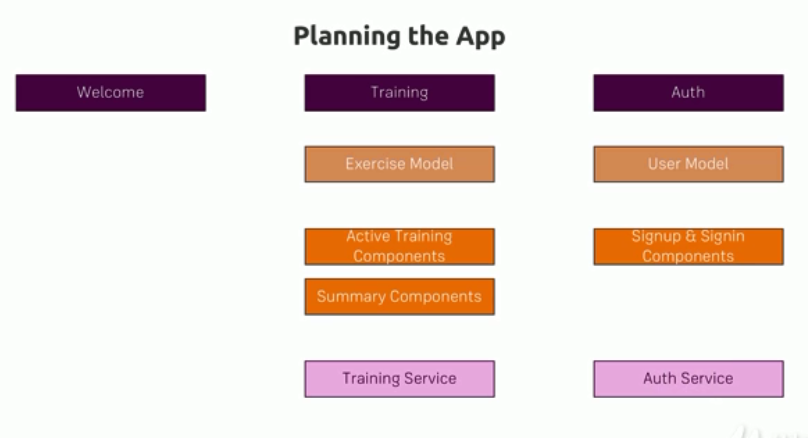
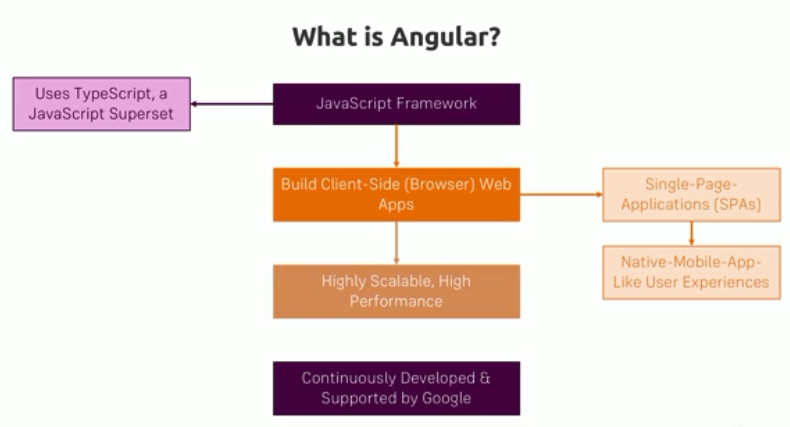
**Angular Material**

# Planning the App



# Angular Refresher

## What is Angular?



* Angular is a JavaScript framework. We build JavaScript client side (in the browser) apps with it to be precise.
* It uses Typescript a JavaScript superset.
* It runs fully in the browser so we can create so-called Single Page Applications where we only download one page from the server and thereafter everything is handled in the browser which feels and looks a lot like native mobile apps.  
  Things happen very fast because we don't need to wait for some of our responses or if we occasionally do we need to wait, we at least wait for a new page but we can show a spinner on the existing page while status fetched in the background.
* Angular framework is highly scalable and offers great performance.
* The core concepts are components and a single page. So we only get one page from the server index.html file and a bunch of javascript files and there after the entire page and what's on it will be managed via javascript which we of course can use to dynamically create and remove elements, we don't write code for that on our own though instead we instruct angler to do so by building so-called components.
* Our whole web site is made up of components.
* So we basically build a bunch of building blocks and tell Angular how to glue them together and Angular will do the rendering part, the communication part and all these fun pieces which we don't want to code on our own.

## Component

* A **component** is just a template where we have HTML code which is enhanced with some special angular syntax like where we may have a click listener and output some content as well as bind to some image property.
* In addition to the template, components are always accompanied by a typescript, (javascript) file where we write some logic for the components for example what should happen up when I click.
* A component is a building block of our app and our angular apps typically consist of many components.
* Annotated with @Component decorator from @angular/core
* The @Component decorator adds metadata which is understood by angular which allows angular to render it to the screen.
* All components go into the template (html) of other components except the root component which goes into the index.html directly as it is the one which needs to be bootstrapped.
* Components communicate with each other via property and event bindings, services.
* Decorator is a typescript feature which kind of manipulates the class behind the scenes. It adds some meta data, we could say.

## How an Angular App Starts and Works

1. During build process CLI will inject script imports into the raw index.html file which just has <app-root>.
2. This <app-root> is the element where our Angular app is rendered during runtime.
3. Angular somehow needs to know that it should render itself though. And that is instructed in the main.ts file.
4. main.ts is the starting file which is first executed when our angular app starts. It's the absolutely first file that runs. It is of course typescript and therefore it will be compiled to javascript during build time.

platformBrowserDynamic().bootstrapModule(AppModule)

  .catch(err => console.error(err));

1. As you can see, the module mentioned in the .bootstrapModule() is bootstrapped, i.e. started on its own, without any other help. The bootstrapModule() function will instruct the Angular to start itself.
2. Here the AppModule is like a root definition file that defines all the pieces our Angular app is made up of.
3. Each Angular **module** is decorated with NgModule decorator which has configurations like declarations (where we declare all components in that module), imports (where other imported modules are declared), providers (wherein we put anything that we want to inject into any place of the app e.g. declare services used in the components of this module).
4. In addition to these configurations, we have to declare a special property in the root module which is called bootstrap. E.g. bootstrap: [AppComponent]
5. The **bootstrap** property will inform Angular that the AppComponent is the root component, i.e. the main component of our application. So Angular will render the content of this AppComponent in the place of <app-root> tag in the index.html. (<app-root> is the selector mentioned for AppComponent typescript file.)
6. Also there is one more least used property called entryComponents, It takes an array where we have to add all components that are never instantiated by using their selector in our template nor by routing. With entryComponents, we tell angular to be prepared to use it.

## String interpolation

* Allows us to refer to a property of our typescript class and output its content in the template.
* Syntax: {{ propertyName }}

## Event Binding

* To listen to the events on an element of component.
* Syntax: (eventName)=”eventHandler()”
* Syntax: (eventName)=”short inline statement”

## Property Binding

* To dynamically assign values to the properties on an HTML element (normal HTML element or Angular component selector)
* Syntax: [ propertyName ] = “tsVariable”  
  e.g. <button [disabled]=”isDisabled”>Click Me </button>

## Two-way data binding

* Typically used with input elements.
* Syntax: [()]
* E.g. <input type=”text” [(ngModel)]=”productName”>  
  This now ensures that the value of this input is the value of the productName variable but that if we set any other value in that input, the productName is updated to whatever we entered.

## Directives

* Allow us to manipulate the content whereas click listeners and content output instructions only change in minor pieces of the app but with directives like ngFor we can dynamically loop for our data and then render whole parts of the page based on that.
* Structural directives are directives that really change the structure of our page by adding and removing elements. E.g. \*ngIf, \*ngFor – the directive is just the ngFor and the \* tells angular that this is the structural directive that will change the document.

## Custom Property binding using @Input directive

* This directive allows cross-component communication in our app.
* We put @Input decorator in front of the property in one component which should be settable from outside of that component.
* E.g.
  + In the ProductComponent (<app-product>), we write  
    @Input() productName: string;
  + So now this property can be set from other component e.g. ProductsContainerComponent using our usual property binding syntax like this –   
    <app-product [productName]=”someValue”></app-product>

## Listening to events from other components using EventEmitter and @Output decorator

* We can listen to events from other component to our component using @Output decorator. For that, we need to declare an EventEmitter in our component and decorate is with @Output decorator. This will enable us to listen to the events from outside of this component.
* @Output decorator is used to pass our custom events to the outside.
* E.g.
  + In ProductComponent, we write this –

@Output productClicked = new EventEmitter();

* + Next, we need to decide on what condition, we should emit an event. Let’s say we want to emit event on click of the container div in the ProductComponent. So –

<div (click)=”onClicked()”>…….</div>

* + Now we need to write handler for this event, wherein we will just emit this event so that it can be listened from outside of this component.

onClicked() { this.productClicked.emit(); }

* + Finally we need to listen to above emitted event in the other component. E.g. ProductsContainerComponent like this –

<app-product (productClicked)=”handler()” [productName]=”someValue”></app-product>

* + Here the handler() function will be written in the other component (ProductsContainerComponent).

## Services

* Services are typescript classes which can be reached from any component, can be injected into any component or actually also into other services.
* A service holds some functionality which you typically want to share across multiple components. This could be data what you want to use from other components. It could be utility functions like logging something or sending HTTP requests.
* We can then use this service in other components simply by injecting into the component constructor.  
  e.g. In ProductsContainerComponent  
  constructor(private productService: ProductService)
* Now in order for other components to be injected with the service, the service needs to be declared in the providers array of that module.  
  e.g. In AppModule,  
  providers: [ProductService]
* With this, Angular is able to resolve this dependency and actually create a new instance of the service and provides/injects into the component constructor.

## Routing

* Routing simply means that our Angular app can have multiple pages even though only one page is sent from the server.
* The other pages are really just simulated.
* Angular just re-renders the entire page to give the illusion of having loaded a new page even though everything was done by javascript which is of course faster than fetching a new page.
* It is a common convention to put routing into its own module.   
  E.g. app-routing.module.ts
* In this file, we can define our routes. E.g.

const appRoutes : Routes = [

    {path: '', component: HomeComponent},

    {path: products, component: ProductsComponent, children: [

      {path: ':id/:name', component: ProductComponent}

    ]},

    {path: '\*\*', redirectTo:"/not-found"},

];

* Now we just need to inform Angular about these routes. We can do this by using RouterModule.

@NgModule({

    imports: [ RouterModule.forRoot(appRoutes) ],

    exports: [ RouterModule ]

})

export class AppRoutingModule {

}

* Every module works stand alone and doesn't spoil its information to other modules. So is we need to use a module in some other module, then the source module needs to export the features its module and the target module needs to import source module. So we need to add another key here to the NgModule definition which is ‘exports’. This ‘exports’ parameter takes an array of all the features from that module like to router module in this case that we want to expose to other modules.
* Now Angular is aware of our routes. We now just need to tell it where to render them.

We can do so by special syntax: <router-outlet></router-outlet>

* router-outlet is a directive coming from angular which marks the place in your document where you want to render a route. Content above or below that will not be replaced by the router component.
* Now we just need to have a way of navigating around. We can do so using routerLink directive. E.g.

<a routerLink="/">Home</a>

<a routerLink="/products">Products</a>

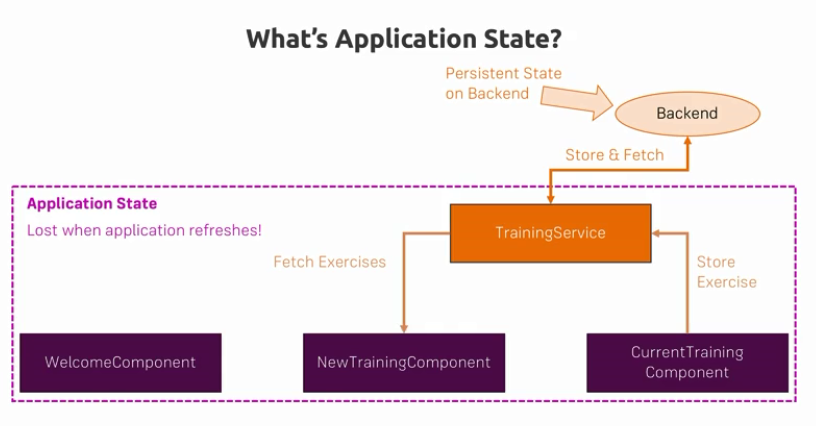
## Modules

* We can divide our App into submodules.
* We should create modules for –
  + Feature modules for each feature (declarations and imports)
  + Core module to put all the features of the app module into that.
  + Shared module to keep all the modules which are being used in other more than one modules (mostly in feature modules). So a Shared module will only have imports and **exports** too.
* **Important:** By default a module keeps its imports and declarations to itself. But if you add **exports,** it actually shares it with any other modules that import this module.
* We should always keep application wide services into the main App module only.
* Keep in mind is that just because we import something into the app.moduel, this does not make it available in other modules even if those modules are kind of child modules to app.module. That is, the modules imported into app.module are not passed down or exchanged to other imported modules in the app.module.
* Each module works standalone.
* Now if you're worried about increasing your bundle size by importing some modules in more than one other modules, then this is not the case. Angular manages this cleverly behind the scenes.

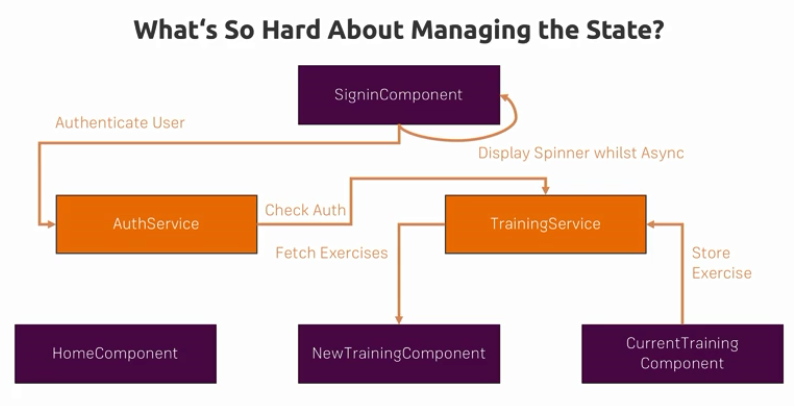
## Application State Management

* NgRX is Angular’s Redux based Application State management solution.

### What is Application State?

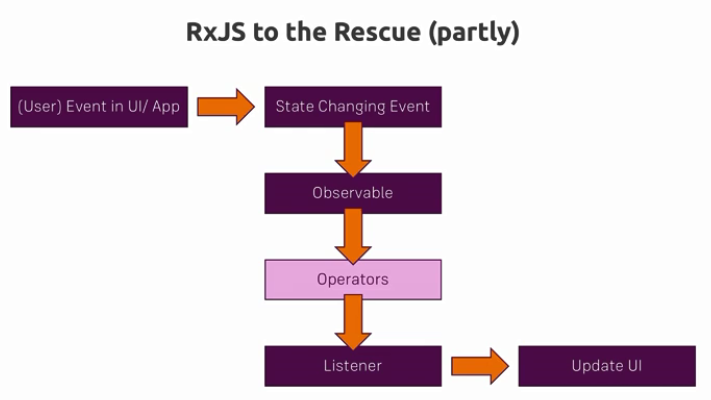


### What is So Hard About Managing the State?



* You can end up with so many connections that you want to have a central state management system.

### RxJS to the Rescue (Partly)

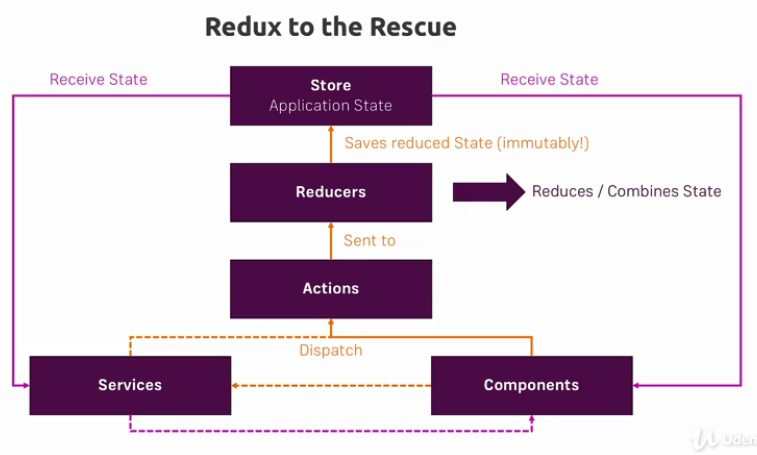


* The first step to getting a predictable and reliable state management is to use our RxJS, but there are some problem with that.

#### Issues with using RxJS

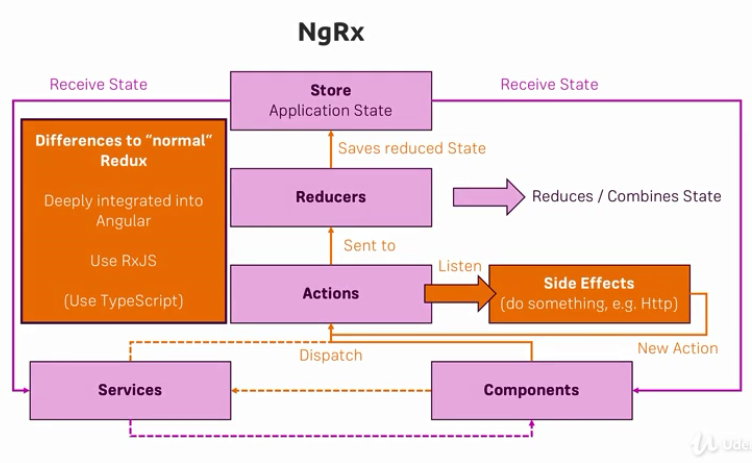
* The state can be updated from anywhere.
  + So in this pattern we can trigger our listeners from everywhere.
* The state is maybe mutable.
* Handling side effects is unclear.
  + Where do we send HTTP requests from?
  + Where do we use AngelFire? Do we do that in a service? Do we do that directly in a component?
* With above problems we have, but we don’t have any specific patterns which we can enforce in our App. That is where Redux comes to the rescue.

### Redux



* Redux is all about having a central store where we store the entire application state. So things like are we loading something, did we load exercises and the array of loaded exercises.
* Then we have our services and components that interact with each other and that receive a state from that central store so that we really have that centralized approach.
* Then we also have **actions** if we want to change the store. We don't directly manipulate it, instead we dispatch actions that reach so-called **reducers**.
* The reducer then takes that action and a potential payload and reduces it to a value which then is stored in that central store in the immutable way (So it doesn't alter the old store. It simply takes the old store and replaces a part of it with a brand new value.)
* This approach is the Redux and we can implement it in Angular and we typically do that with NgRX pattern.

### NgRX

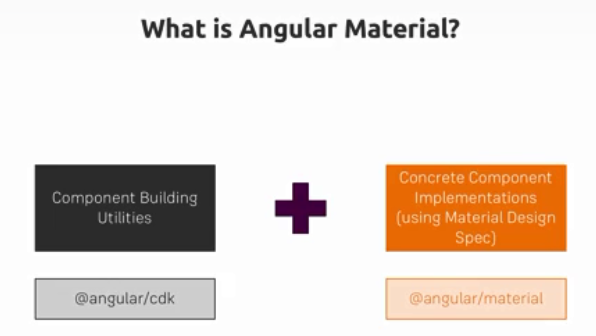


* So we can implement Redux in Angular and we typically do that with NgRX pattern.
* NgRX follows Redux approach but the differences are that it is deeply integrated into Angular – from the naming conventions, by the fact that it uses typescript and that it uses our RxJS.
* And then we have **actions** and also the so-called **side effects**.
* Side effects are things like –
  + I want to change the route.
  + I want to make an HTTP request.
  + So basically things which are not stored in the store (We don't store the route there. We don't store the HTTP request. We do store the results of the HTTP request, but not the HTTP request itself).

# Angular Material

## Getting Started

* Angular material is a third party package which we can use in Angular projects.
* In its core, it's one thing – An angular components suite. It means it's a collection of pre-built and styled angular component. Here we get a package of a lot of components we commonly need in angular apps.
* These components follow the material design spec which is the design spec created by Google which they use in their Android operating system and which is also really popular in the web.
* Angular Material can be thought of as two packages.
  + Component Building Utilities
  + Concrete Component Implementations using that material design spec.
* We will always use both when working with angular material.



### Component Building Utilities

* This feature set is bundled in a package called @angular/cdk. CDK stands for component development kit.
* This does not include any angular material styling.
* This does not include the concrete component implementations.
* This has a lot of utility and helper methods and classes for example for rendering overlays for positioning elements and some raw unstyled components like for example a data table which is a very complex component behind the scenes.
* An Important thing – we can use this CDK as a basis for your own component kit if we want to. So as an alternative to angular material, if we want to create our own components with our own styling.
* <https://material.angular.io/cdk/categories>

### Concrete Component Implementations using that material design spec.

* This is bundled under @angular/material package.
* This is the package which gives us access to all these Angular material components which are pre-styled and which behind the scenes are built up on that CDK package, which is why we need to add @angular/cdk dependency.
* <https://material.angular.io/components/categories>

## Adding Angular Material to a Project

* <https://material.angular.io/guide/getting-started>
* An important note – we don't install the angular material package globally on our machine. It's not a tool. It's not a global program on our computer. It's a per project dependency which we can drop into any angular project. It does of course means that we need an angular project first.
* So here are the steps –

1. Install Angular CLI globally, if we don’t have one

* npm install –g @angular/cli

1. Create an Angular project

* ng new fitness-tracker
* It will install all the dependencies of the project both development dependencies like webpack which bundles and optimizes our code or of course production dependencies like the angular packages

1. Now navigate to this created project. E.g. fitness-tracker
2. Angular Material is just another dependency of our project.
   * We can install it either by very handy Angular CLI Schematics –
     + ng add @angular/material
3. OR follow below manual steps – These manual steps do exactly what above handy Angular CLI Schematic command does.
   * 1) Install both Angular CDK and Angular Material as –

npm install --save @angular/material @angular/cdk

CDK is only needed as a dependency of angular material.

* + 2) Now install Angular Animations package, because @angular/material uses that package. Before installing this, just double check the package.json if this animations package is already installed because in newer versions of Angular, it is already installed.

npm install --save @angular/animations

* + 3) In Angular, we have to explicitly add all the features we want to use so that our code can be optimized.
  + Import BrowserAnimationsModule from @angular/platform-browser/animations package into our app module and this BrowserAnimationsModule in the ‘imports’ of AppModule.
  + 4) Now import the Angular Material component modules we plan on using. We can either add those imports directly into the main AppModule or outsource it by creating a new module and import this newly created module in to the AppModule, which is a better practice.

The idea simply is that we can manage our material imports in a separate file but still have access to them in the entire app.

* + 5) Include a theme –
    - Open style.css and add import for the Angular theme as –  
      @import "~@angular/material/prebuilt-themes/indigo-pink.css";
    - This is important angular material components use a theme, a theme is just a color combination of a primary colour, accent colour and a warning colour in its core.
    - By adding the import in style.css, all angular material components will automatically use this color combination/theme.
  + 6) Add Gesture Support –
    - This is required to be able to use gestures on mobile devices like for example sliding with the finger.
    - For that we need to use an external library which is used behind the scenes by angular material – hammerjs
    - To install it, run below command

npm install --save hammerjs

* + - Now import it in the main.ts file

import 'hammerjs';

* + - This will pull the dependency into our final bundle so that it is available in the code we ship to the user.
  + 7) Add Angular Material icons –
    - Add below link tag in the index.html file

<link href="https://fonts.googleapis.com/icon?family=Material+Icons" rel="stylesheet">

* + - Now we can use the Angular Material icons in our app.
* In Angular Material, every component has its own module which you need to import, to be able to use that component like for example the button component got its own module the checkbox to dropdown and so on.
* The reason for this very simple.
  + By putting everything into its own module our code can be optimized and be made as small as possible.
  + We're not importing all components of angular material by default but only what we need. Which means the code we ship to the user in the end is also only what our app needs. And therefore as small as possible.
* Angular material components use a theme.
  + A theme is just a color combination of a primary colour, accent colour and a warning colour in its core.
  + Now angular material ships with a couple of pre-built themes.
  + We can access these pre-built themes by navigating to node\_modules\@angular\material\prebuilt-themes
* With Angular Material components, we can add components to our page. But in order to control layout of our page using Angular material, we can use Flexbox or Grid.
* Angular flex layout package
  + This is a package that uses CSS Flexbox behind the scenes to allow us to position your components and elements with a couple of nice directives.
  + This package just abstracts away the CSS styles and we use convenient to use directives to assign those CSS classes basically. That’s what happens behind the scenes.
  + Additionally it gives us a responsive API which simply means that we can combine these directives with certain responsive directive additions where we define a certain rule should only apply to certain screen sizes.
  + To install angular flex layout package –

npm install @angular/flex-layout –save

* + Please visit this link to see latest information about installation.

<https://github.com/angular/flex-layout/wiki/Using-Angular-CLI>

* <mat-error>, <mat-hint> are special functionalities which are part of mat-form-field and these are so powerful that with a few simple to add components we turn your normal input into a nice input, providing a good user experience with nice looking hints and errors without us writing any CSS or javascript code to control all of that.
* <mat-form-field> provides lot of out of the box working functionalities to create beautiful forms which provide great user experiences.
* <mat-form-field> is designed to work with only few elements (input elements basically) like input, select, mat-select, mat-hint, mat-error, mat-datepicker, etc.
* Angular Date picker –
  + We can very easily add date validations into the Angular datepicker.
  + E.g. allow only certain date range, minimum allowed date, maximum allowed date, etc.
  + In order to do so, we can use max, min properties of native input element itself.
  + E.g. If we want to ensure that the user is at least 18 years old, we can do so using the max property to the input. This will add the max value validator to the input and if we set this to a date (by binding to mat-datepicker) this will automatically be recognized by the date picker and taken into account.
  + This is very powerful with so much ease to implement.
* Toolbar –
  + A toolbar is a menu bar at the top.
  + Toolbar is more than that though. We can add it anywhere in any container in any place of our application.
  + And in the end it's just a block with a background color with some items on it.
  + It uses flex blocks internally to distribute these items.
* We can add color property to only few Angular material components like button, toolbar, select, etc.
* fxHide.xs means only hide this element on extra small screens. (there are many other options. Checkout <https://github.com/angular/flex-layout/wiki/Responsive-API> )
* fxShow.xs means only show this element on extra small screens. (there are many other options. Checkout <https://github.com/angular/flex-layout/wiki/Responsive-API> )
* Progress Spinner and Progress Bars are available in determinate and indeterminate modes. Indeterminate modes are the right choice if you don't know how long something is going to take and you want to show some progress indicator the determinant question is a good choice.
* Angular Material Dialog is a special material component. It's not added to the template instead it's invoked programmatically.
* Angular Snackbar –
  + MatSnackBar is a service for displaying snack-bar notifications.
  + This is very useful in case we want to show some notifications like errors, success, etc.
  + MatSnackBar is very much configurable. Please see official docs for details.
* Angular Material Data Table –
  + It is an extremely powerful component which is easy to use.
  + It has nice layouts and we can add features like pagination, sorting, filter.
  + Sorting Data table –
    - Import MatSortModule in your module.
    - Add matSort directive to the mat-table element, which makes this table sortable.
    - By default, no column is sortable. For make a column sortable, we have add special directive mat-sort-header to the mat-header-cell
    - Get access to the MatSource using @ViewChild in the .ts file. With this, we get the underlying sorting setup in the Angular material.

@ViewChild(MatSort) sort: MatSort;

ngAfterViewInit() {

this.dataSource.sort = this.sort;

}

* + Filtering Data table –
    - Adding filter to the Data Table is much earlier.
    - First of all, add in input textbox outside mat-table which we will use for filtering. Add keyup event listener
    - For this keyup event listener handler method, just set this
    - E.g.

doFilter(filterValue: string) {

this.dataSource.filter = filterValue.trim().toLowerCase();

}

* + - This is important to trim and convert the filter string to lowercase, because in the end Angular Material data source will concatenate all the values of a row, trim them and converts to lowercase. And then search the filter string into this long concatenated-trimmed-lowercase string. And based on that shows the filtered date on the table.
    - For example, the data object {id: 123, name: 'Mr. Smith', favoriteColor: 'blue'} will be reduced to 123mr. smithblue. If your filter string was blue then it would be considered a match because it is contained in the reduced string, and the row would be displayed in the table.
    - We can override its default filtering mechanism though. Refer Angular Material guide – <https://material.angular.io/components/table/overview#filtering>
  + Pagination –
    - Import MatPaginatorModule
    - We can add pagination to the mat data table using the mat-paginator Angular material element.
    - mat-paginator should be placed outside ma-table in the template.

E.g. <mat-paginator [pageSize]="1" [pageSizeOptions]="[1, 5, 10, 20]"></mat-paginator>

* + - Add @ViewChild for MatPaginator in order to get access to this paginator

@ViewChild(MatPaginator) paginator: MatPaginator;

* + - Plug this paginator to the dataSource in the ngAfterViewInit()

E.g.   
ngAfterViewInit() {

// we have to add sort and pagination only after view is rendered

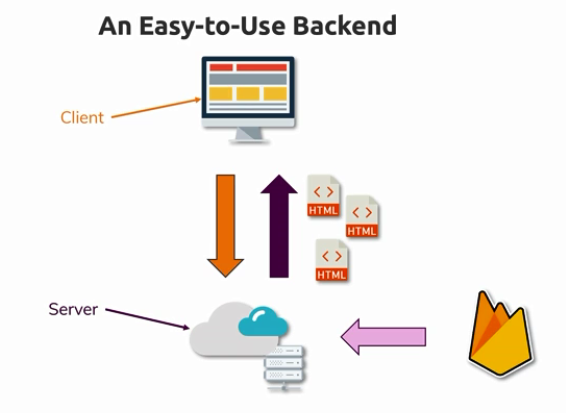
this.dataSource.paginator = this.paginator;

}

# Firebase and AngularFire

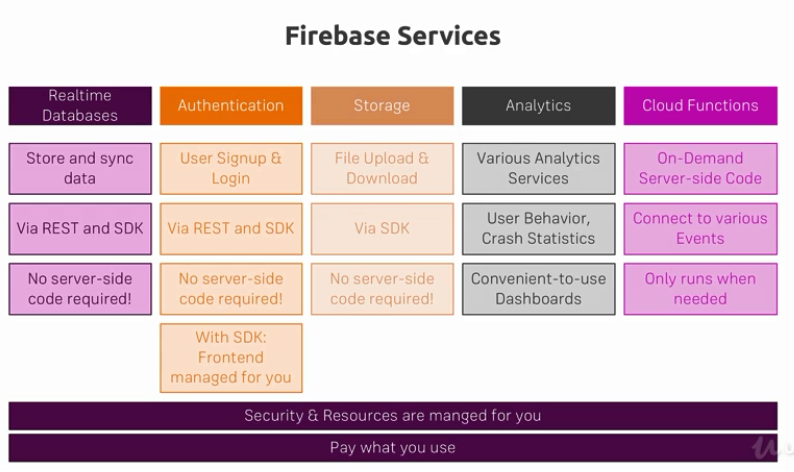
* Firebase is used as a backend to store app data.
* AngularFire is a third party package used to connect to the Firebase from Angular App.

## Firebase



* Firebase is used as a backend to store app data.
* It can be used as a hosting provider to host the regular app but more than that it can be used as a back end to which we connect to store data fetch data run all kinds of calculations or also very important authenticate users.
* Firebase is a popular backend for mobile apps.

### Services offered by Firebase



#### Real Time databases

* Firebase has services connected to real time databases. Real time simply means that the databases that we can use can actually be used in a way that uses Web sockets behind the scenes so that we can get live updates into our page into our angular app without the need to send subsequent rest API requests.
* Firebase offer database solutions using no sequel databases.
* Firestorm is a schema less database. That means we don’t need to have same attributes to each document.
* These databases are used to store and sync data. We can reach them using REST API or official firebase SDK or Angular fire.
* And we don't need to write any service code, they work out the box. We don't need to create rest endpoints. We don't need to write any code to validate data. We can almost (of course there is security layer in between) directly communicate with the database.

#### Authentication Service

* Firebase has an Authentication service. With this service, we can easily add user, sign up and log in.
* Here also we can use these services using REST API or official firebase SDK or with Angular fire without writing any server side code.
* And if we use the SDK then even on the front end we get a lot of utility functions things like storing the token and so on.

#### Storage Services

* Firebase also offers storage services so files storage services which help to a file upload, download, also via firebase SDK here too. No server side code is required.

#### Analytics Services

* Additionally firebase has analytics services where we can integrate analytics into our apps.
* Firebase is a popular backend for mobile apps. So for Android or iOS apps. There we can integrate even a bit more analytics services than web apps.

#### Cloud Functions

* Cloud functions as a service where we can write our own code and execute on demand by attaching it to certain event sources.

#### Note

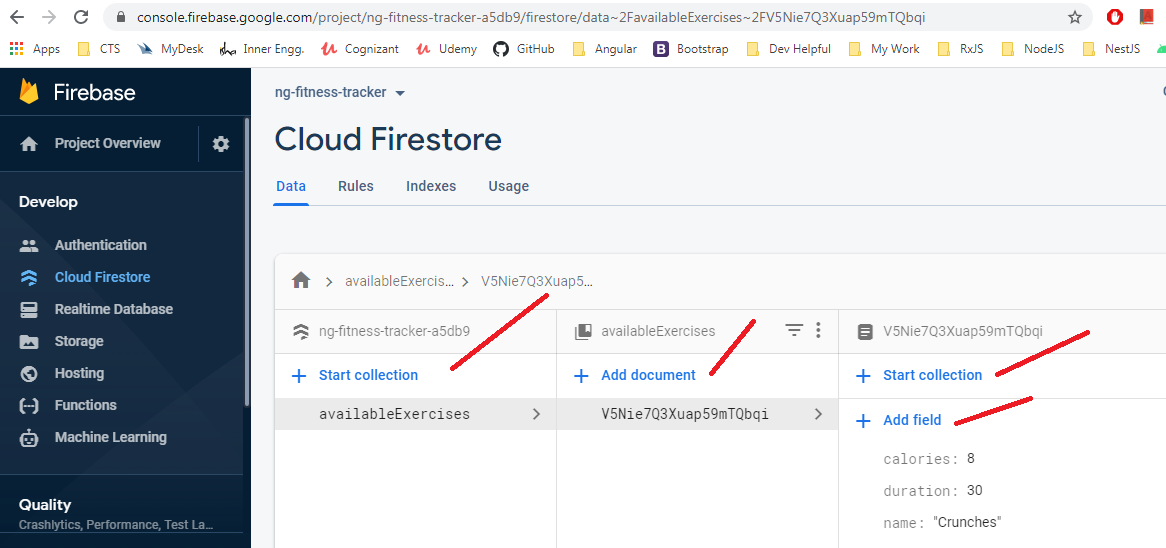
* The cool thing about firebase is all these services is not just the service offerings but that all the resources and services are managed for us.
* We don't need to provision, spin up, manage and secure servers. So that's all done for us by Firebase. We just use the services and we’re charged on a pay what you use basis.

#### Could Firestore

* Cloud Firestore is kind of advanced version of real time database service from Firebase.
* Three terms to know while working with Cloud Firestore – **Collections** which we could compare to folders on our operating system, **Documents** which we can well compare to documents on our operating system which are organized in folders or collections in this case and **Data** inside a document (mostly JSON data).
* In the Firestore, we store documents in the end. A document could be the data about one single user. Multiple users would have multiple documents.
* Now documents are part of collections. You must have documents in a collection.
* Remember – We can also have collections which are nested, though we can't have a collection inside a collection but as a child of a document. E.g. we can have a collection called ‘rooms’, inside which we can have 2 documents as roomA and roomB and inside roomA document, we may have subcollection say ‘messages’.
* So we always have this alternating pattern of collection - document - collection - document.
* More details at – <https://firebase.google.com/docs/firestore?authuser=0>

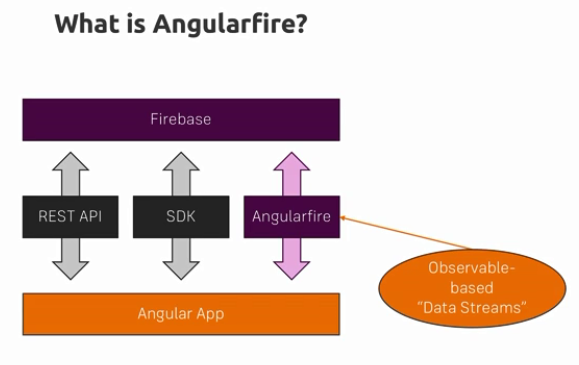
## Adding Firebase to your App

1. We need to have Google Account in order to use Firebase.
2. Open Firebase console in browser – <https://console.firebase.google.com/>
3. Click ‘Add project’ and give some proper name.
4. We this, our project is setup and now we just need to configure services for our project like Authentication, Database Storage.
5. For Storage Services we can either use ‘Realtime Database’ or latest ‘Cloud Firestore’.
6. Let’s use Cloud Firestore –
   * Click on Cloud Firestore
   * Create Database
   * Initially Select “test mode” as true (this will help us in development mode), later once Authentication is implemented, then we can use the “strict mode”.
   * Now Add Collection, Document and Data for the documents



# Angular Fire

* If we have firebase and angular app, then angular fire acts as a bridge between these two things.
* There are multiple solutions to connect Angular app to Firebase like REST API, or Official Firebase SDK (which is a JavaScript package) or Angular Fire.

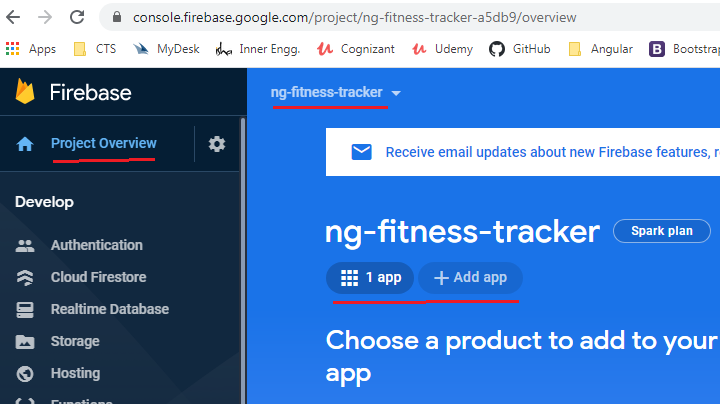


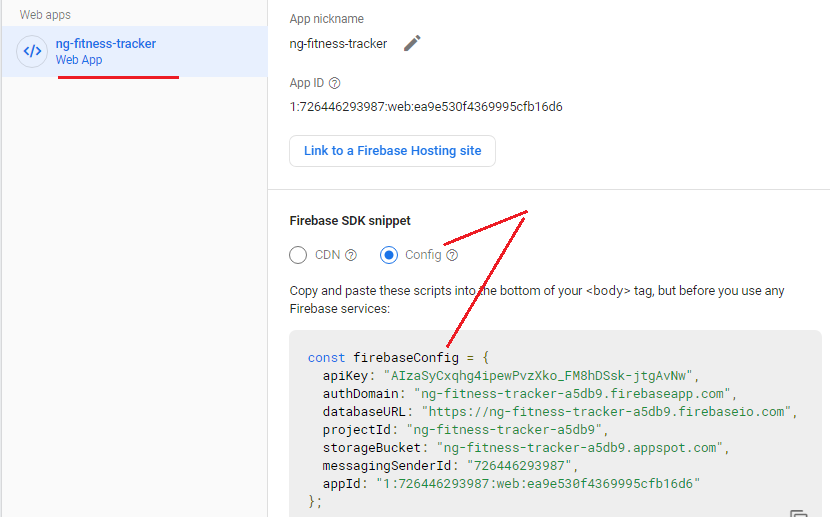
* Angular Fire is a third party package which actually also provides us all the features that the official SDK gives us. But in a manner that embraces observables and nicely integrates into Angular.
* We can only use it in Angular apps. We can't use it in other frameworks.
* In Angular it integrates really nicely and it makes using firebase a breeze.
* Important part is that Angular Fire embraces observable based nature which is absolutely in line with Angular’s strategy of also using observable wherever possible.

## Installing AngularFire

* Follow this link to install and integrate AngulareFire with Firebase <https://github.com/angular/angularfire>
* Here is quick summary –

1. Install via schematic ng add @angular/fire
2. In the Firebase Console (https://console.firebase.google.com/) , register your Web App for given project by navigating through Project Overview -> Add App
   1. We can add Web App, Android App, or iOS app.



1. Once you add you add an app, you will see firebase configuration for that app. Something like this – 
2. Now depending on the framework, we need to add this configuration in our Application Code. For Angular, copy-paste this firebaseConfig into the environment.ts file.

export const environment = {

firebaseConfig: {

apiKey: "AIzaSyCxqhg4ipewPvzXko\_FM8hDSsk-jtgAvNw",

authDomain: "ng-fitness-tracker-a5db9.firebaseapp.com",

databaseURL: "https://ng-fitness-tracker-a5db9.firebaseio.com",

projectId: "ng-fitness-tracker-a5db9",

storageBucket: "ng-fitness-tracker-a5db9.appspot.com",

messagingSenderId: "726446293987",

appId: "1:726446293987:web:ea9e530f4369995cfb16d6"

}

};

1. Next step is to do general setup – Initialize Angular Fire with our firebase data for our Angular app.

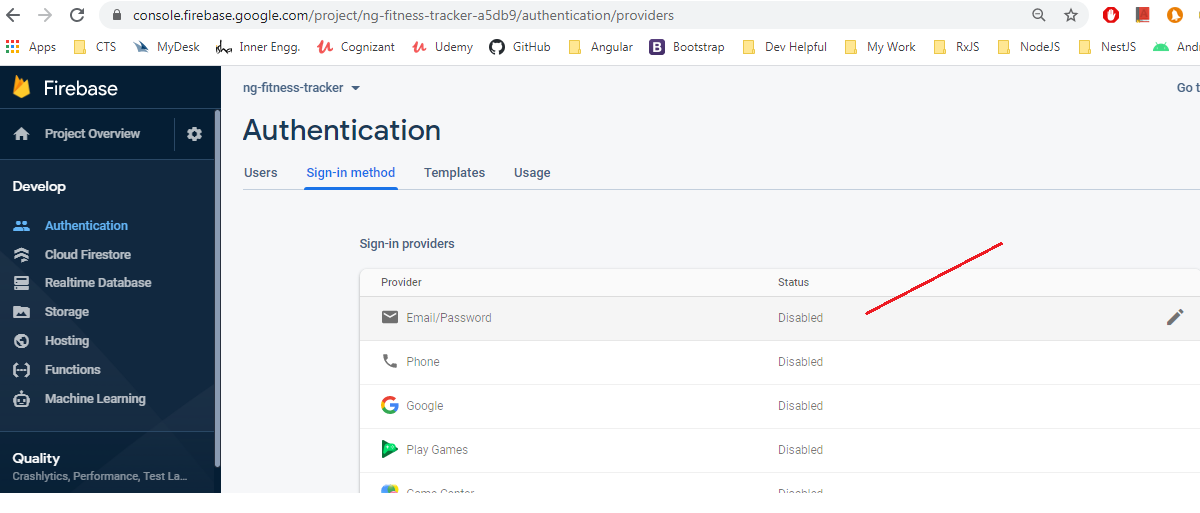
AngularFireModule.initializeApp(environment.firebaseConfig)

1. **To enable Cloud Firestore services**, import AngularFirestoreModule into your app.module.ts file.
2. That’s it! Our configuration is done, that is we are ready to use the Cloud Firestore in our code now.
3. To use Firestrore in our components, inject AngularFirestore into that component constructor.

For e.g. to fetch some data, we can use   
this.exercises$ = this.firestoreDB.collection('availableExercises').valueChanges();

Note: valueChanges() does not give the ID of the document because ID is treated as meta data in Firestore. To get ‘data’ along with meta-deta, we need to use snapshotChanges()

1. **To enable Firebase Authentication services**,
   1. From Firebase console, enable Authentication Services like –

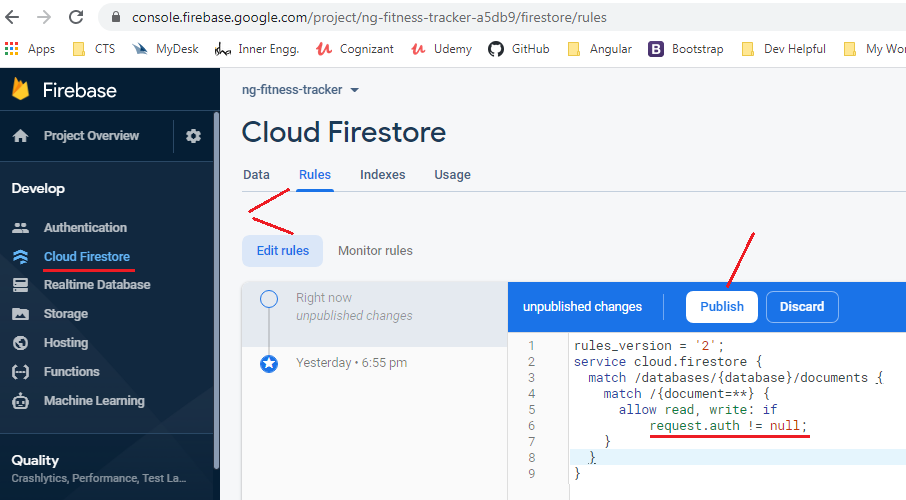


* 1. import AngularFireAuthModule into your app.module.ts file.
  2. Now we can inject AngularFireAuth class and use it for signup/login users.  
     AngularFireAuth has many utility methods to different types of signups and logins.

1. **To enable only Authenticated read/write access to Firebase database (Cloud Firestore),** we need to add the ‘rule’ in the Firebase console as –

allow read, write: if

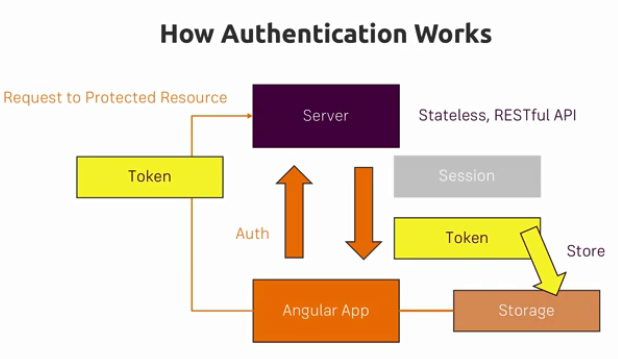
**request.auth != null;**



1. With this setting, now any unauthorized access to our Firestore is not possible. We don’t have to do any code changes in our Angular app. AngularFire and Firebase does everything (managing the token) to protect the store behind the scenes. Read below section for more details – [How Authentication works in SPAs (like Angular)](#_How_Authentication_works)

# How Authentication works in SPAs (like Angular)

* When we authenticate or when we login, we sent the credentials to the server and the credentials are validated and we get back a response that we are authenticated.
* Now in the traditional web app where we have multiple pages, we would use a session to store it. The session would be stored on the server and on the client we would get a cookie or sessionid.
* Now in single page applications, our back ends are stateless (like REST API). Any Angular app is stateless because we always use Ajax HTTP requests behind the scenes as only got one single page and we don't request new pages in between. So therefore the session based approach doesn't work with SPAs. So here we use token instead, to be specific JSON Web Token (JWT).
* JWT is essentially a long string that encodes (not encrypts) some data about our authentication status, data that can't be fiddled with. Because if we would fiddle, then the token would be detected as manipulated on the server and would be invalid.
* So in SPAs, when we send credentials, we get this token from the server after successful authentication. We then should store this token on the front end that is, in the browser storage place like localstorage.
* Now whenever we want to access some protected resource on the server, so let's say our database, we will attach that token to the request.
* Now as we know the token is structured in a way that the server can validate whether it's still a valid token (the one which was sent by the server) or manipulated one.
* So if the token is still the original token and we still are logged in (so the token is still valid because the token will also expire after some time), then we get access. Otherwise we don't.
* This is how authentication works in SPAs.



### JWT Token management in Firebase

* As described above, the JWT token is a crucial part of Authentication.
* The cool thing about Firebase and AngularFire is that we already get this token and we can also do the whole management of this token through firebase.
  + So with Firebase and AngularFire, we don't have to store and extract it manually.
  + We don't have to worry about its expiration because firebase behind the scenes will always give us a fresh token with every request we send. So we never have to worry about this.
  + And if we log out, it will destroy the tokened for us. So that's all managed for us.

# Tips and Tricks

* AngularFire is a third party package that makes connecting to firebase a breeze and establishes real time connections to the database.
* We need development server to run Angular because just double clicking the index.html won't do the trick because double clicking the file won't actually use HTTP to serve the page and therefore some features just won't work.
* ngFor directive is from BrowserModule.
* For [(ngModel)] to work, we need to import FormsModule.
* CDK stands for component development kit.
* <mat-error> overrides <mat-hint>
* ng-container is a default angular component which we can use to group things.