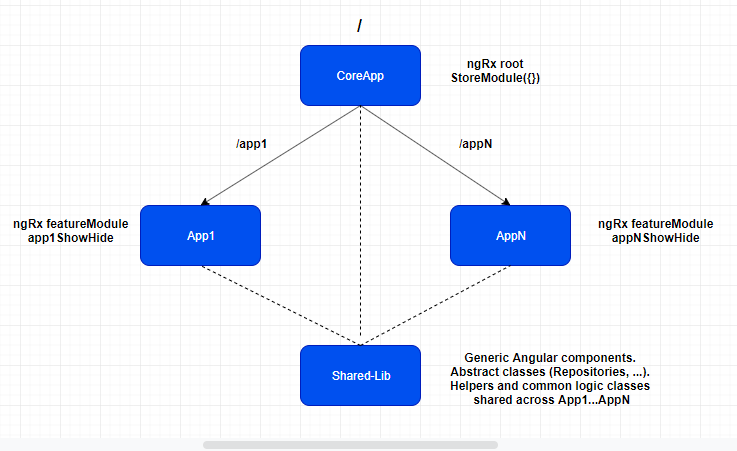
**ARCHITECTURE PROPOSAL**

**The Goal**

1. We will have one **coreApp** called ***ONCE*** (developed later)
2. We have N sub-applications (***StoreLite***, ***Maintenance***, ***Reports***, …)
3. **App1...AppN** can be lazy loaded as routes in **ONCE** **coreApp**. Also **App1...AppN** can be started and developed independently and we don't need **coreApp** **ONCE** to run them.
4. Each **App1...AppN** use NgRx state management.
5. We will have a shared library **shared-lib** with the generic components, abstractions and classes common to all **App1...AppN** applications. It can be built and used in any of apps (**ONCE**, **App1...AppN**).

A simple project diagram looks like this:



**Create Workspace**

The concept was introduced in Angular 7, and the CLI also received a new flag:

--create-application



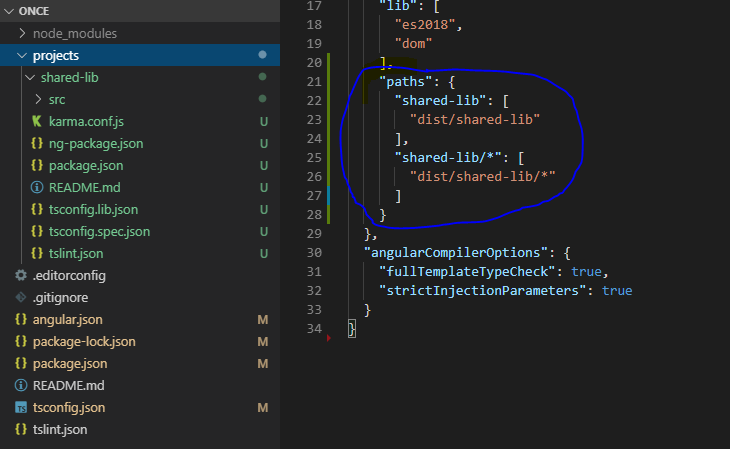
The command will generate a new workspace for Angular application and libraries with no default application, allowing now to add libraries and applications in the workspace and name them however required.

**Create library**

Libraries can be added via the CLI since version 6, but by using the workspace concept it makes things much cleaner.



By using this command the CLI will create a new directory inside the workspace called projects. This where the library **shared-lib** is located and it's also the place where future applications will be added. It also extends the paths configuration in the root tsconfig.json:



This allows to use the library inside the Workspace as if it is loaded from npm. It requires the library to be build inside the dist directory, which can be done by using the CLI:



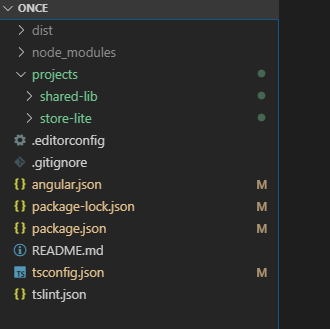
The compiled library will be located inside the dist folder of the wokspace. It contains the code that will be loaded inside the workspace (this is also the code that can be published on NPM).

**Create feature application**

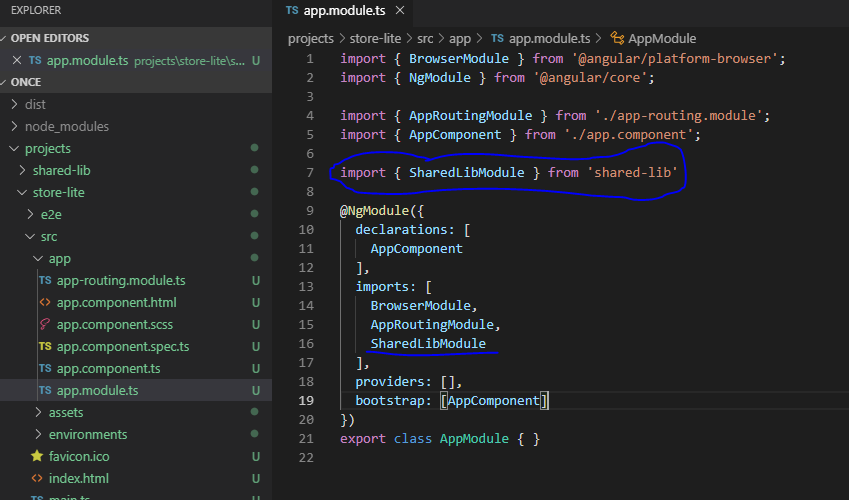
We are going to create **StoreLite** feature application:

****

By using this approach the application **store-lite** will be added also inside the projects directory:



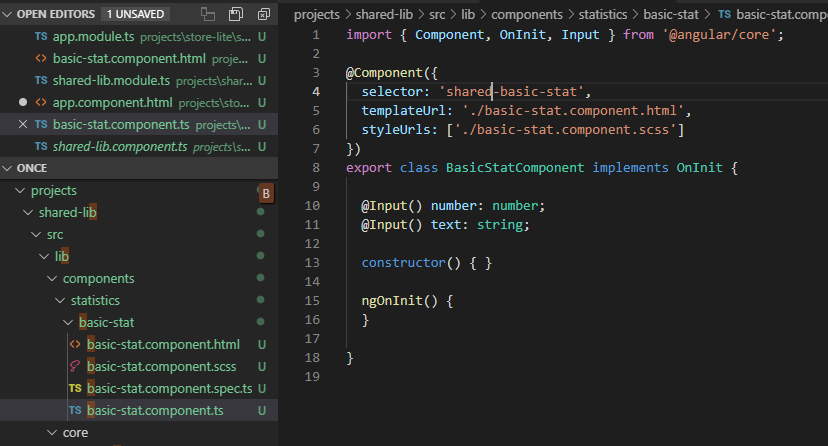
Now you can go to the app.module.ts of the **store-lite** app and load the library:

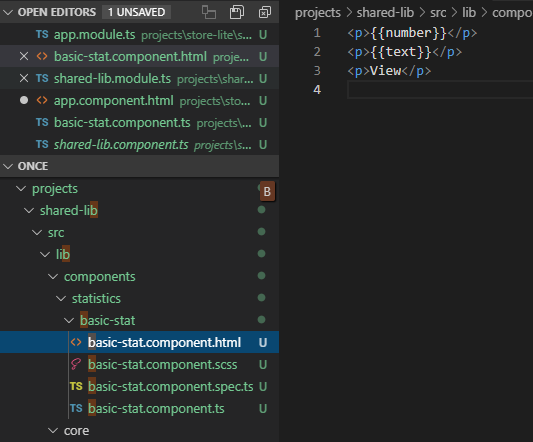


**Create Generic Component**

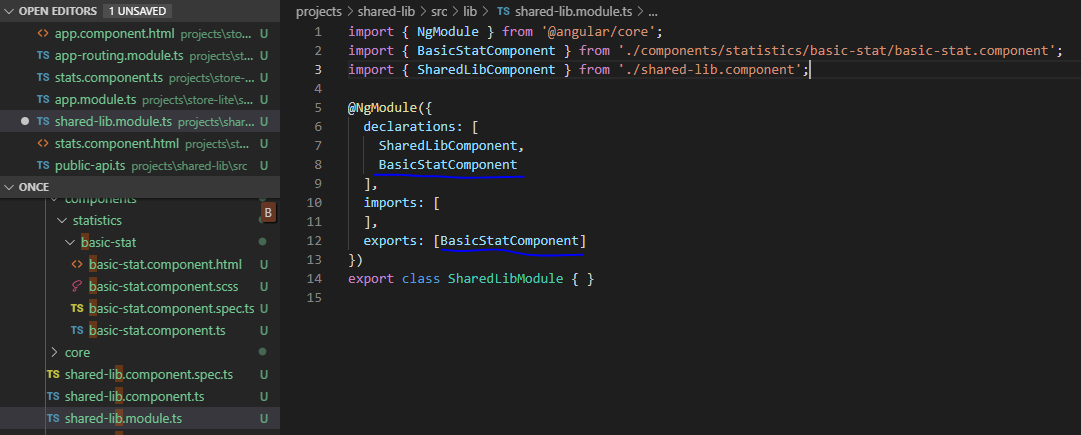
It is obvious that sometimes we will need to create generic components that will be reused across all feature applications. For example, for **StoreLite** we need to create a component (container) that contains three stats components inside.

It's clear that the three stats components inside the main container component are very similar, so instead of creating three components we can create a generic component parameterizable and display this generic component three times, by this way we don't repeat ourselves. On the other hand, What happen if any other feature application needs to display this generic stat component?. For these reasons, it would be a good practice to create a generic component in the shared library which can be displayed several times across several feature applications. As an example, we are going to create a basic-stats component that shows a number at the top and a text message at the bottom:





On the shared-lib.module.ts we need to declare and export the new component to make it accesible from feature applications:

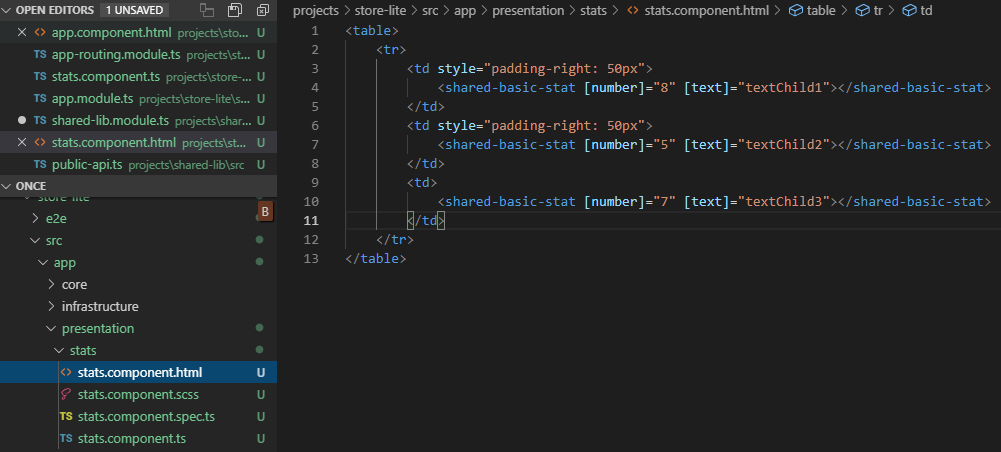
****

**Create Component On Feature Application**

We are going to create a component on **StoreLite** feature application that shows three times the generic component previously created on **shared-library**.

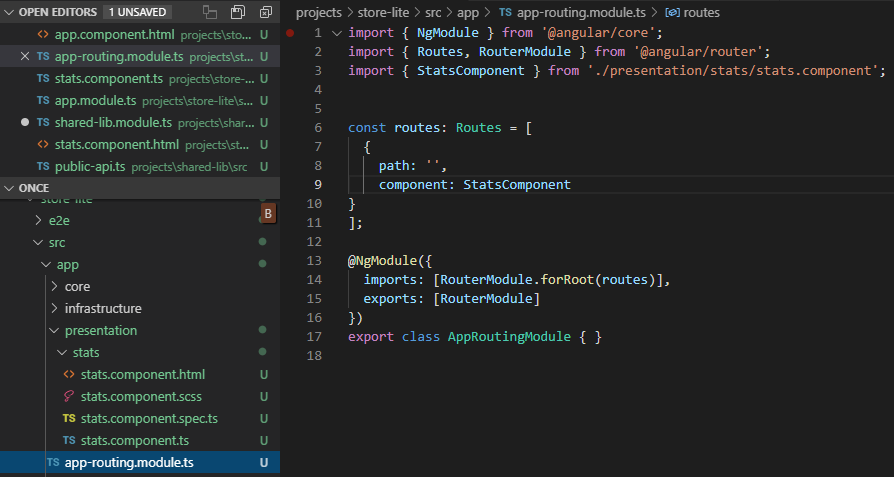
For the feature application we can follow the **“clean architecture”** approach, so we will have our app divided in three parts: **core, infrastructure** and **presentation**.

Inside **presentation** folder is where we create our components:

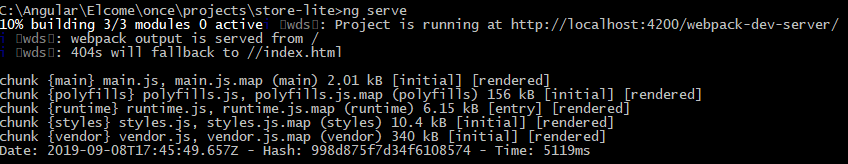


For now, the inputs for the generic component are static. We only want to check that we can use the generic component created in the library.

In the **app-routing.module.ts** we define a default route to this new component:



Now we are ready to launch **StoreLite** application:



As we can see, the feature app runs perfectly and is making use of the **shared-library**.

**Can we have each project on it's own workspace / repository?**

Yes, I have made a proof of concept and it works. We can create a workspace for the library, and a different workspace for each feature application. Then, we can have a github repository for each workspace.

Although it would work, in my opinion, the best approach is to have all projects in the same workspace. If we keep all together and the team working on FeatureApp1 needs to make a change in the shared library, they can easily realise if the change made causes a build error on any other FeatureApp. In the opposite case, if we split it into different workspaces, the team working on FeatureApp1 needs to download the shared-library project, make the change on it, build it and copy the compilated code to the FeatureApp1 dist folder (or use npm to publish and use the shared-library), but would be very difficult to detect if the change made to the shared-library affects to any other FeatureApp.

**Abstraction Layer**

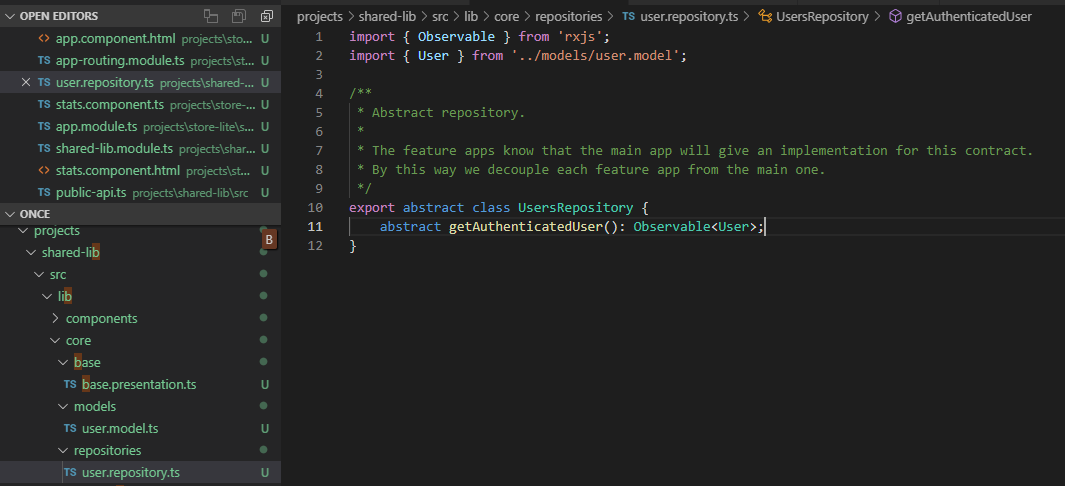
One of our goals is to isolate all feature apps from **ONCE** main app. In fact, **ONCE** main app will be developed later than most feature apps. But what about data that is common to most feature apps but will be provided by **ONCE** main app??.

Imagine we need to display the logged user name in our stats component on **StoreLite**. **ONCE** main app is the responsable to manage the user state and also most feature apps will be interested in have access to the user info.

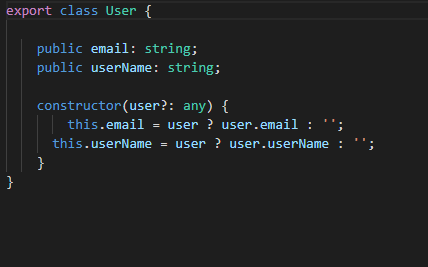
At this point, we need to introduce an abstraction layer. For interact with data managed by **ONCE**, what we need to use is a contract to define what must be provided. By this way, when I'm developing a feature application I care about the data I will have available from **ONCE**, but I don't care how **ONCE** will get and process that data. We can use ***abstract repositories*** for this purpose.

If the data is common to more than one feature app we need to define these repositories into the **shared-library**.

Let's create our repository for users:

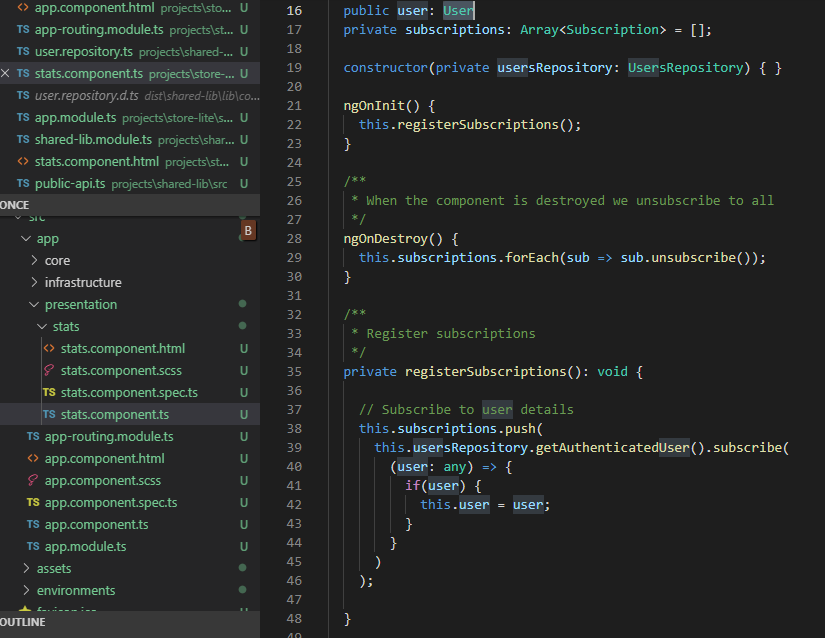


We define an abstract class with the methods we need. **ONCE** will be responsible to extend this class and provide an implementation for the methods, however, the feature applications will work with the abstract class all the time. The method we are defining will give us the authenticated user as an observable. We have defined a model for the user as well:

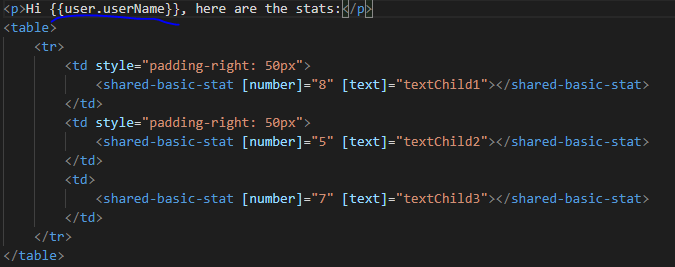


*(Don't forget to rebuild the library after any change)*

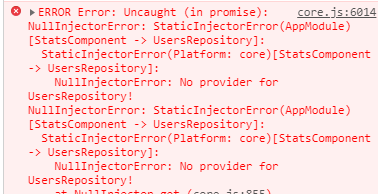
Now, we can change our stats component in **StoreLite** feature app to subscribe to the user data provided by the repository. We inject the abstract repository in the component constructor:



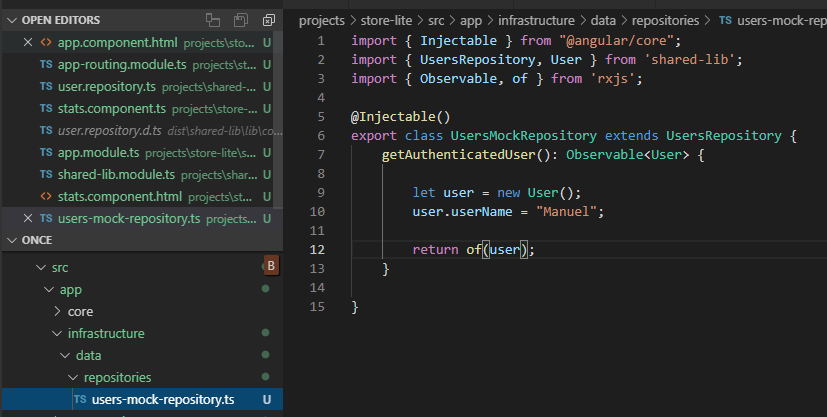
Now we can display a message using the user name:



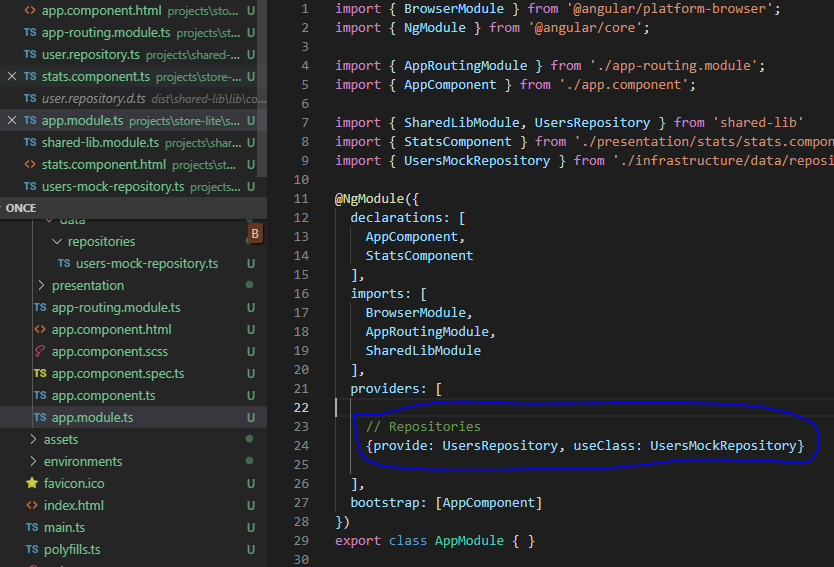
We run the **StoreLite** application and we get this error:



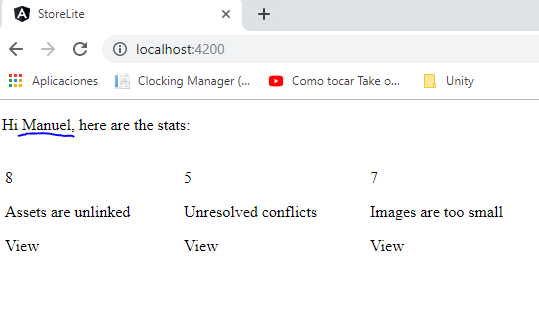
The problem is that we expect **ONCE** main app to provide an implementation to the abstract repository injected, but we are running **StoreLite** in an isolated way, so no one is providing an implementation for the abstract class. What we can do at this stage is provide such implementation by ourselves inside **StoreLite**. In this case we are going to provide a mock for the repository inside **StoreLite infrastructure** folder:



Now we have to define the *provider* in the app.modul.ts of the **StoreLite** feature app:



If we run now the **StoreLite** feature app we can see we have the user name available in our stats component:

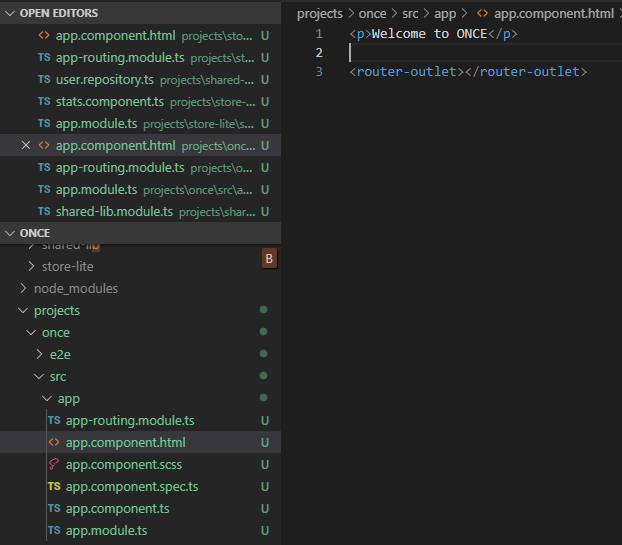


**Later Creation Of ONCE**

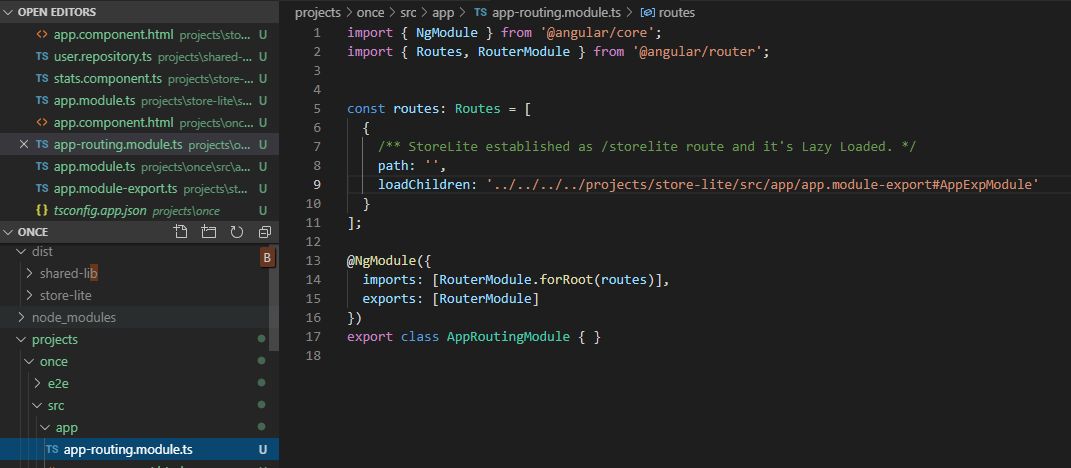
In the future, we will need to create **ONCE** main app. We only have to create another application project inside the workspace:



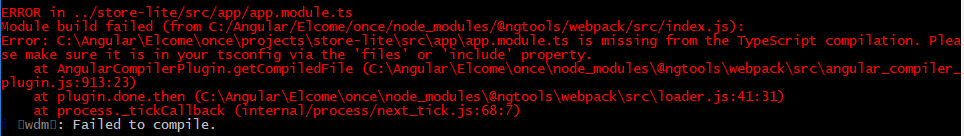
This creates a new project in our workspace. We modify **app.component.html** to reflect we are loading **ONCE** app:



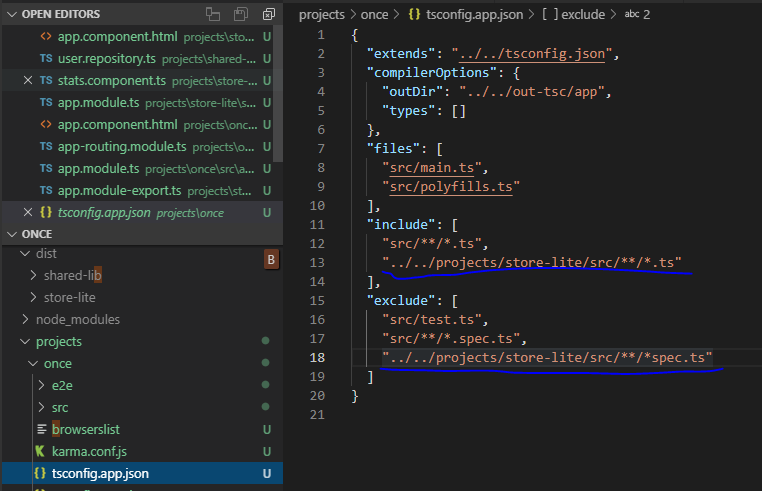
And we modify **app-routing.module.ts** to lazy load **StoreLite** feature application if we navigate to **/storelite**:



And we try to run **ONCE** main app but we get this error:



By default, **StoreLite** typescript code is not part of the main **ONCE** project. So we have to configure it in a **src/tsconfig.json** file:



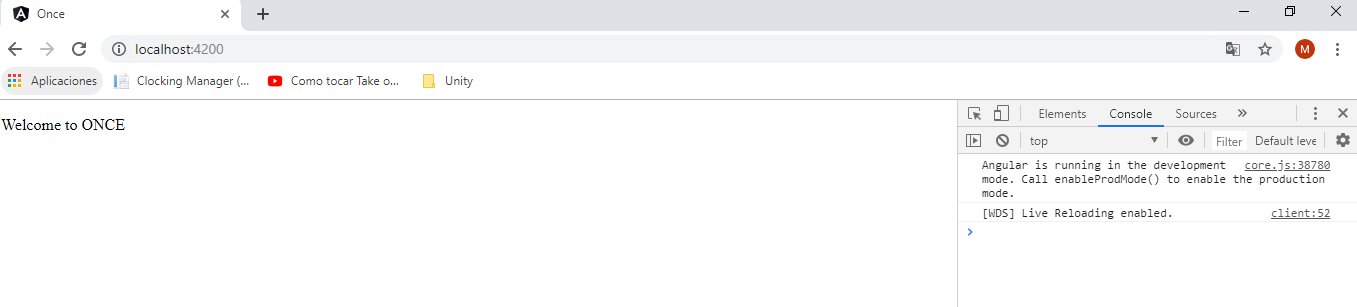
You may have noticed that in route loadChildren param I have not used **app.module.ts** (of a respective feature application), but instead used: **app.module-export.ts.**

I did this because when an application is started separately — we need all the possible modules for its independent work. But for using an application as a module for **ONCE** we have to change its main module.

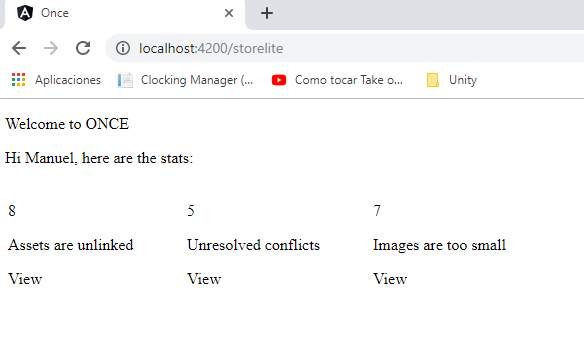
So each feature application (**StoreLite**, **Maintenance**, ...) has:

* app.module.ts — for independent development
* app.module-export.ts — to be used when included in **ONCE** main app

If now we launch **ONCE** application:

It works !

Let's try /storelite:



We can see that it's loading **ONCE** as the main application and then **StoreLite** as a feature application or sub-application because we are navigating to **/storelite**. Also note the **StoreLite** sub-application has been *lazy loaded*.

**Central Store (NgRx)**

*Why Should I Use NgRx for State Management?*

Using NgRx Effects and Store, any interaction with external resources side effects, like network requests, web socket and any business logic can be isolated from the UI. This isolation allows for more pure and simple components, and keep the **single responsibility principle**.

Testing becomes very straightforward because things are more decouple so you don’t have to deal with lot of dependencies while writing tests (you can use mocks).

In particular, you might use NgRx when you build an application with a lot of user interactions and multiple data sources, when managing state in services are no longer sufficient.

Another reason is following the **SHARI** (Shared, Hydrated, Available, Retrieved, impacted) principle.

*StoreLite example*

For example, on StoreLite we need to deal with Stats data. This data will be used by more than one component (Store Stats and Storage Stats) and will be sensitive to be updated due to the user interactions through other components, like for example File Upload component (SignalR).

So we have several actors interacting with the same data, therefore, wouldn’t be a good idea to isolate this data management into it’s own module? By this way, if one actor updates the data, all changes will be reflected automatically to all subscribers.

Additionally, if we follow the **single responsibility principle**, the components or services don’t need to care about how the data is managed, they only should care about their responsibilities, the data management should be responsibility of a third part module built for that purpose.

Can we use services for state management?

Yes, we can do it. Shared services (Dependency Injection) solves the problems of state management to an extent. However:

1. One can still easily make the mistake of copying state from a service and modifying it within the component.
2. Lots of services required for even a medium-sized user interactive application. If we split our data across several services, we are going to need to write similar methods in all of them to manage or deal with that data.
3. The application is highly prone to changing requirements.
4. More tightly coupled components.
5. Potential Issues with Angular change detection e.g. data will not be changed in components if the data reference changes in the service.

NgRx is based on the flux architecture with the concept of a single universal store which serves as the single source of truth in the application at all times. i.e so long as data is from the store, it is always correct.

Furthermore, **NGRX is based on Reactive Programming principles which take advantage of the Angular OnPush** change detection strategy as well as keep track of state changes in the application at run-time.

Although an NgRx based project may require more knowledge, it teaches you to adapt to a new development mindset, makes your application less prone to change requirements and helps you to develop maintainable, scalable and easily debuggable Angular applications.

Also NgRx provides Support for lazy loading through the StoreModule.forFeature() and EffectsModule.forFeature() module imports which helps to improve app performance as each module loads only the parts of the store it requires.

In terms of complexity, the store and unidirectional data flow greatly reduce coupling between parts of your application. This reduced coupling reduces the complexity of your application, since each part only cares about specific states.

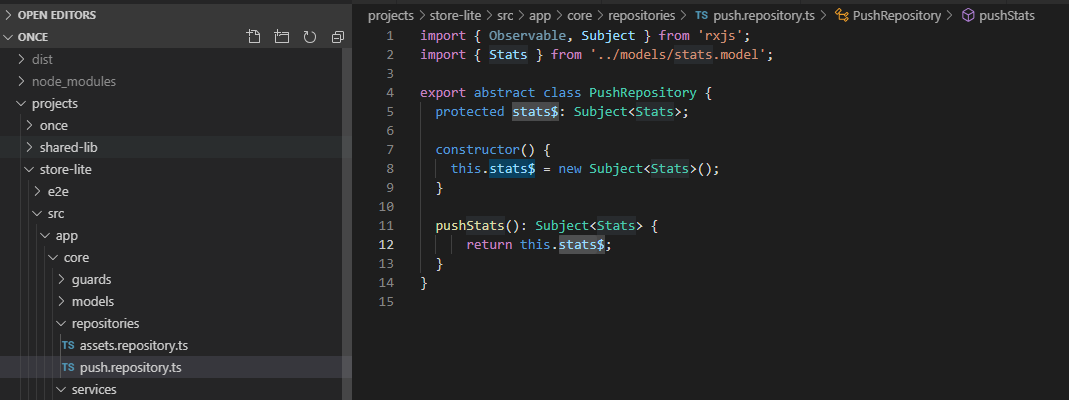
The entire state of your application is stored in one place, so it is easy to have a global view of your application state and helps during development. Also, with Redux comes a lot of nice dev tools that take advantage of the store and can help to reproduce a certain state of the application or make time travel, for example.

Ngrx shines the most when you have a lot of external actors that can modify your application, such as a monitoring dashboard. In those cases, it is hard to manage all the incoming data that are pushed to your application, and state management becomes hard. That is why you want to simplify it with an immutable state, and this is one thing that the Ngrx store provides us with.

**Push Data**

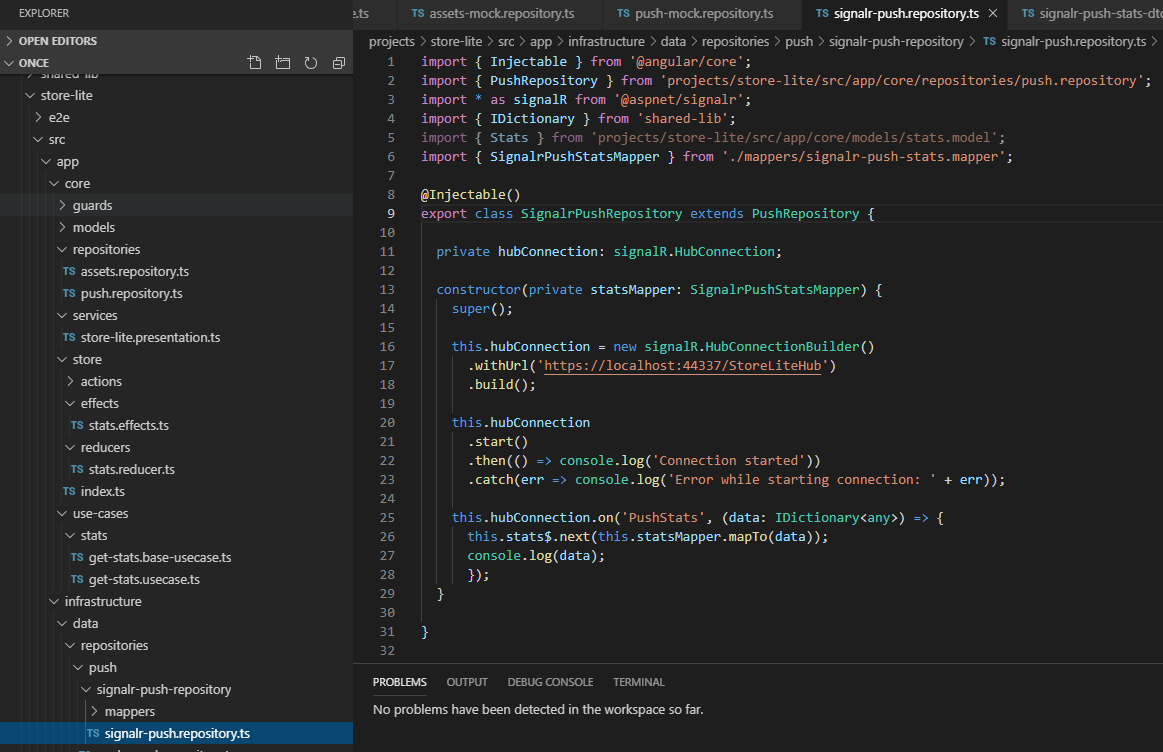
We want to be able to receive data from external resources like SignalR. But we want to make it technology agnostic, could be SignalR or any other technology.

So to accomplish this, we need to add an abstraction layer. We have created an abstract class for manage the data related to stats that the server can push to the app:

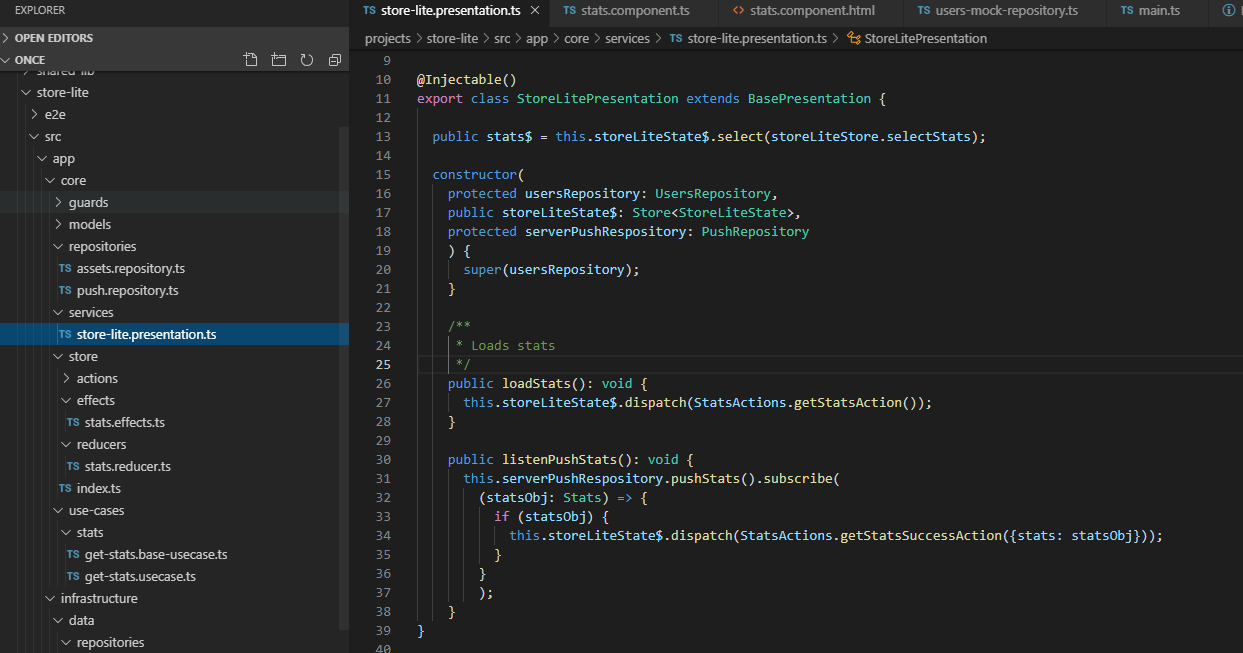


Any different kind of technology we want to use to push stats data to the app will have to inherit this class, because our services are going to subscribe to the observables inside this class.

For example, for using SignalR we can extend this class like this:



Basically is a class that will deal with SignalR technology but ultimately will change the value of the abstract class observable, so all our subscribers will be notified. For example the presentation service will be notified when the stats observable changes:



When the presentation service is notified about the new stats value, it dispatches the store action “getStatsSuccessAction”, which is the same action that we dispatch if we get the stats data from an api, by this way we are reusing code due to the use of **store pattern**.