**Using the State Manager when generating new modules and services.**

The use of this functionality once in place will allow for rapid development and use of state management when needing it.

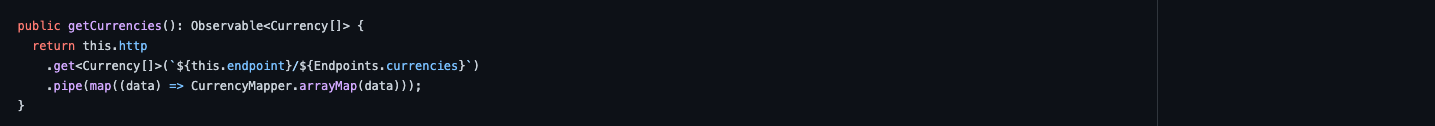
**Benefits:**   
  
1. Complete ownership of the functionality

2. Less templating

3. Rapid production

***NB: State is not always needed when creating new calls to the back end, but in some clear examples such as gaining access to user profile data (see Futurama Project), we can save state and make use of the same data without needing additional calls to the back end for the same data.***

**Step One: Creating the service call:**

Your service call method might look like this:   
  


***NB: this service call was written for another project using a Mapper to decouple naming rules from the front to the back end.***

This service call does not handle state, but let’s look at Futurama where we want to save the user detail state.



In the Futurama Project navigating to the user.service.ts, you will notice that the service extends a helper directive, this in turn extends to the state manager directive. It is here that state is managed.  
  
after .pipe(… we can call this.updateState() method and pass it an enum to save the unique state type we require.

The above example uses the enum located at the top of the module in the \_enum directory. user-state.enum.ts and looks like this:   
  
A picture containing graphical user interface

Description automatically generated

The exported enum is a very simple example, but where needed, additional instances of state can be added. Here we use just user details, so set **User** only. E.g. You could set:   
  
**InitialUser** and save that state as a first point, but set **User** later, so you have 2 states of the user data.  
  
We also export a constant which defines a state interface/model – using an additional enum StateType, we can define the type of data we want to save in state. BehaviourSubject, ReplaySubject etc.

This will allow us to **set** the state at any point during the lifecycle of the components you require using this state.



Looking back at the service method, we can then use the new enum or state type once we have data from the back end, by simply calling this.updateState(ENUM.Type).

This will get us a state to listen to.

**Step 2: Using state inside our components:**

The idea behind this is to ensure we are not required to make multiple endpoint calls for the same data, so in the Futurama Project, the one call for user data is made in the home-landing.component.ts – this is really the feature landing page which loads first, so calling for the data here means that routing into other modules, we can make use of the call once we have loaded the app.

1. Start by listening for State

Graphical user interface, application

Description automatically generated

In the home-landing.component.ts there is a method which assigns the app a user which gives us the ability to listen for state. Thanks to a subscription and state manager directive, we get and set, saving the response as a variable we might want to use in our component logic.

Once we have the response, we can then **set** this response as BehaviourSubject, which can then pass around the application.  
  
on the init lifecycle hook, we load and assign data.

1. Loading the data from the Back End  
     
   Shape

   Description automatically generated

Using the subscription directive, we can call the dispatch to our endpoint. This is the only call we use to the back end on the Futurama Project.  
  
**Step 3: Using State in other modules:**

Once we have set the state of our user details, we are able to use this data across the boundaries of unassociated modules.

Graphical user interface, text

Description automatically generated

To ensure that we are not required to write duplicate code across the application, you will notice in \_common, an additional directive user-details.directive.ts. This is a simple directive which uses the user service to obtain values from the saved state. And means that by extending your component class where necessary, you are able to access this value by simply adding this.assignUser() during the lifecycle of the component.   
  
In the event you require updating the saved state elsewhere this.setLoggedUser() will re-save the state.

Your component will require an extension to the directive and also passing the user service into super(). This will give the component access to the saved state.  
  
Graphical user interface, application

Description automatically generated

In the update-user.component.ts, we can see how this is used. Once this is complete, the component can make use of the state to continue developing solutions from business requirement.