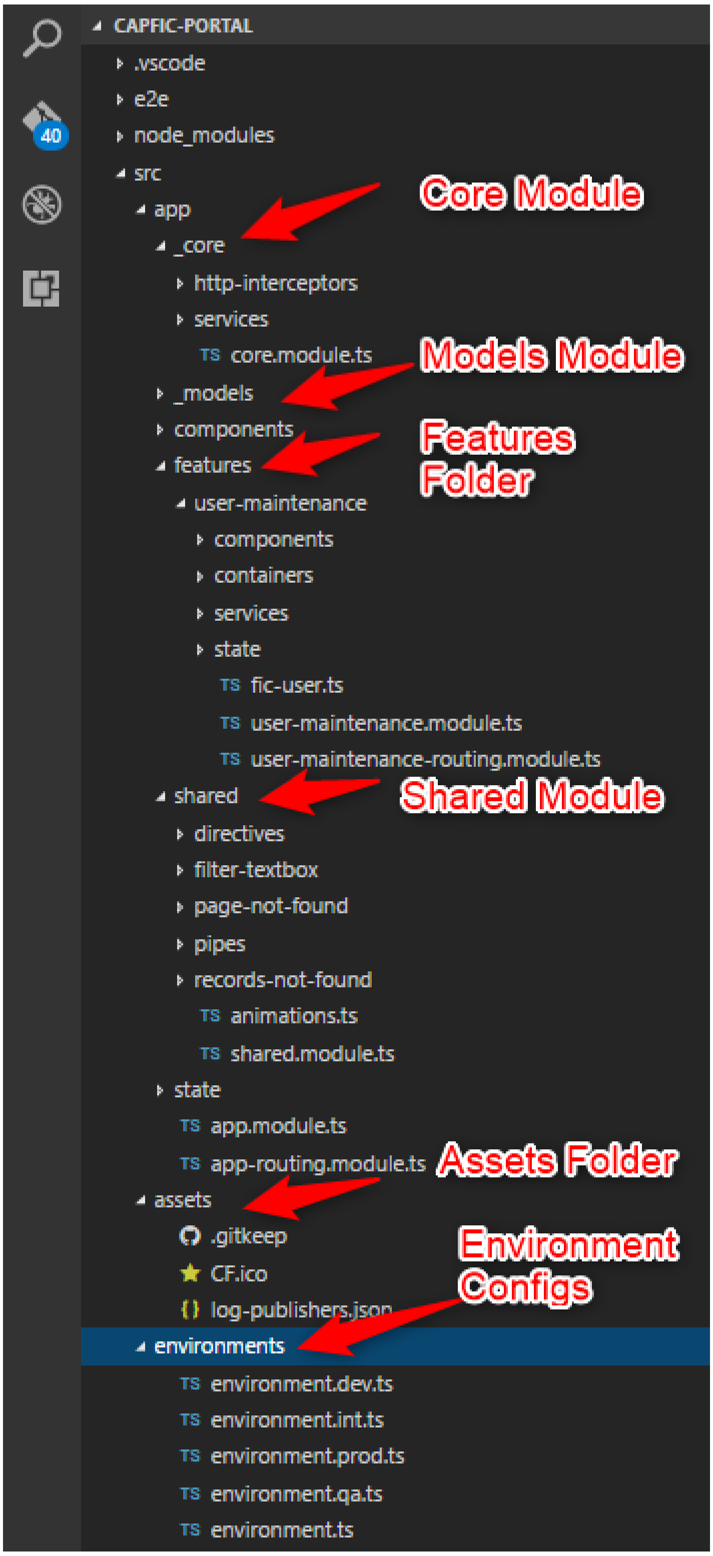
CapFic Angular7 Prototype

Prerequisites

You should watch the below plural sight videos to give you an understanding of the patterns and technologies that we will be using in this application. if you are new to Angular, watch the Angular getting started video too:

- Angular: Getting Started https://app.pluralsight.com/library/courses/angular-2-getting-started-update/table-of-contents
- Unit Testing in Angular https://app.pluralsight.com/library/courses/unit-testing-angular/table-of-contents
- Angular Routing https://app.pluralsight.com/library/courses/angular-routing/table-of-contents
- Angular NgRx: Getting Started https://app.pluralsight.com/library/courses/angular-ngrx-getting-started/table-of-contents
- Angular HTTP Communication https://app.pluralsight.com/library/courses/angular-http-communication/table-of-contents
- Angular Reactive Forms https://app.pluralsight.com/library/courses/angular-2-reactive-forms/table-of-contents

Application Structure



```
CAPFIC-PORTAL
  .vscode
  e2e
  node_modules
  src
    app
      _core
        http-interceptors
        services
        core.module.ts
      _models
      components
      features
        user-maintenance
          components
          containers
          services
          state
          fic-user.ts
          user-maintenance.module.ts
          user-maintenance-routing.module.ts
      shared
        directives
        filter-textbox
        page-not-found
        pipes
        records-not-found
        animations.ts
        shared.module.ts
      state
      app.module.ts
      app-routing.module.ts
    assets
      .gitkeep
      CF.ico
      log-publishers.json
    environments
      environment.dev.ts
      environment.int.ts
      environment.prod.ts
      environment.qa.ts
      environment.ts
```

Core Module

Models Module

Features Folder

Shared Module

Assets Folder

Environment Configs

Core Module

The core module is used to host any kind of "singleton", something that only gets instantiated, declared or used once in the application. An example of an item declared in the core module is a service. All services should be saved in the core module unless it is only used in a particular feature. Features are discussed later. HTTP Interceptors are another example of items used in the core module as they are used once in one place throughout the application.

Models Module

This module is used to declare all models that are common throughout the application. An example would be a lookup model that gets referenced in multiple features. Models that are particular to a specific feature can be placed in that feature folder.

Features Folder

The Features Folder is used to house all feature modules created in the application. Each feature should be created in its own module to enable lazy loading of the feature when required.

Shared Module

The shared module is used to house all dumb components, pipes and directives that will be used by multiple features.

Assets Folder

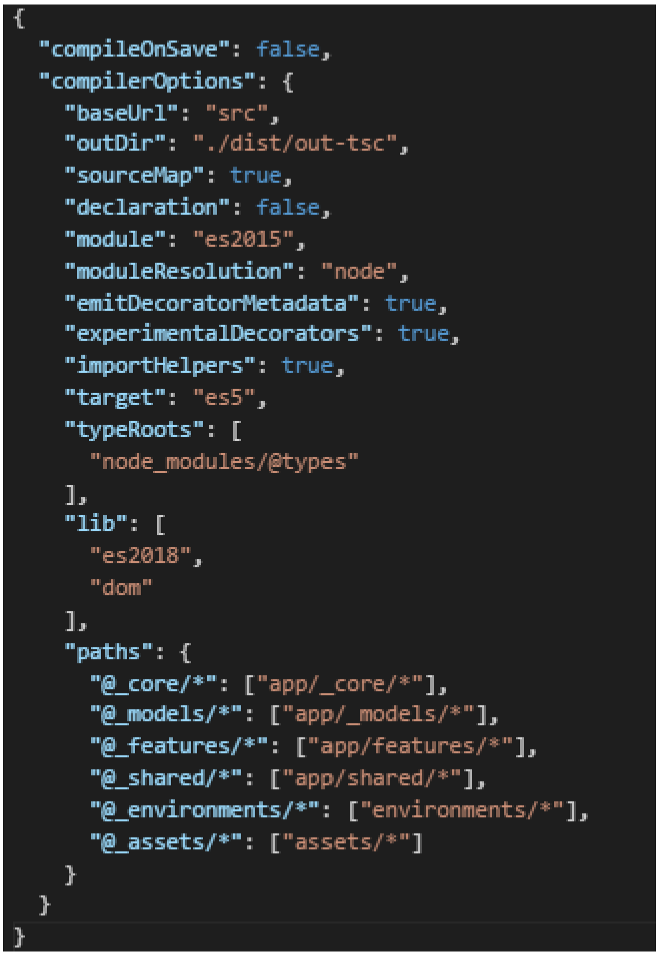
The assets folder is used to store files to reference in your application. Images, config files etc.

Environment Configs

These .ts files are swapped out with the original environment.ts file when building the application for a specific environment. If we do a production build, the environment.ts file will be replaced by the environment.prod.ts file with properties specific to the production environment.

tsconfig.json

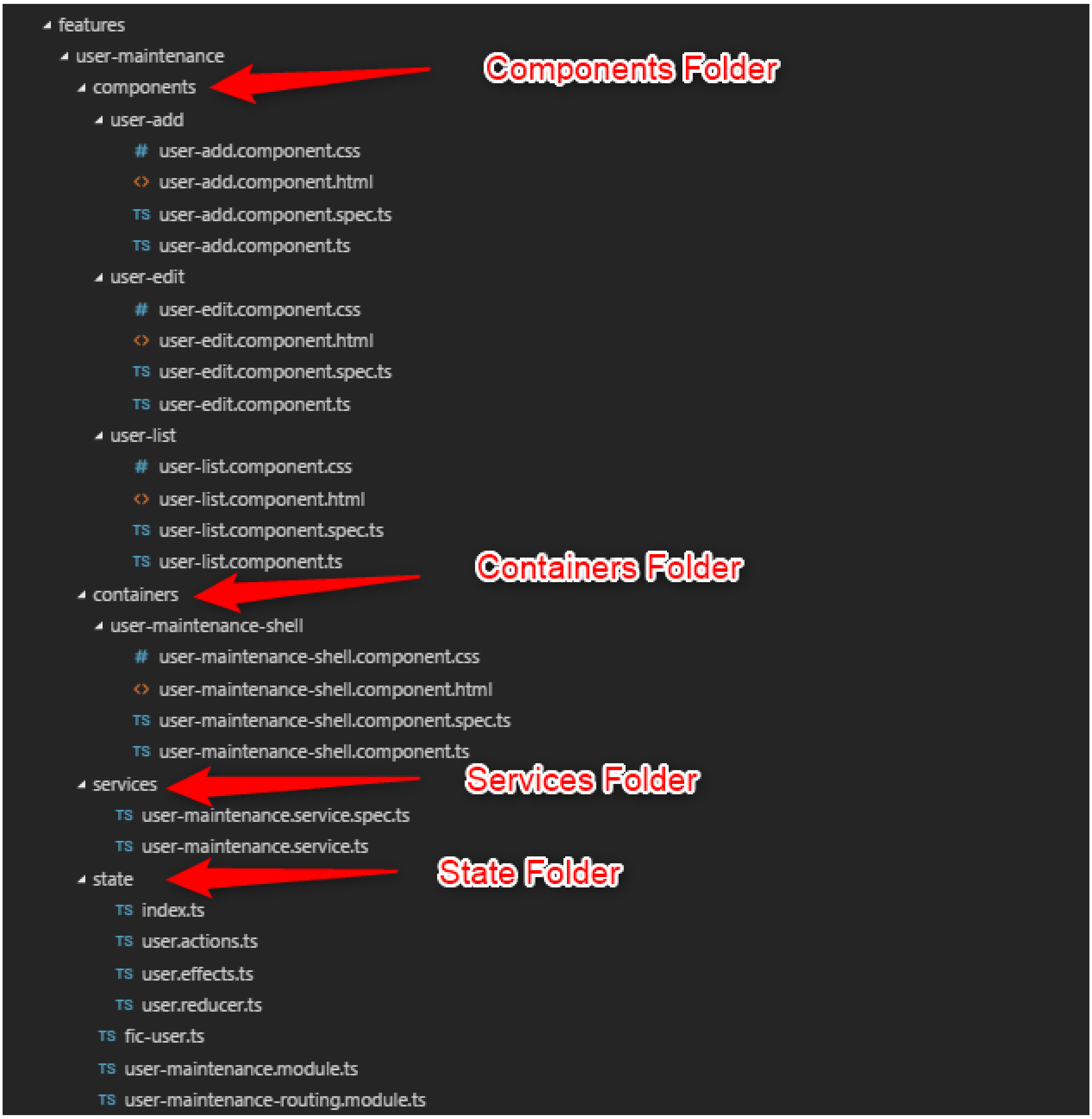
The *tsconfig.json* file in the route of the *src* file has aliases created to reference each major folder described above to aid in referencing files in the application.



```
{
  "compileOnSave": false,
  "compilerOptions": {
    "baseUrl": "src",
    "outDir": "./dist/out-tsc",
    "sourceMap": true,
    "declaration": false,
    "module": "es2015",
    "moduleResolution": "node",
    "emitDecoratorMetadata": true,
    "experimentalDecorators": true,
    "importHelpers": true,
    "target": "es5",
    "typeRoots": [
      "node_modules/@types"
    ],
    "lib": [
      "es2018",
      "dom"
    ],
    "paths": {
      "@_core/*": ["app/_core/*"],
      "@_models/*": ["app/_models/*"],
      "@_features/*": ["app/features/*"],
      "@_shared/*": ["app/shared/*"],
      "@_environments/*": ["environments/*"],
      "@_assets/*": ["assets/*"]
    }
  }
}
```

To reference the filter service in the _core module from the user-list.component.file will now be '**@_core/services/filter.service'** instead of '**../../../_core/services/filter.service'**. It becomes quite tedious to traverse a folder structure with ../

Feature Modules



```
◢ features
  ◢ user-maintenance
    ◢ components
      ◢ user-add
          # user-add.component.css
          <> user-add.component.html
          TS user-add.component.spec.ts
          TS user-add.component.ts
      ◢ user-edit
          # user-edit.component.css
          <> user-edit.component.html
          TS user-edit.component.spec.ts
          TS user-edit.component.ts
      ◢ user-list
          # user-list.component.css
          <> user-list.component.html
          TS user-list.component.spec.ts
          TS user-list.component.ts
    ◢ containers
      ◢ user-maintenance-shell
          # user-maintenance-shell.component.css
          <> user-maintenance-shell.component.html
          TS user-maintenance-shell.component.spec.ts
          TS user-maintenance-shell.component.ts
    ◢ services
        TS user-maintenance.service.spec.ts
        TS user-maintenance.service.ts
    ◢ state
        TS index.ts
        TS user.actions.ts
        TS user.effects.ts
        TS user.reducer.ts
      TS fic-user.ts
      TS user-maintenance.module.ts
      TS user-maintenance-routing.module.ts
```

A typical feature folder will have a components, containers, services (if it has services specific to the feature) and state folder each discussed below.

Components Folder

The components folder is used to house all the dumb components for the feature module. A dumb component is a component that is only used to render the view. It should not make use of any HTTP calls or any type of service that makes calls outside of the application. These types of services should be done in the smart components in the container folder. A dumb component may make use of services such as the filter service which only provides filtering logic or the sort service which provides sorting logic. Looking at the components in the components folder we see a user-list component which is only used to display a list of users in a table. The user-edit component is used to edit a users details and send an event containing the user being edited to the parent container class so that the container class can make the API call to update the user. A similar scenario occurs with the user-add component.

By default Angular uses the strategy *`ChangeDetectionStrategy.Default`*. The default strategy doesn't assume anything about the application, therefore every time something changes in our application, as a result of various user events, timers, XHR, promises, etc., a change detection will run on **all components**. This means anything from a click event to data received from an ajax call causes the change detection to be triggered. Now, imagine a big application with thousands of expressions; If we let Angular check every single one of them when a change detection cycle runs, we might encounter a performance problem.

We make use of *`ChangeDetectionStrategy.OnPush`* `in this application. This tells Angular that the component only depends on its @inputs() ( aka pure ) and needs to be checked only in the following cases:`

- **The Input reference changes**. By setting the `onPush` change detection strategy we are signing a contract with Angular that obliges us to work with immutable objects. The advantage of working with immutability in the context of change detection is that Angular could perform a simple reference check in order to know if the view should be checked. Such checks are way cheaper than a deep comparison check.

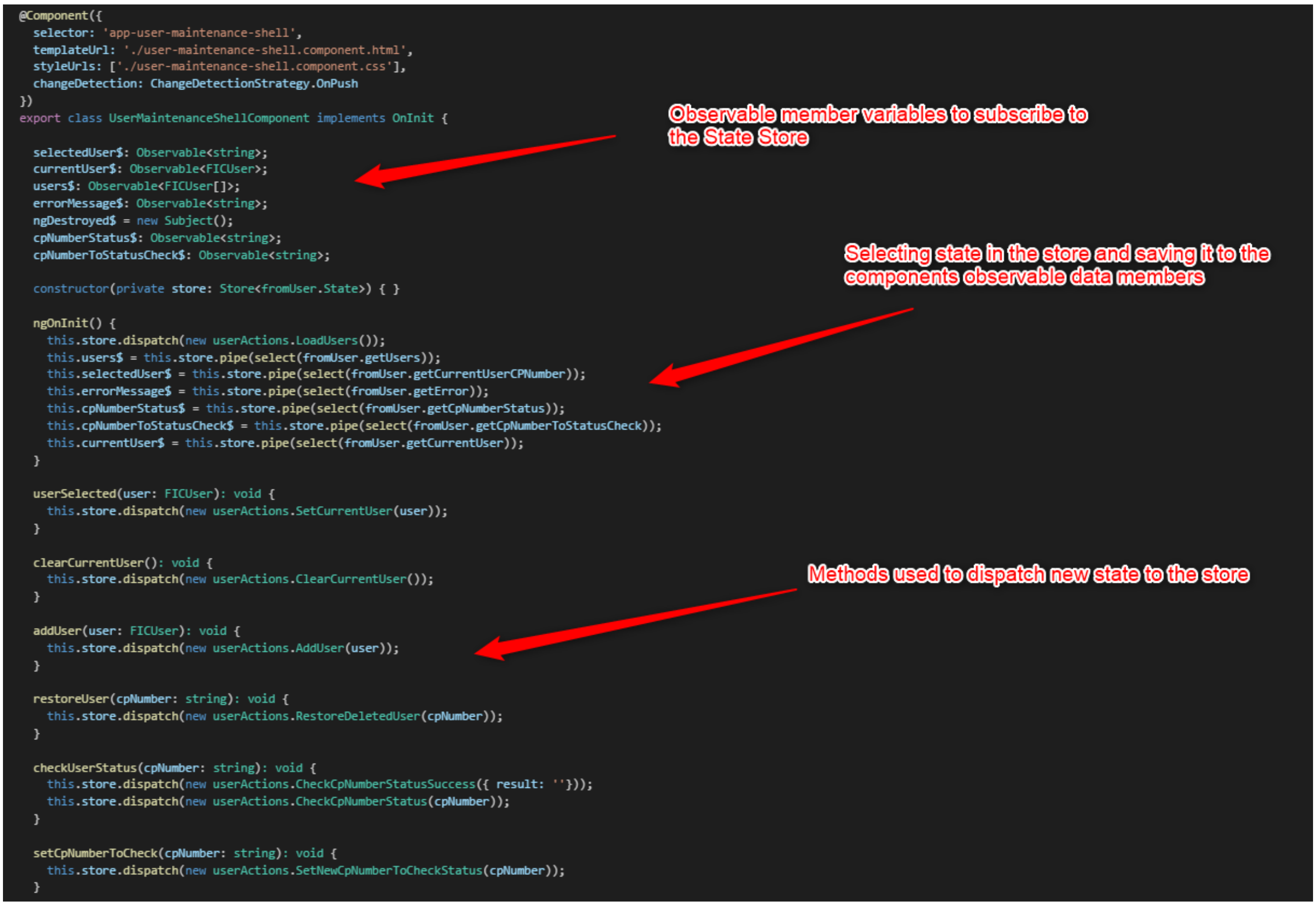
- **An event originated from the component or one of its children.** A component could have an internal state that's updated when an event is triggered from the component or one of its children. This rule only applies to DOM events so having a method which updates the internal state called from the DOM (html file) will trigger an update of the UI. An event that does not originate from the DOM (html file) will not trigger an update to the UI.
- **We run change detection explicitly**. Angular provides us with three methods for triggering change detection ourselves when needed. The first is `detectChanges()` which tells Angular to run change detection on the component and his children. The second is `ApplicationRef.tick()` which tells Angular to run change detection for the whole application. The third is `markForCheck()` which does NOT trigger change detection. Instead, it marks all `onPush` ancestors as to be checked once, either as part of the current or next change detection cycle.

You need to define the changeDetection strategy in the @Component decorator as seen in the below screenshots. You will need to think in terms on immutability when moving data around and make use of immutable strategies to copy objects or you will be very puzzled when the UI does not update. You will need to have knowledge of reactive forms as using two way binding goes against the OnPush change detection strategy.

Containers Folder

The main objective of the smart components that are stored in the containers folder is to make any kind of HTTP service or state calls for the dumb components to render on the screen. Using a parent container helps us break down our components into smaller pieces making them easier to test. The dumb components only need to worry about rendering the state on the screen while the smart components only need worry about performing service calls that change the state of the application.

Below is a typical layout of a smart component. Using this style is clean and elegant as we do not need to worry about subscribing to or unsubscribing from the observable's. We have methods that dispatch changes to the state in the store and we have selectors that grab the state in the store and store them in the observable's. Each time state is updated by the methods dispatching new or changed state, the observable's automatically get updated with the new values in the store.

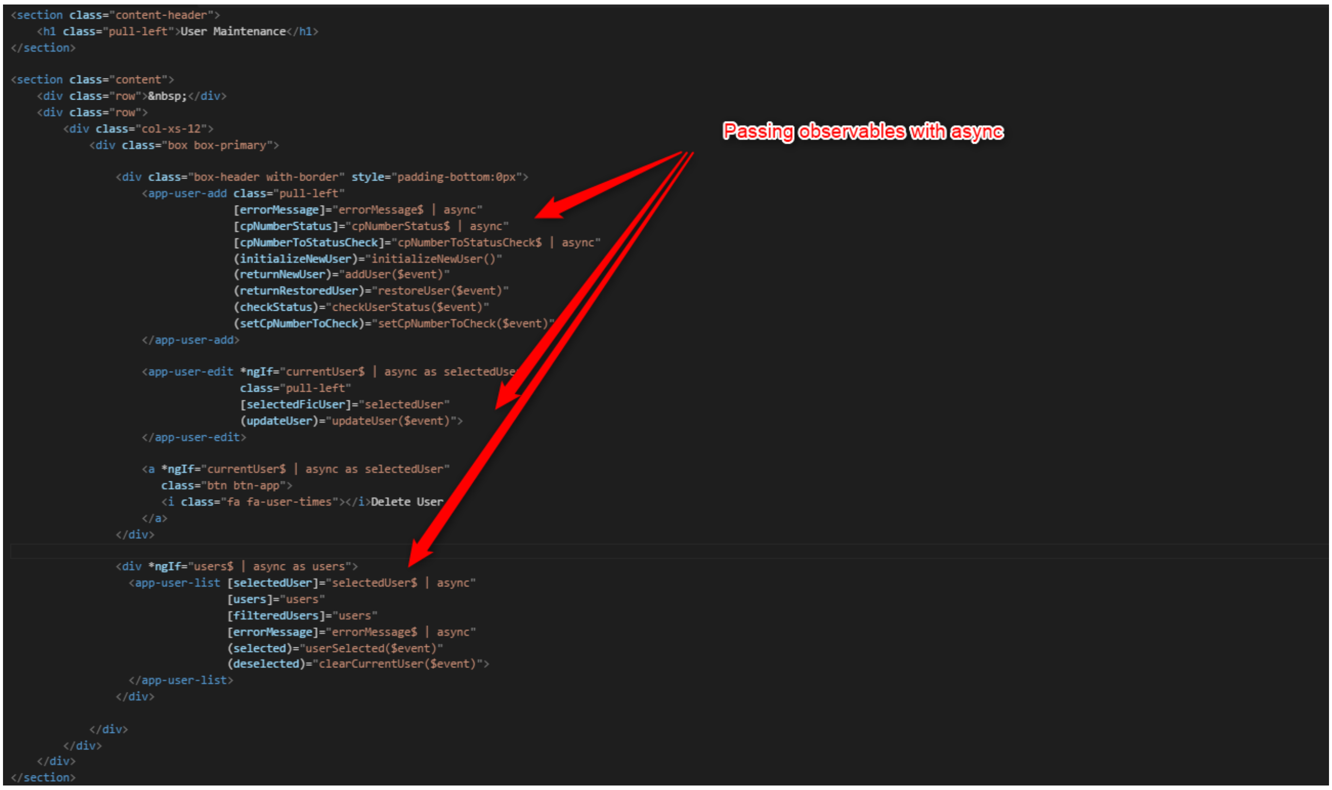


```
@Component({
  selector: 'app-user-maintenance-shell',
  templateUrl: './user-maintenance-shell.component.html',
  styleUrls: ['./user-maintenance-shell.component.css'],
  changeDetection: ChangeDetectionStrategy.OnPush
})
export class UserMaintenanceShellComponent implements OnInit {

  selectedUser$: Observable<string>;
  currentUser$: Observable<FICUser>;
  users$: Observable<FICUser[]>;
  errorMessage$: Observable<string>;
  ngDestroyed$ = new Subject();
  cpNumberStatus$: Observable<string>;
  cpNumberToStatusCheck$: Observable<string>;

  constructor(private store: Store<fromUser.State>) { }

  ngOnInit() {
    this.store.dispatch(new userActions.LoadUsers());
    this.users$ = this.store.pipe(select(fromUser.getUsers));
    this.selectedUser$ = this.store.pipe(select(fromUser.getCurrentUserCPNumber));
    this.errorMessage$ = this.store.pipe(select(fromUser.getError));
    this.cpNumberStatus$ = this.store.pipe(select(fromUser.getCpNumberStatus));
    this.cpNumberToStatusCheck$ = this.store.pipe(select(fromUser.getCpNumberToStatusCheck));
    this.currentUser$ = this.store.pipe(select(fromUser.getCurrentUser));
  }

  userSelected(user: FICUser): void {
    this.store.dispatch(new userActions.SetCurrentUser(user));
  }

  clearCurrentUser(): void {
    this.store.dispatch(new userActions.ClearCurrentUser());
  }

  addUser(user: FICUser): void {
    this.store.dispatch(new userActions.AddUser(user));
  }

  restoreUser(cpNumber: string): void {
    this.store.dispatch(new userActions.RestoreDeletedUser(cpNumber));
  }

  checkUserStatus(cpNumber: string): void {
    this.store.dispatch(new userActions.CheckCpNumberStatusSuccess({ result: ''}));
    this.store.dispatch(new userActions.CheckCpNumberStatus(cpNumber));
  }

  setCpNumberToCheck(cpNumber: string): void {
    this.store.dispatch(new userActions.SetNewCpNumberToCheckStatus(cpNumber));
  }
```

Observable member variables to subscribe to the State Store
Selecting state in the store and saving it to the components observable data members
Methods used to dispatch new state to the store

You will notice that all the data passed to the dumb components are passed with the async pipe. Passing observable's with async lets angular manage the subscribing and unsubscribing of for you. This way you wont need to worry about memory leeks.

The output methods receive events from the dumb components to send changed data to the Store. The (selected) method passed to the <app-user-list> component below allows the user-list component to send the selected FicUser in the list of users to the smart component via the userSelected method which then dispatches the selected user to the store.

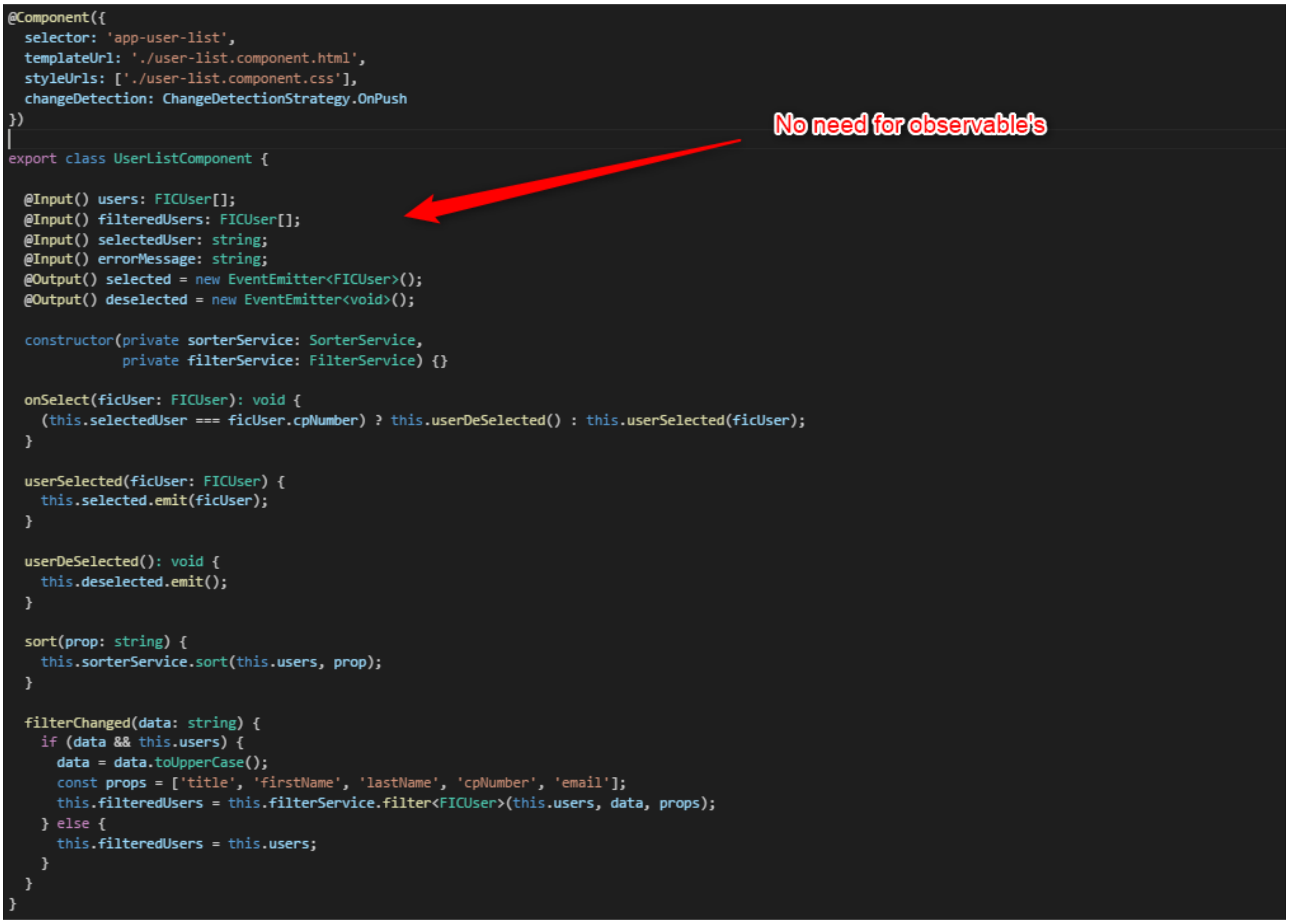


Passing observables with async

```html
<section class="content-header">
    <h1 class="pull-left">User Maintenance</h1>
</section>

<section class="content">
    <div class="row">&nbsp;</div>
    <div class="row">
        <div class="col-xs-12">
            <div class="box box-primary">

                <div class="box-header with-border" style="padding-bottom:0px">
                    <app-user-add class="pull-left"
                                  [errorMessage]="errorMessage$ | async"
                                  [cpNumberStatus]="cpNumberStatus$ | async"
                                  [cpNumberToStatusCheck]="cpNumberToStatusCheck$ | async"
                                  (initializeNewUser)="initializeNewUser()"
                                  (returnNewUser)="addUser($event)"
                                  (returnRestoredUser)="restoreUser($event)"
                                  (checkStatus)="checkUserStatus($event)"
                                  (setCpNumberToCheck)="setCpNumberToCheck($event)"
                    </app-user-add>

                    <app-user-edit *ngIf="currentUser$ | async as selectedUse
                                   class="pull-left"
                                   [selectedFicUser]="selectedUser"
                                   (updateUser)="updateUser($event)">
                    </app-user-edit>

                    <a *ngIf="currentUser$ | async as selectedUser"
                       class="btn btn-app">
                       <i class="fa fa-user-times"></i>Delete User
                    </a>
                </div>

                <div *ngIf="users$ | async as users">
                  <app-user-list [selectedUser]="selectedUser$ | async"
                                 [users]="users"
                                 [filteredUsers]="users"
                                 [errorMessage]="errorMessage$ | async"
                                 (selected)="userSelected($event)"
                                 (deselected)="clearCurrentUser($event)">
                  </app-user-list>
                </div>

            </div>
        </div>
    </div>
</section>
```

A dumb component is now much smaller and only focuses on displaying data. We do not need to have complex subscribing in the constructor or the ngOnInit life cycle hook, and no unsubscribing in ngOnDestroy life cycle hook. We also do not need to worry about memory leeks.

The result is a much cleaner component as seen below.



No need for observable's

```typescript
@Component({
  selector: 'app-user-list',
  templateUrl: './user-list.component.html',
  styleUrls: ['./user-list.component.css'],
  changeDetection: ChangeDetectionStrategy.OnPush
})

export class UserListComponent {

  @Input() users: FICUser[];
  @Input() filteredUsers: FICUser[];
  @Input() selectedUser: string;
  @Input() errorMessage: string;
  @Output() selected = new EventEmitter<FICUser>();
  @Output() deselected = new EventEmitter<void>();

  constructor(private sorterService: SorterService,
              private filterService: FilterService) {}

  onSelect(ficUser: FICUser): void {
    (this.selectedUser === ficUser.cpNumber) ? this.userDeSelected() : this.userSelected(ficUser);
  }

  userSelected(ficUser: FICUser) {
    this.selected.emit(ficUser);
  }

  userDeSelected(): void {
    this.deselected.emit();
  }

  sort(prop: string) {
    this.sorterService.sort(this.users, prop);
  }

  filterChanged(data: string) {
    if (data && this.users) {
      data = data.toUpperCase();
      const props = ['title', 'firstName', 'lastName', 'cpNumber', 'email'];
      this.filteredUsers = this.filterService.filter<FICUser>(this.users, data, props);
    } else {
      this.filteredUsers = this.users;
    }
  }
}
```

Services Folder

The services folder is used to house any services particular to the feature, otherwise the service is stored in the core module.

State Folder

The State folder is used to store all the files used to manage the state in the feature. We are using ngRX in this application for state management so the following files will be in the folder:

- an index.ts file for creating selectors which grab the state from the store
- an actions.ts file for creating actions which are classes derived from the Action class to send commands along with a payload to the reducer to update the store.
- an effects.ts file for performing calls outside of the application like HTTP calls which in turn return a success or fail action to the reducer to update the store with the result.
- a reducer.ts file which updates the store according to the Action sent to it.

Feature models

Any models used specifically in the feature and nowhere else are stored in the root of the feature folder. You can create a models folder inside the feature folder if you have more than one model.

HTTP Interceptors

HTTP Interceptors are used as the name suggests, to intercept the HTTP request before it gets sent to the external API and the response coming back from the external API.

In this application, we make use of two interceptors. One to attach headers to the request before it gets sent to the API and one to handle errors in the HTTP response from the server. They are discussed below:

header-interceptor.ts

```
@Injectable()
export class HeadersInterceptor implements HttpInterceptor {

    constructor() { }

    intercept(req: HttpRequest<any>, next: HttpHandler):
Observable<HttpEvent<any>> {

        const jasonReq: HttpRequest<any> = req.clone({
            setHeaders: {
                'Content-Type': 'application/json',
                'Access-Control-Allow-Origin': '*',
                'Accept': 'application/json'
            }
        });

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

Here we take the req paramater and use the HttpRequest cloning function to clone the request and add the headers needed for all requests. We have set the *Content-Type* and *Accept* to 'application/json' and set the '*Access-Control-Allow-Origin*' to '*'. The *Access-Control-Allow-Origin* is set because the external API requires this header to be attached for CORS "Cross Origin Resource Sharing". This is necessary because the Angular app and the API are using different domains. Even though the API and Angular app are hosted on the same server, they use different ports which classifies them as different domains. The *Content-Type* of 'application/json' describes the format of the data being sent in the request.

The second interceptor handles HTTP errors:

http-error-interceptor

```
export class HttpErrorInterceptor implements HttpInterceptor {
    intercept(request: HttpRequest<any>, next: HttpHandler):
Observable<HttpEvent<any>> {
        return next.handle(request)
            .pipe(
                retry(1),
                catchError((error: HttpErrorResponse) => {
                    let errorMessage = '';
                    if (error.error instanceof ErrorEvent) {
                        // client-side error
                        errorMessage = `Error: ${error.error.message}`;
                    } else {
                        // server-side error
                        errorMessage = `Error Code:
${error.status}\nMessage: ${error.message}`;
                    }
                    window.alert(errorMessage);
                    return throwError(errorMessage);
                })
            );
    }
}
```

Here we are using the rxjs pipe operator on the response to retry the API call if it fails one extra time. If it fails again, we handle the error here. We then display an alert on the screen. and throw the error. The alert is using the default browser window alert and a better alert should be used to display the error to the user. Handling the error in an HTTP interceptor is the right way to handle HTTP errors. This way they are all handled in one place so you don't need to change this method in every service in your application.

Shared Components

We need to reuse as much code as possible. We can reuse dumb components as well. All these shared components, pipes and directives should be saved in the shared module.

There are a few shared components, directives and services that can be reused when using a data table. They are listed below.

Filtering a table

The shared module contains a very simple filter textbox that can be used throughout the application. It contains a model with a string variable to store filtered text and then outputs the value to its parent component as seen below

filter-textbox.component.ts

```
export class FilterTextboxComponent {

    model: { filter: string } = { filter: null };

    @Output()
    changed: EventEmitter<string> = new EventEmitter<string>();

    filterChanged(event: any) {
      event.preventDefault();
      this.changed.emit(this.model.filter);
    }
}
```

filter-textbox.component.html

```
<div class="box-header with-border">
    <input type="text"
           name="filter"
           [(ngModel)]="model.filter"
           class="form-control"
           placeholder="Text filter"
           (keyup)="filterChanged($event)">
  </div>
```

The parent component making use of the filter textbox will simply use the filtered text sent from the output event and send it to the FilterService to perform the filtering on the list as seen in the filterChanged method below. The filter service will be able to filter on all the properties provided to it.

user-list.ts filterChanged()

```
filterChanged(data: string) {
    if (data && this.users) {
      data = data.toUpperCase();
      const props = ['title', 'firstName', 'lastName', 'cpNumber', 
'email'];
      this.filteredUsers = 
this.filterService.filter<FICUser>(this.users, data, props);
    } else {
      this.filteredUsers = this.users;
    }
  }
```

This method is called from <app-filter-textbox> component in the user-list.component.html as seen below

user-list.component.html app-filter-textbox component

```
<app-filter-textbox
    (changed)="filterChanged($event)">
  </app-filter-textbox>
```

Sorting a table

The shared module contains a *sortby* directive which is used to sort a table by column.

sortby.directive.ts

```
@Directive({
    selector: '[appSortBy]'
})
export class SortByDirective {

    private sortProperty: string;

    @Output()
    sorted: EventEmitter<string> = new EventEmitter<string>();

    constructor() { }

    @Input('appSortBy')
    set sortBy(value: string) {
        this.sortProperty = value;
    }

    @HostListener('click')
    onClick() {
        event.preventDefault();
        this.sorted.next(this.sortProperty);
    }
}
```

You need to attach the appSortBy directive to the table header you want to sort on with the properties of an item in the list you want to sort by as seen below. The appSortBy directive listens for click events so we need to create a (sorted)="sort($event)" method that will pass the string the directive is listening to to the sort() method in the component.

user-list.component.html appSortBy directive

```
<table id="users-table-header"
         class="table table-bordered table-hover dataTable"
         role="grid"
         style="table-layout: fixed;">
    <thead>
      <tr role="row"
          class="bg-light-blue-gradient">
        <th appSortBy="title" (sorted)="sort($event)" style="width:
50px">Title</th>
        <th appSortBy="firstName" (sorted)="sort($event)" style="width:
150px">Name</th>
        <th appSortBy="lastName" (sorted)="sort($event)" style="width:
150px">Surname</th>
        <th appSortBy="cpNumber" (sorted)="sort($event)" style="width:
100px">CP Number</th>
        <th appSortBy="email" (sorted)="sort($event)" style="width:
200px">Email</th>
      </tr>
    </thead>
  </table>
```

The sort method passes the string and the list to the sort service for sorting.

user-list.component.ts sort() property

```
...


  sort(prop: string) {
    this.sorterService.sort(this.users, prop);
  }


  ...
```

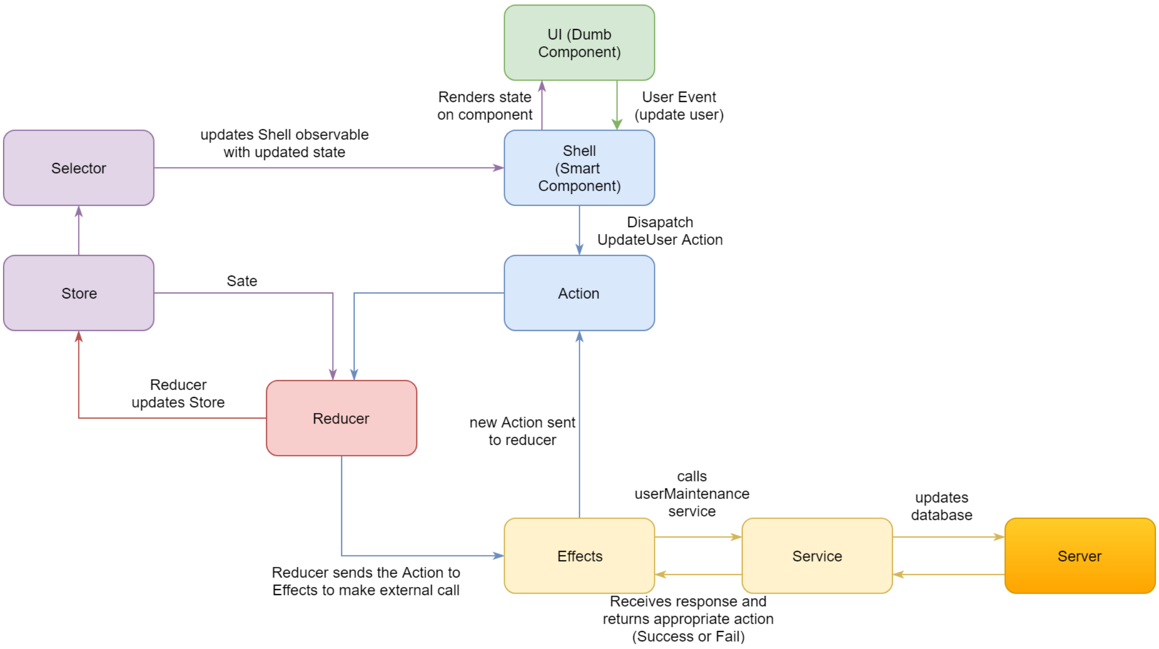
Create as much shared components and directives as possible and add them to the shared module.

State Management

The driving force behind this application is NgRx state management. To successfully adopt this prototype, you need to start thinking in terms of state management and immutability. If you change the state of a component, whether it be an item in a list or a property of a class that you want to track in multiple components, you will need to update the Store by sending an action to do so. Lets walk through an example.

updating a user example

In the user-edit component, after selecting a user and editing their details, we click the Update button in the UI to save the details of the user to the DB. The process will work as follows.


UI (Dumb Component)
Renders state on component
User Event (update user)
Selector
updates Shell observable with updated state
Shell (Smart Component)
Disapatch UpdateUser Action
Store
Sate
Action
Reducer updates Store
Reducer
new Action sent to reducer
calls userMaintenance service
updates database
Effects
Service
Server
Reducer sends the Action to Effects to make external call
Receives response and returns appropriate action (Success or Fail)

The **UI (Dumb Component)** will send a user event to the **Shell (Smart Component)** when the Update button is clicked as seen below with the *updateUser.emit(this.selectedFicUser)* method.

user-edit.component.ts

```
onSubmit(): void {
    this.getUserDetails();
    this.updateUser.emit(this.selectedFicUser);
    this.ConfirmationModal.close();
    this.ManageUserModal.close();
  }
```

The Smart component receives the event through the @Output() decorator and calls the updateUser($event) method. The FicUser is passed to the method inside the $event parameter.

user-maintenance-shell.component.html

```
<app-user-edit *ngIf="currentUser$ | async as selectedUser"
 class="pull-left"
    [selectedFicUser]="selectedUser"
    (updateUser)="updateUser($event)">
</app-user-edit>
```

The update user then dispatches an Action to update the user in the Store

user-maintenance-shell.component.ts

```
updateUser(user: FICUser): void {
    this.store.dispatch(new userActions.UpdateUser(user));
  }
```

We either listen for Actions in the Reducer or Effects. In this case, we need to make an API call so we send the UpdateUser Action to the Effects via the Reducer. The Effect will then use the UserMaintenanceService to update the user using the FicUser sent with the Action.

user.effects.ts

```
@Effect()
    updateUser$: Observable<Action> = this.actions$.pipe(
        ofType(userActions.UserActionTypes.UpdateUser),
        map((action: userActions.UpdateUser) => action.payload),
        mergeMap((user: FICUser) =>
            this.userMaintenanceService.updateUser(user).pipe(
                map(data => new userActions.UpdateUserSuccess(user)),
                catchError(err => of(new
userActions.UpdateUserFail(err)))
            )
        )
    );
```

First off, we want to listen for all actions being dispatched in our application via the injected this. actions$ observable and use an RxJS *pipe* to pass in our first operator. The *ofType* operator that we can get from NgRx effects and pass in a string of the action types we want to filter on. We use our UserActionTypes enum to use the UpdateUser to filter on. We will then use the map operator to grab the payload from the UpdateUser Action. Next we'll use the mergeMap operator to map over the emitted actions and return the result of calling our injected userMaintenanceService updateUser method, using another pipe to pass in a map operator to map over the results and get the response, which will be of type Boolean. We will then return a new UserActions. UpdateUserSuccess action along with the user we updated to add to the store. If an error occurs, we send a UserActions.UpdateUserFail action with the error message.

The new Action will be sent to the Reducer and the two cases in the switch statement for UpdateUserSuccess and UpdateUserFail are seen below.

user.reducer.ts

```
case UserActionTypes.UpdateUserSuccess:
 const updatedUsers = state.users.map(
     user => action.payload.cpNumber === user.cpNumber ? action.payload
: user
    );
    return {
     ...state,
        users: updatedUsers,
        currentUserCPNumber: action.payload.cpNumber,
        error: ''
 };

case UserActionTypes.UpdateUserFail:
 return {
     ...state,
        error: action.payload
    };
```

When UpdateUserSuccess Action is returned, we map through the list of users checking the cpNumber of the user we just updated. When we find the user, we replace it with the newly updated user in the action payload else leave the user unchanged. We use the map operator here because the reducer is a pure function. We cannot just change the user because the array that we pass to the store will still have the same reference. When we use the map operator, it returns a new array. We need to always keep in mind that State Management works with immutable objects or we will struggle with the *onChangeDetectionStrategy.OnPush* strategy in our components. In the return statement, we copy the current state object using the typescript spread operator along with the object *...state* and then update the properties that have changed namely the list of users amd the *currentUserCPNumber* and we set the error to an empty string. We set the *currentUserCPNumber* to the updated user so that the current selected user is the newly updated user.

If the UpdateUserFail Action gets returned, we copy the state using the spread operator, and update the error property with the error received from the NgRx Effects.

The store will not have the updated user and our UI should have the updated list of users. The reason for this has to do with selectors. In our Shell component, we have a users$ observable that continually checks for changes in the users property in the store. We use the rxjs *pipe* operator to call the NgRx select operator to select the users property. as seen below.

user-maintenance-shell.component.ts

```
ngOnInit() {
    this.store.dispatch(new userActions.LoadUsers());
    this.users$ = this.store.pipe(select(fromUser.getUsers));
    this.selectedUser$ =
this.store.pipe(select(fromUser.getCurrentUserCPNumber));
    this.errorMessage$ = this.store.pipe(select(fromUser.getError));
    this.cpNumberStatus$ =
this.store.pipe(select(fromUser.getCpNumberStatus));
    this.cpNumberToStatusCheck$ =
this.store.pipe(select(fromUser.getCpNumberToStatusCheck));
    this.currentUser$ =
this.store.pipe(select(fromUser.getCurrentUser));
  }
```

Inside the Index.ts file in the state folder of the userMaintenance feature is where we create our NgRx selectors. We firstly create a feature selector called *users* in the store and assign it to a const variable called *getUserMaintenanceFeatureState*. We then create a getUsers selector that makes use of the *getUserMaintenanceFeatureState* selector and selects the property users from the store as seen below.

index.ts

```
export interface State extends fromRoot.State {
    users: fromUsers.UserState;
}

const getUserMaintenanceFeatureState =
createFeatureSelector<fromUsers.UserState>('users');

export const getUsers = createSelector(
    getUserMaintenanceFeatureState,
    state => state.users
);
```

Now whenever the state changes, the users$ observable updates automatically. The users$ observable is passed through to the UI component (user-edit.component) via the async pipe in the user-maintenance-shell.component.html file and updated accordingly.

NgRx is quite complex until you understand the pattern. I advise that you watch the prerequisite course on PluralSight listed at the beginning of this page so that you can get an in-depth understanding.