INTERTECH

NgRx Tutorial 5-Part Series

Table of Contents

NgRx Code Review Checklist

NgRx Code Review Checklist

INTERTECH

Is NgRx setup properly?
Is the AppModule setting up the root reducers and effects properly? Same with feature modules? Are developers able to use Redux Devtools to view state? If you follow this guide, the answer to these questions will be "yes".

Is the code in the correct location in the folder structure?
Your organization should agree upon standards for where to put store-related code. I personally believe it should be under a "store" folder at the main level and for each feature module. Much easier to find code in this structure. See a picture of a good folder structure near the bottom of this blog.

Are the actions in their own file and organized according to best practices?
Each reducer should have a corresponding actions file that has the string constants, action class which derives from Action and a type of all available actions. See example here.

Is your new reducer added to the index.ts State and ActionReducerMap properly?
If you are creating a new reducer, make sure it is added to the main or feature State interface as well as the "reducers" **ActionReducerMap**. See setup here.

For the feature module State interface, does it extend the main State interface?
Feature modules reducers need to define a State interface that extends the main State interface so that the feature module has access to its own State and the main State. It's broken down here.

Are the selectors created for State properties properly?
State should be accessed via selectors. These selectors are created with the "createFeatureSelector" and "createSelector" functions from NgRx. See here and here for detailed information on selectors.

Are you retrieving parameters from the URL using NgRx?
It is very important to have one source of truth with NgRx and programming in general. If you are storing things like customerId in the store but it's always available in the URL, you're doing it wrong. Find details here.

Are you loading and updating the store from the API using effects?
It can be tempting just to call a service on the component and dispatch the results to store the data in NgRx but this is the wrong approach. It's best to use an effect to call the API and store the resulting data in State via another action that the reducer handles. The effect is called via a store dispatch of an action that the effect is listening for. For details, click here.

You've downloaded our NgRx tutorial series, a great first step to ensuring that your getting the most out of NgRx. The second step? Download our free NgRx code review checklist. A helpful resource to help you successfully review your NgRx code.

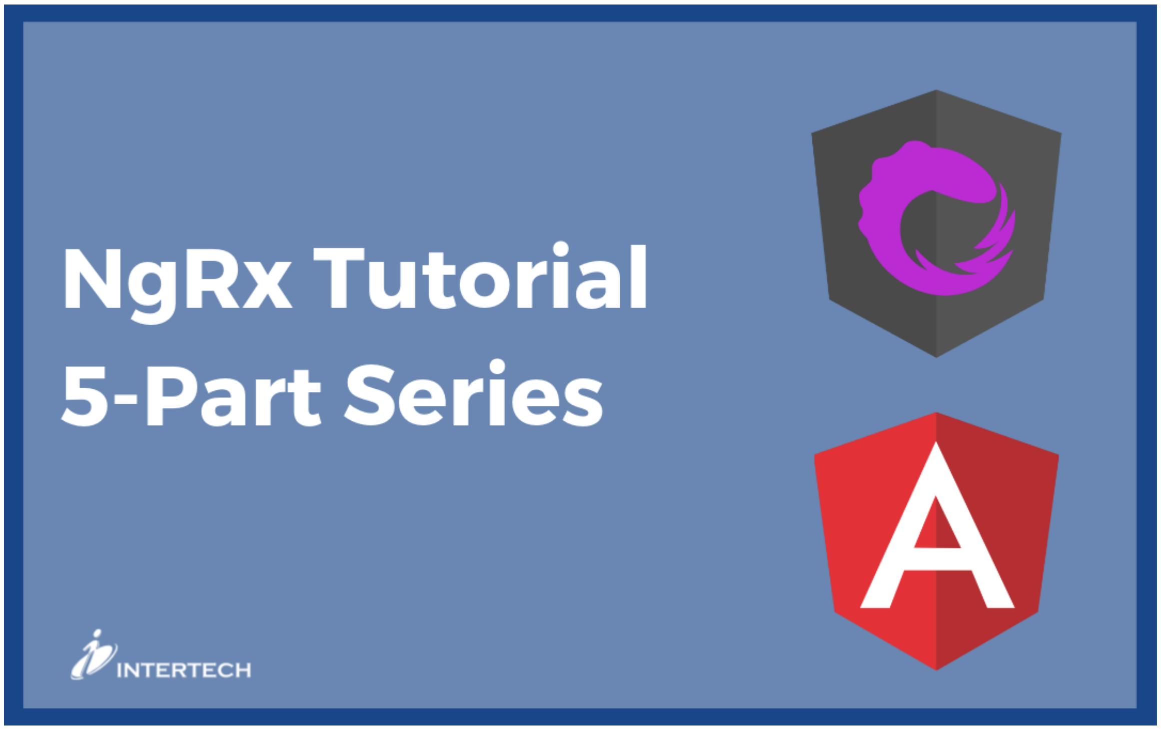
DOWNLOAD NOW

NgRx Free Project Consult

Serious about ensuring your NgRx project is on the right track? For a limited time, Rich Franzmeier, the author of the NgRx Tutorial Series and Checklist, will be offering short one-hour consults. Limited available slots. Visit the link below to learn what you can expect and to reserve your spot.

SIGN UP TODAY

Introduction



NgRx (Reactive Extensions for Angular) is becoming increasingly popular in the Angular community. It is a great way to manage state using the redux pattern and keep your application scalable.

This series, by Intertech developer Rich Franzmeier, is for beginners and those experienced with NgRx alike, walking readers through the basics of NgRx while also diving into more advanced concepts.

About Intertech

Founded in 1991, Intertech delivers technology training and software development consulting to Fortune 500, Government and Leading Technology institutions.

Learn more about us. Whether you are a developer interested in working for a company that invests in its employees or a company looking to partner with a team of technology leaders who provide solutions, mentor staff and add true business value, we'd like to meet you.

www.intertech.com

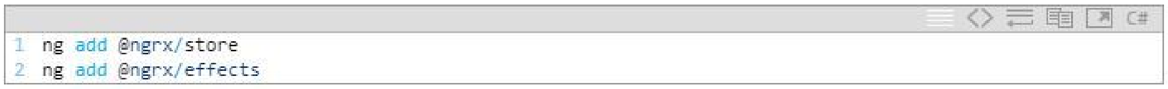
Chapter 1: Quickly Adding NgRx to Your Angular 6 Project

NgRx (Reactive Extensions for Angular) is becoming more and more popular in the Angular community. It is a great way to manage state using the redux pattern and keep your application scalable. When I first started using it, my biggest complaint was that it was a lot of typing basically the same thing over and over. Thankfully, the NgRx team addressed this with @ngrx/schematics. This package enhances the Angular CLI with new commands for NgRx. For example, to create a new actions file, simply type: **ng generate action ActionName**

This first chapter is meant to help you get up and running quickly with NgRx by using the @ngrx/schematics package. It is using NgRx version 6.01 so if it doesn't look the same to you, it could be things have changed. No knowledge of NgRx is necessary to read this post and setup your project!

UPDATE

Note: Use the "ng add" command to achieve most of the things done using schematics in this article. Run these commands for an existing project that doesn't have NgRx:



```
ng add @ngrx/store
ng add @ngrx/effects
```

After this completes, the only things necessary to do in this article will be the following:

- Setup – under here, you may want to install the rest of the ngrx tools
- NgRx Startup Code
- Store Command – under here, you'll have to add the "reducers" folder to a "store" folder to get the same structure as in the post
- Effect Command – under here you'll have to add the "effects" folder (which wasn't added for you) to a "store" folder to get the same structure as in the post

Setup

I start by creating a project using the standard Angular CLI command:

ng new ngrx-tutorial

This will give you the standard Angular starter project.

Now let's install the NgRx schematics package:

```
npm install @ngrx/schematics --save-dev
```

Install the rest of NgRx:

```
npm install @ngrx/store @ngrx/effects @ngrx/store-devtools @ngrx/router-store --save
```

NgRx Startup Code

Now that the project is setup and we have NgRx installed, we are ready to see the power of Schematics. Before we start running schematic commands, we will register it as the default collection in the Angular project by using this command:

ng config cli.defaultCollection @ngrx/schematics

Running this command adds the following to the angular.json file:

"cli": {

"defaultCollection": "@ngrx/schematics"

}

Store Command

Now we can run the 'store' command:

```
ng generate store State --root --statePath store/reducers --module app.module.ts
```

This command did the following:

- Created a 'store' folder at the app level
- Created a 'reducers' folder in the 'store' folder
- Created an 'index.ts' file in the 'reducers' folder
    - This is the main reducer file for the root store (as opposed to feature store(s)) which contains:
        - State interface definition (empty)
        - reducers – ActionReducerMap (empty)
        - metaReducers – MetaReducer<State>[] (empty)
- Imported the following to the AppModule:
    - StoreModule.forRoot(reducers, { metaReducers }),
        - Prepares the app for reducers and metaReducers
    - !environment.production?StoreDevtoolsModule.instrument() : []
        - Instruments the app for development

If I run "ng build" now, I get an error:

ERROR in src/app/app.module.ts(8,29): error TS2307: Cannot find module '../../../environments/environment'.

Ok, so the schematics aren't perfect. I think this is because I put the index.ts file two levels deep from 'app'. To fix this simply go to app.module and remove two levels of relative path for this import:

import { environment } from '../../../environments/environment';

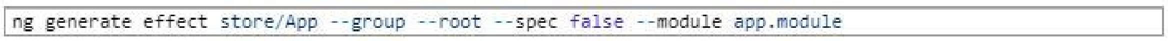
This should do the trick:

import { environment } from '../environments/environment';

Effect Command

The next command to run is the "effect" command. It's purpose is to get our first effect registered with the application so it is ready to go.

This is the command to run:



```
ng generate effect store/App --group --root --spec false --module app.module
```

This command did the following:

- Created an “effects” folder under “store”
  - I like to keep actions/effects/reducers under the “store” folder
- Added the app.effects.ts file in the “effects” folder
  - It is simply an effect with a constructor that has the Actions injected into it
- Imported the following to the AppModule:
  - EffectsModule.forRoot([AppEffects])
- Did not create a spec (unit test) file – just leave out “–spec false” if you want that file

Conclusion

Getting up and running with NgRx is now quicker than ever with the @ngrx/schematics package. Without knowing anything about NgRx, you can set up your project for it using best practice code so that your team is ready to start using NgRx. In the next chapter, I'll cover some of the other schematics commands to keep you productive as you code NgRx in Angular.

Chapter 2: Actions, Reducers and Effects

In the first chapter, I wrote about setting up NgRx in your Angular 6 application. Now it's time to focus on actions, reducers and effects. These are the heart and soul of your NgRx code and will be the ones you use most on a day to day basis. You will learn what they are, how to generate them, and how they work together in an Angular application.

The NgRx store is an implementation of the Redux pattern. Learn more about that here. Actions and reducers are a big part of the redux pattern. Effects are NgRx constructs to help with asynchronous operations.

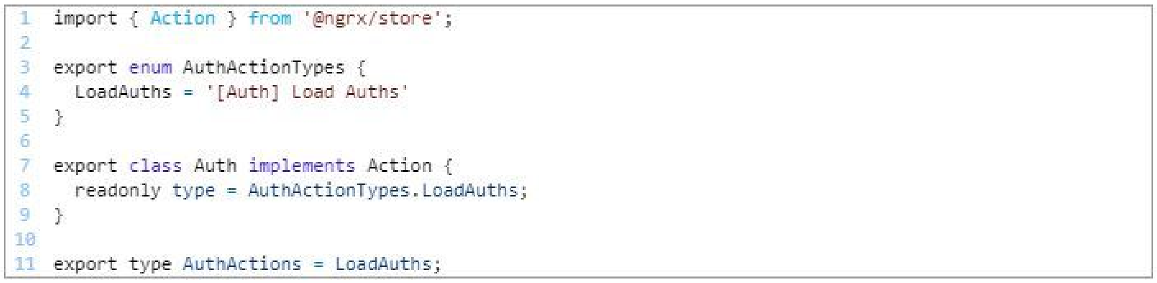
Actions

Actions are objects that extend the NgRx Action class with a 'type' property. They have an optional 'payload' property (naming is up to you but the standard is to name it 'payload') for sending in data to the effect/reducer and are dispatched by the store to either run an effect or change state in a reducer. So you can see that actions aren't all that complicated but NgRx schematics does generate action files for you and can help you standardize them in your project.

To generate an action file, run this command:

ng generate action store/actions/auth

This generates the following file:



```
import { Action } from '@ngrx/store';

export enum AuthActionTypes {
  LoadAuths = '[Auth] Load Auths'
}

export class Auth implements Action {
  readonly type = AuthActionTypes.LoadAuths;
}

export type AuthActions = LoadAuths;
```

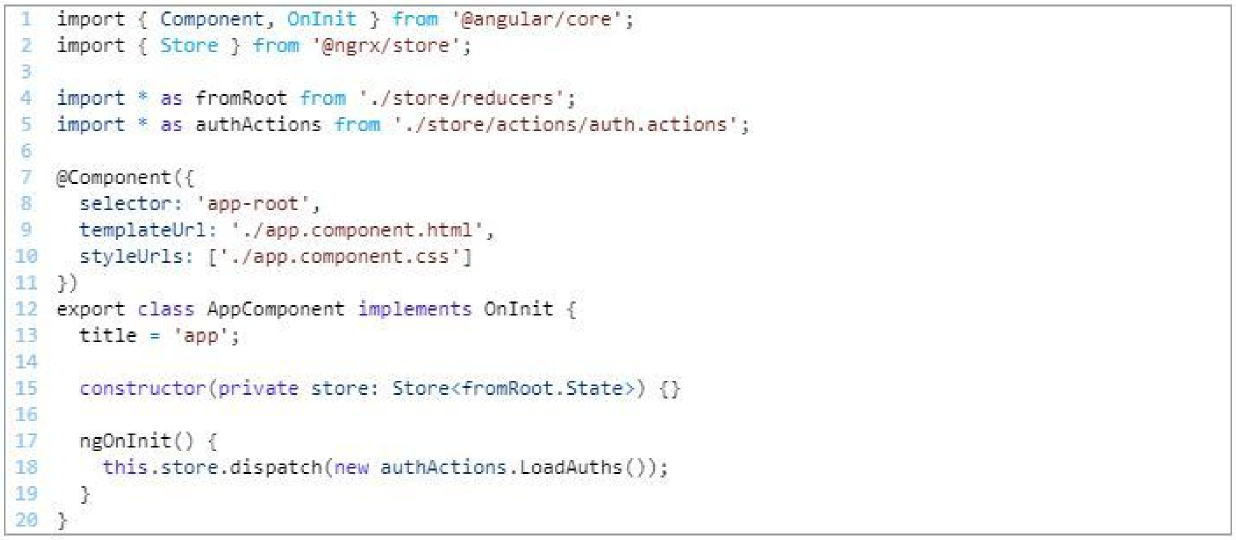
Notes:

- It generates a sample action
- Notice the error – LoadsAuths on the last line should be Auth (or the Auth action should be named LoadAuths – better yet)
    - This is an error with NgRx schematics but is no big deal as it gets you a template to follow
- The action constants are stored as an enum (AuthActionTypes)
- The action class has a type (you can add optional payload)
- The AuthActions type helps you to define all of your actions for Auth as a type – in the reducer you'll see why this is important

Dispatching the Action

Actions live to be dispatched. Reducers and effects just wait until an action is dispatched so they can do their job. But how are they dispatched?

To dispatch the action we just created, you would typically do that from your component. Here is the typical code you need:



```
import { Component, OnInit } from '@angular/core';
import { Store } from '@ngrx/store';

import * as fromRoot from './store/reducers';
import * as authActions from './store/actions/auth.actions';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent implements OnInit {
  title = 'app';

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

  ngOnInit() {
    this.store.dispatch(new authActions.LoadAuths());
  }
}
```

Notes:

- Imports:
    - Store from @ngrx/store
    - import * as fromRoot… – this is where the main State interface lives in the index.ts file
    - import * as authActions… – this is where our LoadAuths actions live

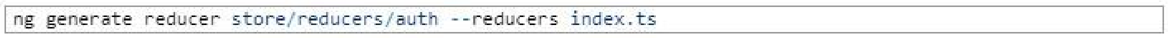
Notes (cont.):

- Inject the store
    - In the constructor, inject the store as shown in the code
- When it's time to dispatch (sometimes in ngOnInit, sometimes from a button click, etc.), run this command:
    - this.store.dispatch(new authActions.LoadAuths());

Reducers

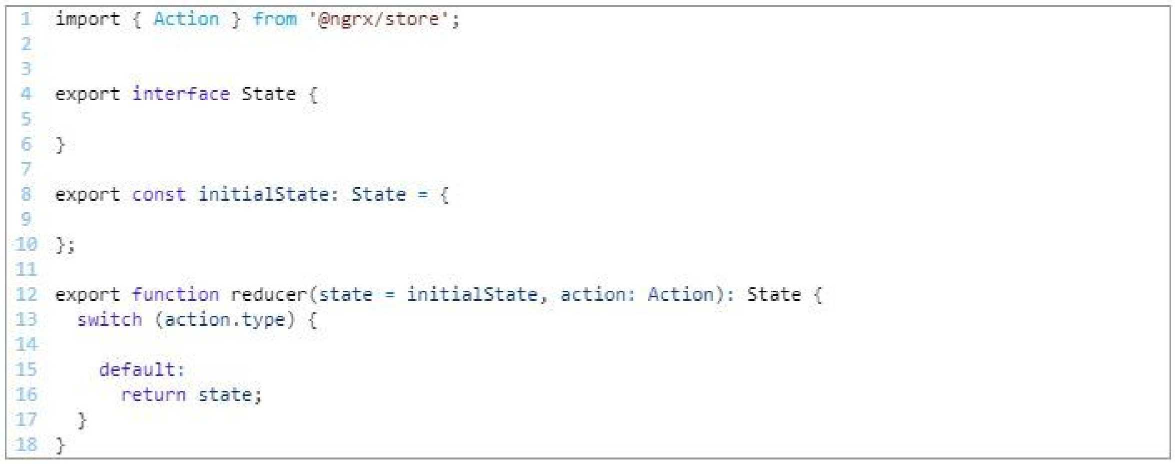
Reducers are pure functions that are the only ones that can change state. They aren't really changing state but making a copy of existing state and changing one or more properties on the new state.

To generate a reducer file, run this command:



```
ng generate reducer store/reducers/auth --reducers index.ts
```

This generates the following file:

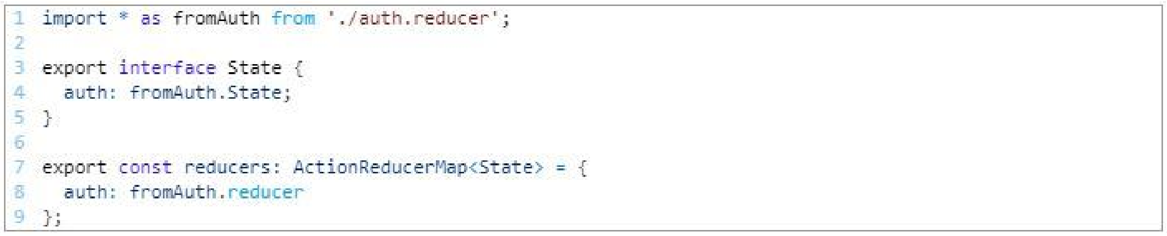


```
import { Action } from '@ngrx/store';


export interface State {

}

export const initialState: State = {

};

export function reducer(state = initialState, action: Action): State {
  switch (action.type) {

    default:
      return state;
  }
}
```

The reducer file adds:

- The State for the reducer – this state is added to the main state (see code below)
- Initial state which are your starting values
- A reducer function that will be added to the main reducer (see code below)

The main reducer file (index.ts) was changed to this:



```
import * as fromAuth from './auth.reducer';

export interface State {
  auth: fromAuth.State;
}

export const reducers: ActionReducerMap<State> = {
  auth: fromAuth.reducer
};
```

Notes:

- It added import shown above
- It added auth: fromAuth.State; to the State interface
- It added auth: fromAuth.reducer to the reducers constant

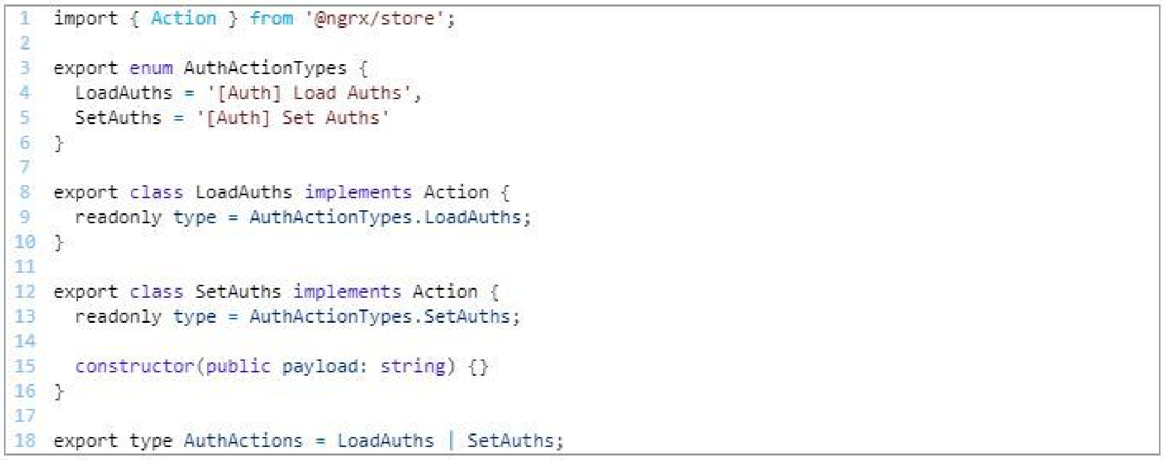
Add Action to Reducer

Now let's see how actions and reducers fit together.

This is what I'll do:

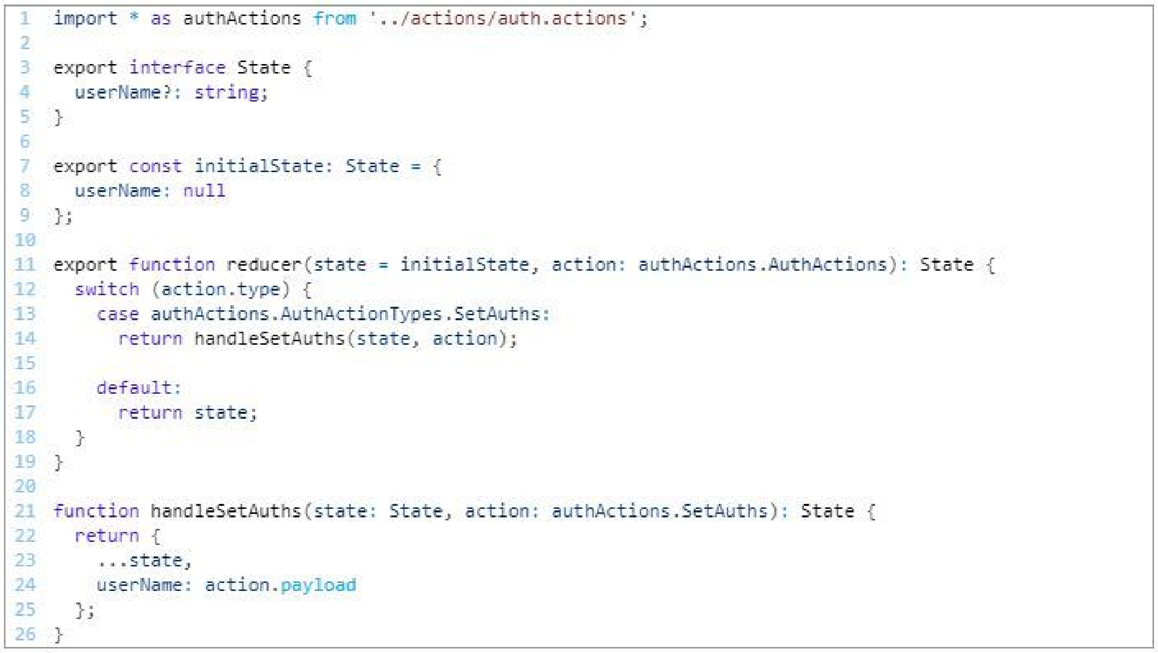
1. Add the 'userName' property to the auth reducer's State
2. Add a 'SetAuth' action which will set the userName property on State (payload is userName)
3. Update the reducer to handle this new action

Updated auth.actions.ts file:



```
import { Action } from '@ngrx/store';

export enum AuthActionTypes {
  LoadAuths = '[Auth] Load Auths',
  SetAuths = '[Auth] Set Auths'
}

export class LoadAuths implements Action {
  readonly type = AuthActionTypes.LoadAuths;
}

export class SetAuths implements Action {
  readonly type = AuthActionTypes.SetAuths;

  constructor(public payload: string) {}
}

export type AuthActions = LoadAuths | SetAuths;
```

Updated auth.reducer.ts file:



```
import * as authActions from '../actions/auth.actions';

export interface State {
  userName?: string;
}

export const initialState: State = {
  userName: null
};

export function reducer(state = initialState, action: authActions.AuthActions): State {
  switch (action.type) {
    case authActions.AuthActionTypes.SetAuths:
      return handleSetAuths(state, action);

    default:
      return state;
  }
}

function handleSetAuths(state: State, action: authActions.SetAuths): State {
  return {
    ...state,
    userName: action.payload
  };
}
```

Notes:

- The userName property is added to State and initialState (not necessary for initialState of course)
- In the 'reducer' function, the action is changed to authActions.AuthActions (which is the exported type AuthActions)
- The case statement is added for SetAuths
    - I like to add a function to handle each action so the switch doesn't get so huge and ugly
- The handleSetAuths function returns a new copy of state
    - The …state spread operator basically copies existing state
    - userName: action.payload then overwrites the userName property of State (which is the only one at this time, but more should be added)

Effects

Effects allow us to handle asynchronous operations in NgRx.

- Most times this will be calling an API
- The resulting data should be stored in state by returning an action for the reducer
- Effects always return one or more actions (unless you decorate @Effect with {dispatch: false})
- You can inject services into your effects as well so if you need to access those in NgRx, effects are the place to do it

To generate an effect file, run this command:

```
ng generate effect store/effects/auth --module app.module --root true
```

This generates the following file:

```
import { Injectable } from '@angular/core';
import { Actions, Effect } from '@ngrx/effects';


@Injectable()
export class AuthEffects {

  constructor(private actions$: Actions) {}
}
```

It also updates your app.module.ts file:

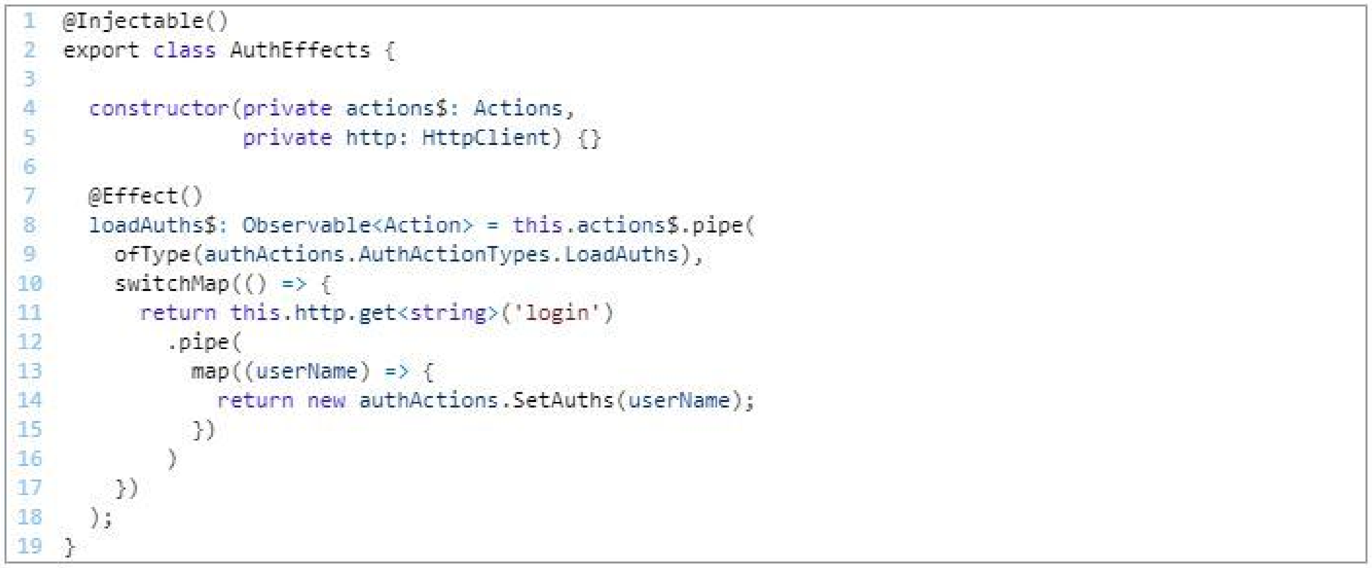
```
import { EffectsModule } from '@ngrx/effects';
import { AuthEffects } from './store/effects/auth.effects';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    StoreModule.forRoot(reducers, { metaReducers }),
    !environment.production ? StoreDevtoolsModule.instrument() : [],
    EffectsModule.forRoot([AuthEffects]),
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

The main thing the schematics generated here is EffectsModule.forRoot([AuthEffects]). This registers our new AuthEffects class with NgRx so that it starts to listen for dispatched actions.

Create an Effect

The generated effect file doesn't give you a skeleton effect to follow like the action file does, so I'll explain how you would do that here.

Here is the final effect (see notes below for explanation of how to create it):



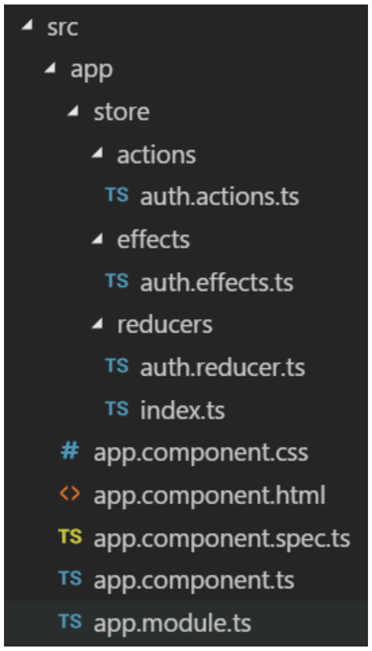
```
@Injectable()
export class AuthEffects {

  constructor(private actions$: Actions,
              private http: HttpClient) {}

  @Effect()
  loadAuths$: Observable<Action> = this.actions$.pipe(
    ofType(authActions.AuthActionTypes.LoadAuths),
    switchMap(() => {
      return this.http.get<string>('login')
        .pipe(
          map((userName) => {
            return new authActions.SetAuths(userName);
          })
        )
    })
  );
}
```

Notes:

- Decorate the effect with @Effect()
- Name the effect using camel case of the action name and end with $ to denote it is an Observable (loadAuths$)
    - The type of this variable should always be Observable<Action>
- The 'ofType' function is what is triggering this effect - whenever LoadAuths is dispatched as an action, this effect will run
    - Note it is using the string LoadAuths here and not the action class
- Use http to do whatever you need to do, in this case log the user in and return the user name
    - Return from the map a SetAuths action with the userName
        - This will automatically dispatch the SetAuths action to the reducer to update the userName on state
        - If you want to return multiple actions, return an array of actions

Project

So, over the last two chapters, this is how the project looks with NgRx:



```
src
  app
    store
      actions
        auth.actions.ts
      effects
        auth.effects.ts
      reducers
        auth.reducer.ts
        index.ts
    app.component.css
    app.component.html
    app.component.spec.ts
    app.component.ts
    app.module.ts
```

I like to have the actions, effects and reducers in their own folder under a 'store' folder so that they are easy to find. I have seen it done other ways - without a 'store' folder for example - but this is my preference. In a future chapter, I'm going to talk about feature modules so I'll start having one 'store' folder for each feature. So the pattern will reproduce itself many times in the project.

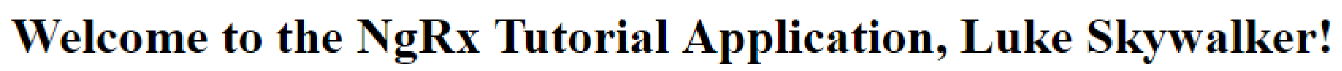
Conclusion

In this chapter, you learned how to generate actions, reducers and effects using NgRx schematics. You also learned what actions, reducers and effects are for and how they work together to help you to manage state using NgRx in your Angular 6 application. In the next chapter, I plan to show you how to access the state from your application.

Chapter 3: Accessing State in the Store

In this chapter, we continue our NgRx Tutorial by showing how to access state in the store by using selectors. It will complete the circle that started with dispatching an action, then an effect doing asynchronous work and finally the reducer updating state in the store. The missing piece is accessing that state in a component so that it can be displayed or used in various ways. For purposes of demonstration, I will be simulating a login in my 'auth' slice of state and store a user name and friendly name. This friendly name will be shown to the user on the welcome page. I'll of course use the Star Wars API to get the name.

This is the goal for the page output:



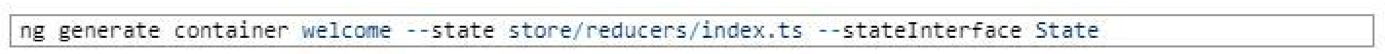
Welcome to the NgRx Tutorial Application, Luke Skywalker!

Code: github

Add Welcome Component

First I'll add a welcome component that will show the message. Of course use the @ngrx/schematics package to do it.

To generate a container, run this command:



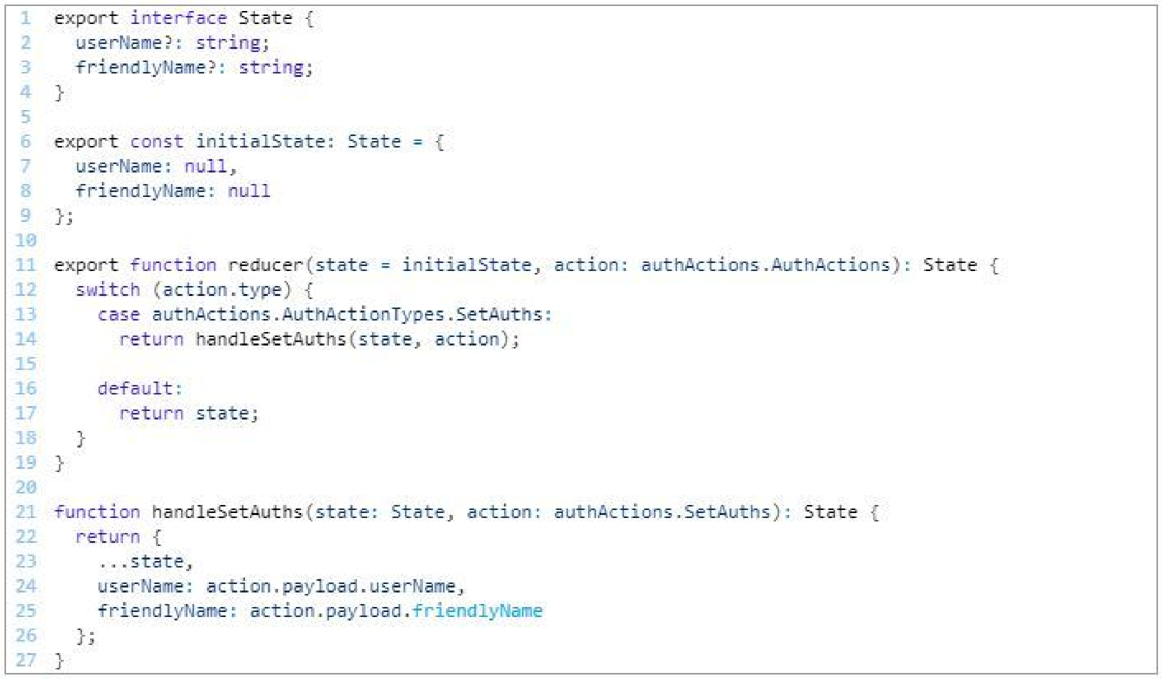
```
ng generate container welcome --state store/reducers/index.ts --stateInterface State
```

This basically is the same as **ng g component** but adds the store to the constructor:

We'll get back to this component later. First we need to understand how to access state from the store using selectors.

Add Selectors to the Auth Reducer

Before getting to selectors, I'm going to update the auth reducer as shown:



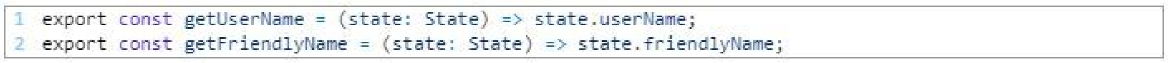
```
export interface State {
  userName?: string;
  friendlyName?: string;
}

export const initialState: State = {
  userName: null,
  friendlyName: null
};

export function reducer(state = initialState, action: authActions.AuthActions): State {
  switch (action.type) {
    case authActions.AuthActionTypes.SetAuths:
      return handleSetAuths(state, action);

    default:
      return state;
  }
}

function handleSetAuths(state: State, action: authActions.SetAuths): State {
  return {
    ...state,
    userName: action.payload.userName,
    friendlyName: action.payload.friendlyName
  };
}
```

Notes:

- Added 'friendlyName' to State – this will be shown on the welcome page
- Updated handleSetAuths to take in more than just userName as payload (actions were also updated with a SetAuthsPayload interface with userName and friendlyName as properties)

Auth Selectors

Now I'm ready to add selectors to this reducer. Selectors help us get at the data in the store by using pure functions and keeping most of the logic on the store instead of in the components. The first step with selectors is to add selectors in your reducer for each property of state as follows:



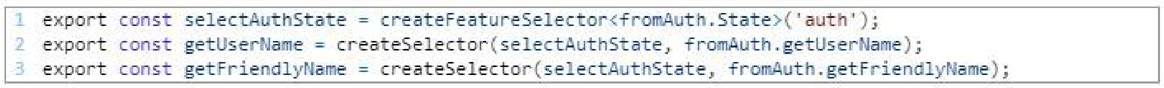
```
export const getUserName = (state: State) => state.userName;
export const getFriendlyName = (state: State) => state.friendlyName;
```

These are just pure functions that take a State parameter and return a value on state. This will be used in the main reducer file to create the selectors there.

Selectors in the Main Reducer (index.ts)

It’s convention to put the selectors that everyone uses in the main reducer. For feature modules, you’ll do it in the feature module’s main reducer (that’s a future post). These selectors are the way for the consumer (usually components) to access a slice of state in the store. It seems like a lot of busy work (and it is) but we do it this way so that it can be easily unit tested in one spot (the reducers) and easily consumed from the components.

Here are the auth selectors:



```
export const selectAuthState = createFeatureSelector<fromAuth.State>('auth');
export const getUserName = createSelector(selectAuthState, fromAuth.getUserName);
export const getFriendlyName = createSelector(selectAuthState, fromAuth.getFriendlyName);
```

Notes:

- createFeatureSelector and createSelector are imported from ‘@ngrx/store’
- selectAuthState: this creates a feature selector of type fromAuth.State (the state in the auth reducer)
    - The ‘auth’ string must match the property in state
    - This will be used to get the auth State for the upcoming createSelector functions
- getUserName: this creates a selector using selectAuthState and our getUserName selector we defined above
- getFriendlyName: this creates a selector using selectAuthState and our getFriendlyName selector we defined above

For a thorough look at selectors in NgRx, take a look at Todd Motto’s blog post on this topic.

Add Friendly Name to the Welcome Component

Now we are ready to access our friendly name that lives in auth state from our welcome component.

This is our updated welcome component:

```
export class WelcomeComponent implements OnInit {
  name$: Observable<string>;

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

  ngOnInit() {
    this.name$ = this.store.select(fromStore.getFriendlyName);
  }
}
```

Notes:

- First, define a 'name$' Observable<string> property
  - The html will reference this
- Now, populate the name$ with this.store.select(fromStore.getFriendlyName)
  - This is using our selector created in the main reducer
  - Whenever this value is updated, it will automatically update the component

This is the html portion:

```
<h1>Welcome to the NgRx Tutorial Application, {{name$ | async}}!</h1>
```

Notice the 'async' pipe. That is needed for Observables. It will handle the unsubscribing so there's no need for you to do it.

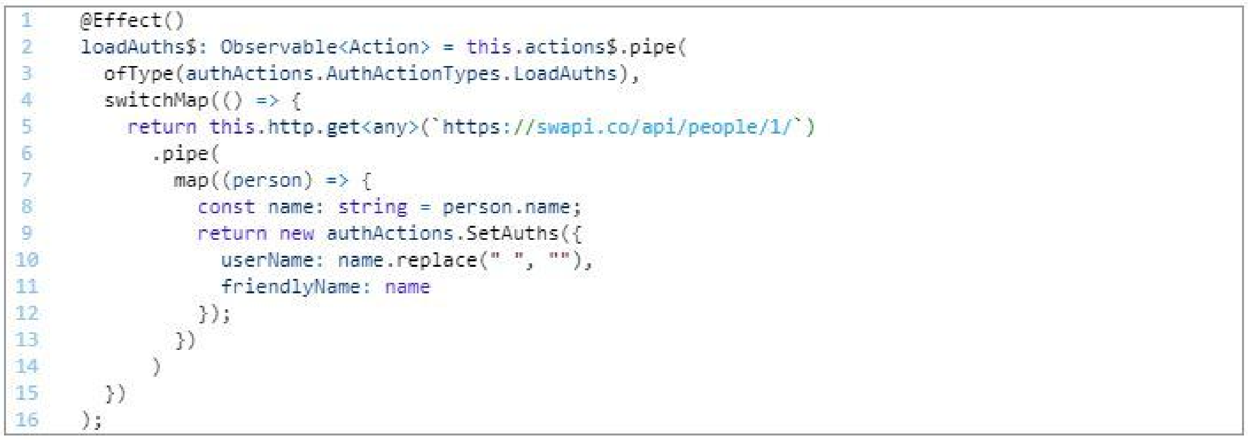
This code makes it super easy to access state and show it on your page. As an alternative, you could have accessed state in this way:

```
this.name$ = this.store.select((state) => state.auth.friendlyName);
```

This is problematic, though, because you have to do null checking and other work to ensure you don't get errors accessing the state. With the selector approach, that can be taken care of in a centralized place and heavily unit tested.

Update the Effect to get Star Wars Name

One last thing to do is update the LoadAuths effect so that it gets us person one from the Star Wars API, namely Luke Skywalker.



```
@Effect()
loadAuths$: Observable<Action> = this.actions$.pipe(
  ofType(authActions.AuthActionTypes.LoadAuths),
  switchMap(() => {
    return this.http.get<any>(`https://swapi.co/api/people/1/`)
      .pipe(
        map((person) => {
          const name: string = person.name;
          return new authActions.SetAuths({
            userName: name.replace(" ", ""),
            friendlyName: name
          });
        })
      )
  })
);
```

Notice how SetAuths is now passing an object with userName and friendlyName. This will call the SetAuths reducer and update state. Friendly name will finally appear on our welcome component.

Conclusion

Accessing state in NgRx is extremely important and is what the components (mainly) consume. It is done using state selectors that we define in the reducers. These selectors access a slice of state and should be fully unit tested. With the knowledge gained in these first three chapters, the reader should be able now to develop an Angular app using NgRx for state management. The final piece of this puzzle is setting up feature modules with their own state and all that goes with that. That will be tackled in upcoming chapters.

Chapter 4: Add State to Feature Module

Feature modules are Angular modules that group a bunch of components/services, etc. into a module. This allows us to separate concerns nicely. Most likely, you've dealt with feature modules if you have 1 month experience in Angular. Since feature modules can be lazy loaded and because they are separate from the main app module, NgRx does things a little differently. In this chapter, I plan to describe how to setup a lazy loaded feature module with NgRx.

The code is on github.

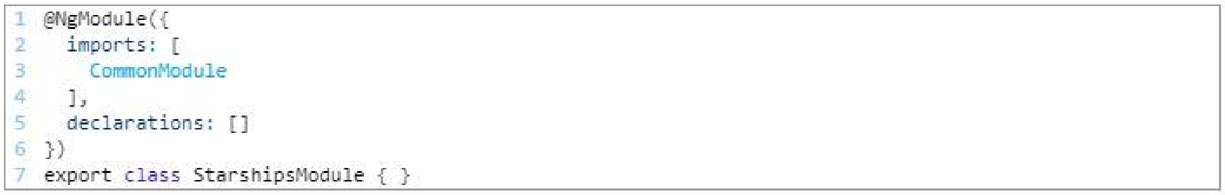
Create a Feature Module

Staying with the Star Wars API theme, I'll create a "Starships" feature module that will display a list of the first ten starships in the Star Wars universe.

This is the Angular CLI command to create the module:

ng g module starships

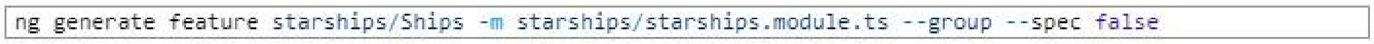
This creates the following module code:



```
@NgModule({
  imports: [
    CommonModule
  ],
  declarations: []
})
export class StarshipsModule { }
```

Create the NgRx Feature Module Code

Before creating the component in the module to show the starships, I'm going to generate the NgRx actions/effects/reducers using the following NgRx Schematics command:



```
ng generate feature starships/Ships -m starships/starships.module.ts --group --spec false
```

This does the following:

- Creates actions, effects and reducers folders with their respective files
    - ships.actions.ts, ships.effects.ts, ships.reducer.ts
- Updates the starships module we created first thing as shown:

```
@NgModule({
  imports: [
    CommonModule,
    StoreModule.forFeature('starships', fromShips.reducer),
    EffectsModule.forFeature([ShipsEffects])
  ],
  declarations: []
})
export class StarshipsModule { }
```

The following were added:

- StoreModule.forFeature('starships', fromShips.reducer)
- EffectsModule.forFeature([ShipsEffects])

So you can see that our feature module has it's own module setup but "forFeature" instead of "forRoot" as in the app.module.

Also note that instrumentation wasn't setup in the feature module, that is only required in the root module (app.module). For a refresher, this is the setup of instrumentation in the app.module imports:

```
!environment.production ? StoreDevtoolsModule.instrument() : []
```

This gives us the ability to view state from a Chrome browser extension called Redux DevTools.

I'm also going to move the actions, effects and reducers folders under a "store" folder. I do this so it's easier to find all things NgRx within the feature module.

Last thing is to fix up the actions file because this is how it was generated:

```
import { Action } from '@ngrx/store';

export enum ShipsActionTypes {
  LoadShipss = '[Ships] Load Shipss'
}

export class Ships implements Action {
  readonly type = ShipsActionTypes.LoadShipss;
}

export type ShipsActions = LoadShipss;
```

As you can see there are some problems here.

- LoadShipss – problem with plurality
- Ships action should be LoadShips

Reorganize the Starships Reducer

The way it's setup now, the ships reducer is setup to handle only one reducer in the feature module. That's obviously not ideal so a little reorganization is in order.

First, I add some state to the ships reducer:

```
export interface State {
  allShips: any[];
}

const initialState: State = {
  allShips: []
};
```

Now I'm going to create an index.ts file in the starships/store/reducers folder:

```
import {
  createSelector,
  createFeatureSelector,
  ActionReducerMap,
} from '@ngrx/store';

import * as fromShips from './ships.reducer';
import * as fromRoot from '../../../store/reducers';

export interface StarshipsState {
  ships: fromShips.State;
}

export interface State extends fromRoot.State {
  ships: StarshipsState;
}

export const reducers: ActionReducerMap<StarshipsState> = {
  ships: fromShips.reducer
};
```

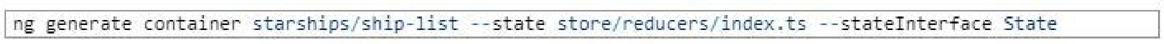
These are the important points:

- Created interface **StarshipsState** – this will hold all of the different states in the different reducers we may create in this feature module
- Created an **ActionReducerMap** of StarshipsState – this will define the reducers that correspond to the properties in StarshipsState
- Created interface **State** that extends root or main State – this is the one that will be referred to in our components in this feature module and will allow us to see state from the app.module level
  - Note you still can't see state from other feature modules
  - We won't add any more code to this state as all additional state/reducers will be added to StarshipsState

Create Component to Show Starships

Now we are ready to create the component to show the starships and inject the store into the component.

I'll use this NgRx schematics command to generate a container:

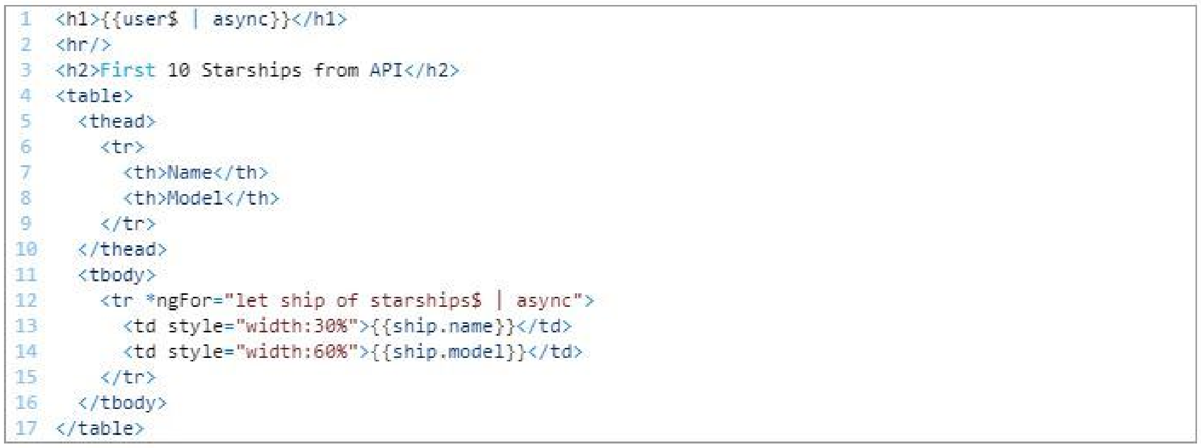


```
ng generate container starships/ship-list --state store/reducers/index.ts --stateInterface State
```

This does the following:

- Creates the ship-list component
    - Injects the store into the component (using our State that extends root State)
- Adds the component to our starships.module declaration

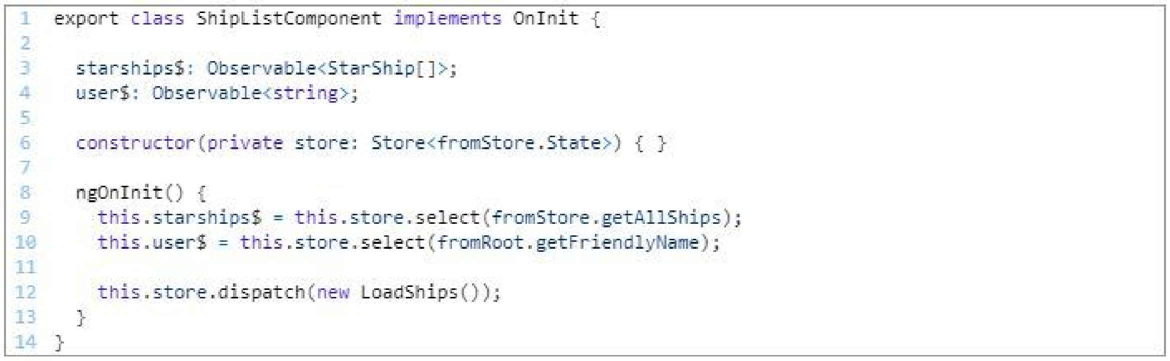
Here is the HTML for the component:



```html
<h1>{{user$ | async}}</h1>
<hr/>
<h2>First 10 Starships from API</h2>
<table>
  <thead>
    <tr>
      <th>Name</th>
      <th>Model</th>
    </tr>
  </thead>
  <tbody>
    <tr *ngFor="let ship of starships$ | async">
      <td style="width:30%">{{ship.name}}</td>
      <td style="width:60%">{{ship.model}}</td>
    </tr>
  </tbody>
</table>
```

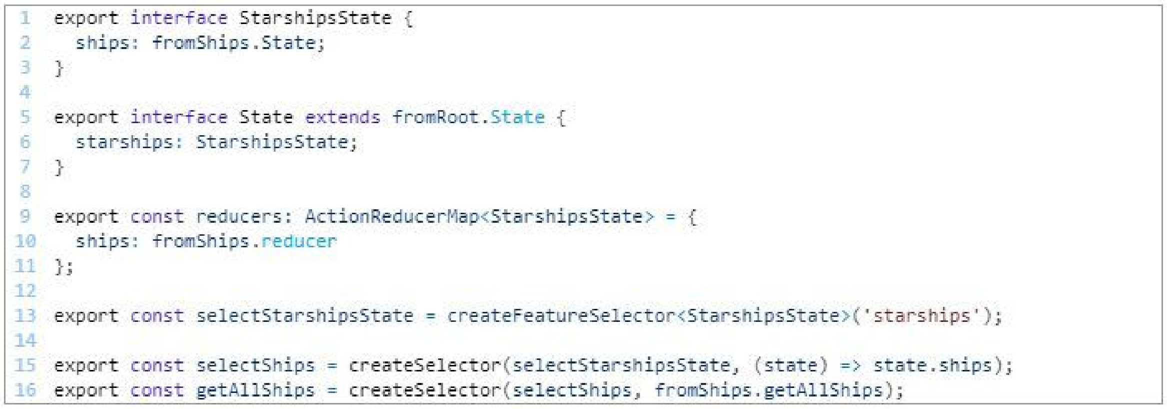
- The user$ variable is there to use the root part of state that we previously had retrieved (Luke Skywalker)
- The starships$ variable holds the results of state so initially it will be empty and then it will load once state changes

And then the component code:



```
export class ShipListComponent implements OnInit {

  starships$: Observable<StarShip[]>;
  user$: Observable<string>;

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

  ngOnInit() {
    this.starships$ = this.store.select(fromStore.getAllShips);
    this.user$ = this.store.select(fromRoot.getFriendlyName);

    this.store.dispatch(new LoadShips());
  }
}
```

- Dispatch the LoadShips action to load the ships using the Star Wars API in the effect (see code below)
- Use the fromStore.getAllShips to get the ships from our feature module…here's how this is setup in the feature module index.ts reducer:

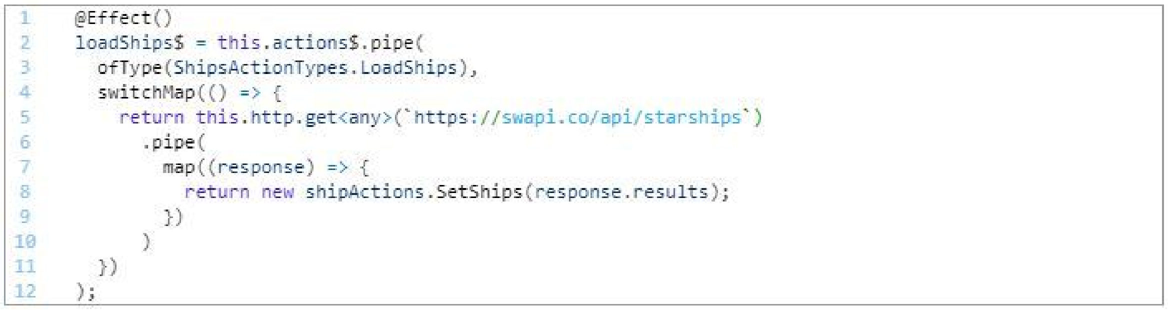


```
export interface StarshipsState {
  ships: fromShips.State;
}

export interface State extends fromRoot.State {
  starships: StarshipsState;
}

export const reducers: ActionReducerMap<StarshipsState> = {
  ships: fromShips.reducer
};

export const selectStarshipsState = createFeatureSelector<StarshipsState>('starships');

export const selectShips = createSelector(selectStarshipsState, (state) => state.ships);
export const getAllShips = createSelector(selectShips, fromShips.getAllShips);
```

- The 'starships' in the State interface must match the 'starships' in the selectStarshipsState selector
    - And it must match the forFeature declaration in the feature module
- Note that the "createFeatureSelector" command doesn't match (as of 8/18/2018) what is in the NgRx example app
    - That's because it is a future thing that we don't yet have access to (see Stack Overflow)
- selectShips gets the 'ships' property of StarshipsState
- getAllShips then just gets the allShips property of fromShips.State

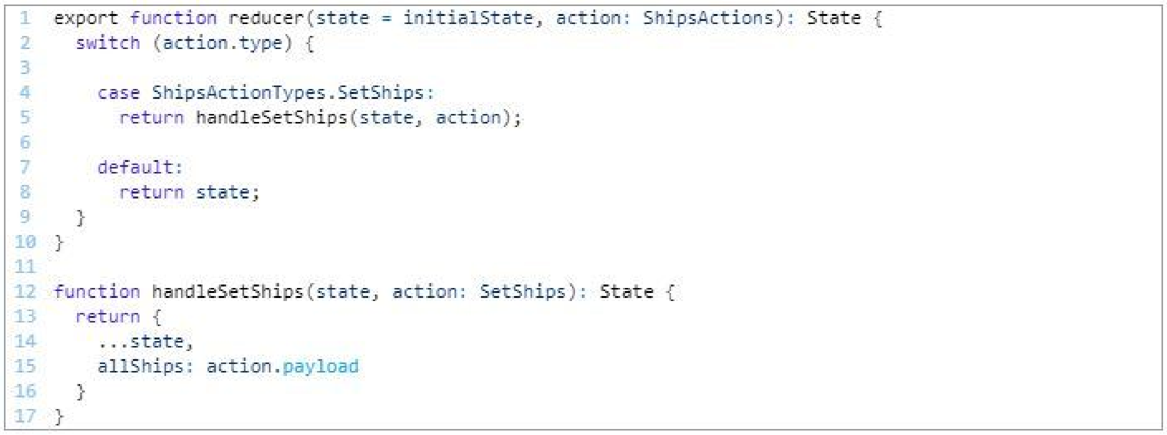
LoadShips Action in the Effect

Here is the LoadShips action in the effect (which will call the Star Wars API to get the first ten ships):



```
  @Effect()
  loadShips$ = this.actions$.pipe(
    ofType(ShipsActionTypes.LoadShips),
    switchMap(() => {
      return this.http.get<any>(`https://swapi.co/api/starships`)
        .pipe(
          map((response) => {
            return new shipActions.SetShips(response.results);
          })
        )
    })
  );
```

- SetShips will then call the reducer function to update state as shown in ships.reducer:

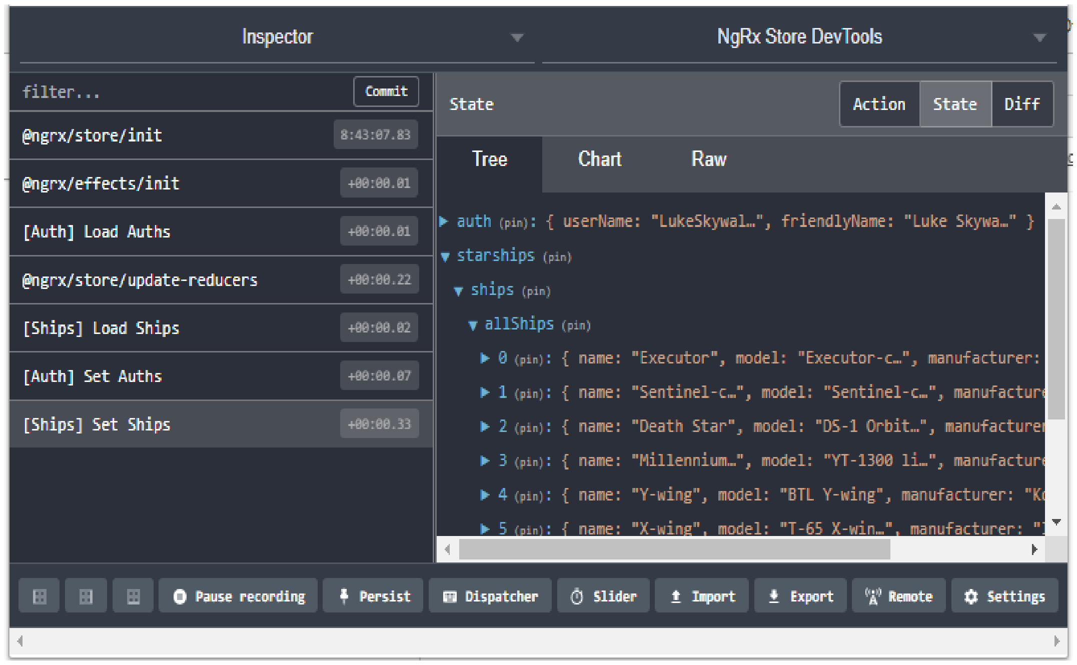


```
export function reducer(state = initialState, action: ShipsActions): State {
  switch (action.type) {

    case ShipsActionTypes.SetShips:
      return handleSetShips(state, action);

    default:
      return state;
  }
}

function handleSetShips(state, action: SetShips): State {
  return {
    ...state,
    allShips: action.payload
  }
}
```

Redux DevTools to View State

I mentioned Redux DevTools, a Chrome extension, earlier. Let's take a look at it now. It is a useful tool when developing to see what is actually in state and the actions that got it that way.

Here is a screenshot of it for when we view the starships:



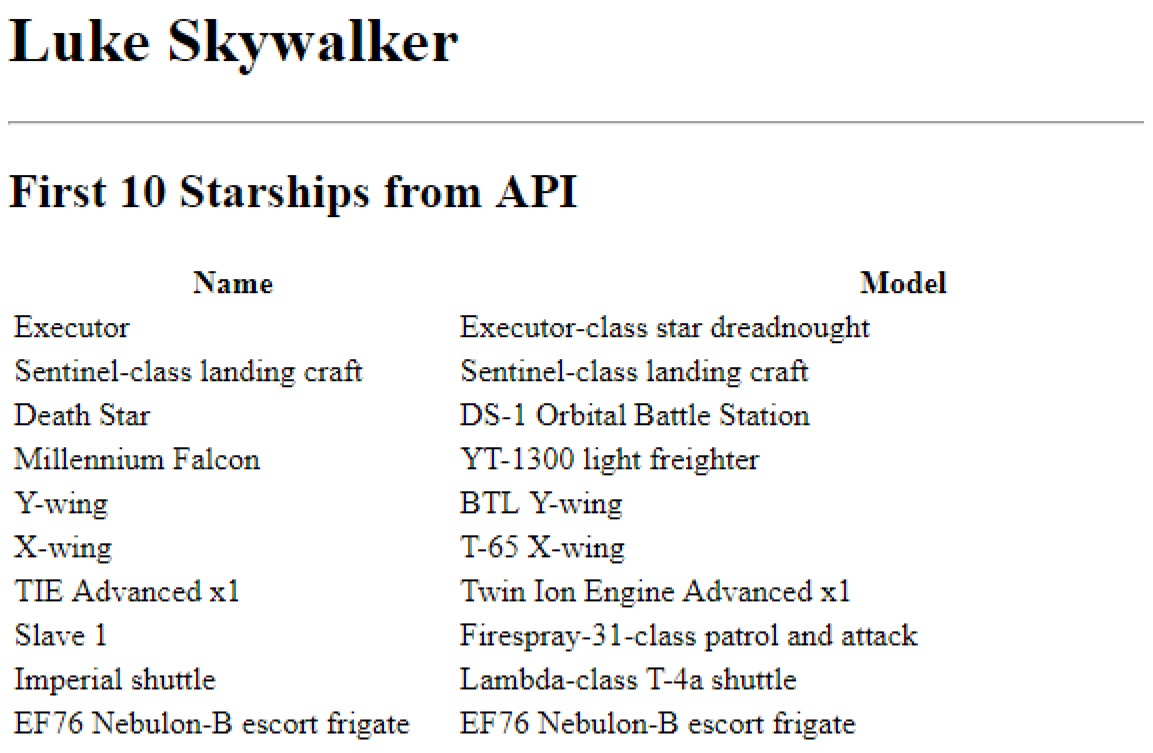
```
@ngrx/store/init                8:43:07.83
@ngrx/effects/init              +00:00.01
[Auth] Load Auths               +00:00.01
@ngrx/store/update-reducers     +00:00.22
[Ships] Load Ships              +00:00.02
[Auth] Set Auths                +00:00.07
[Ships] Set Ships               +00:00.33
```

```
▶ auth (pin): { userName: "LukeSkywal…", friendlyName: "Luke Skywa…" }
▼ starships (pin)
  ▼ ships (pin)
    ▼ allShips (pin)
      ▶ 0 (pin): { name: "Executor", model: "Executor-c…", manufacturer:
      ▶ 1 (pin): { name: "Sentinel-c…", model: "Sentinel-c…", manufactur
      ▶ 2 (pin): { name: "Death Star", model: "DS-1 Orbit…", manufacturer
      ▶ 3 (pin): { name: "Millennium…", model: "YT-1300 li…", manufactur
      ▶ 4 (pin): { name: "Y-wing", model: "BTL Y-wing", manufacturer: "K
      ▶ 5 (pin): { name: "X-wing", model: "T-65 X-win…", manufacturer: "
```

- Notice that we have selected "[Ships] Set Ships", that's the action that calls the reducer to change state
    - Load Ships is called before – that one got the ships from the API
- "starships" is our feature name
- "ships" is the property on StarshipsState
- "allShips" is the property on the ships.reducer State

How the Page Looks

If you are curious, here is my award-winning web page for showing the starships:



Luke Skywalker

First 10 Starships from API

Name	Model
Executor	Executor-class star dreadnought
Sentinel-class landing craft	Sentinel-class landing craft
Death Star	DS-1 Orbital Battle Station
Millennium Falcon	YT-1300 light freighter
Y-wing	BTL Y-wing
X-wing	T-65 X-wing
TIE Advanced x1	Twin Ion Engine Advanced x1
Slave 1	Firespray-31-class patrol and attack
Imperial shuttle	Lambda-class T-4a shuttle
EF76 Nebulon-B escort frigate	EF76 Nebulon-B escort frigate

Conclusion

So that was quite a lot! But we were able to do the following:

1. Add a lazy loaded feature module (see the code on github for how to lazy load a module)
2. Add NgRx state to it
3. Prepare for future expansion within the feature module by reorganizing reducer code
4. Create a cool component using NgRx state
5. View the contents of the state in the browser using Redux DevTools

After these four chapters, you should be getting pretty good at using NgRx to manage state in an Angular application. I'm sure most projects are using feature modules these days and being able to manage state with NgRx is a valuable skill to learn.

Chapter 5: Add Router Info to State

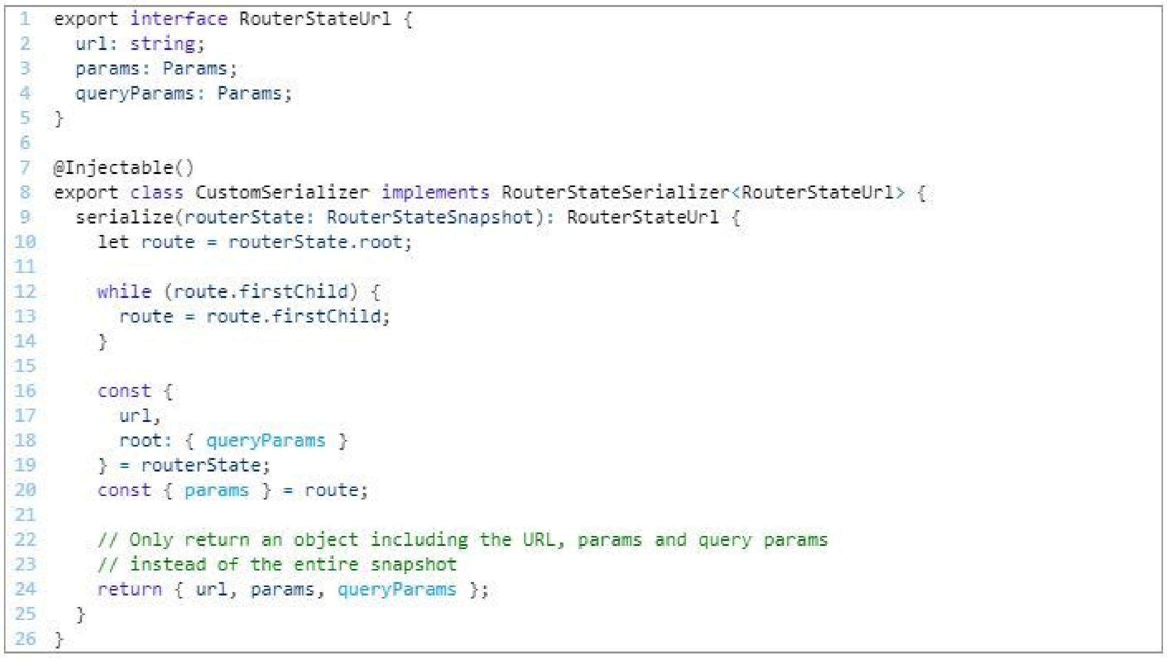
Adding Angular router information to state is very important for NgRx. Why, though, does it matter if the Url, parameters and query parameters are stored in state? Good question! This chapter aims to answer the question. I'll be building upon the code that has been written in my previous four posts on NgRx. I plan to add a starship detail page to the app which features the Star Wars API.

The code is on github.

Add Custom Router Serializer

The first thing I'll do is get the router information into state. There is some pretty boilerplate code that we'll add to our root reducer to make that happen. You can find it in the NgRx docs here. Note that the code there isn't perfect and I'll point that out as we go.

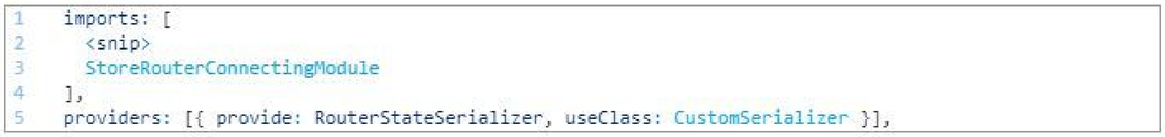
In our app/store/reducers/index.ts file, let's add this code:



```
export interface RouterStateUrl {
  url: string;
  params: Params;
  queryParams: Params;
}

@Injectable()
export class CustomSerializer implements RouterStateSerializer<RouterStateUrl> {
  serialize(routerState: RouterStateSnapshot): RouterStateUrl {
    let route = routerState.root;

    while (route.firstChild) {
      route = route.firstChild;
    }

    const {
      url,
      root: { queryParams }
    } = routerState;
    const { params } = route;

    // Only return an object including the URL, params and query params
    // instead of the entire snapshot
    return { url, params, queryParams };
  }
}
```

Notes:

- RouterStateUrl is an interface which defines what we want to store about the router
    - It can be anything that is on RouterStateSnapshot (however, there are some things on RouterStateSnapshot that are not immutable and will break NgRx store freeze)
- The CustomSerializer class will basically be the code that copies things from the RouterStateSnapshot into our RouterStateUrl interface and returns that
    - To get this code to run, you'll have to add the following code to the app.module:

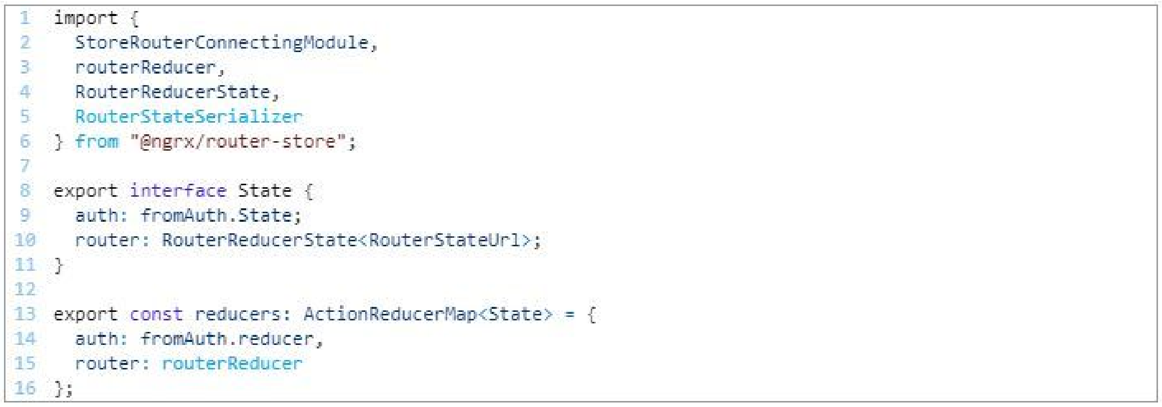


```
imports: [
  <snip>
  StoreRouterConnectingModule
],
providers: [{ provide: RouterStateSerializer, useClass: CustomSerializer }],
```

- Import the StoreRouterConnectingModule
    - **Note that this is where the NgRx docs' code has it wrong** (at least as of 9/9/2018) – there is no need for the forRoot({serializer: RouterStateSerializer}) – in fact it won't compile if you have it
- Provide the RouterStateSerializer to be our CustomSerializer

Add the Router Info to State

Now that the CustomSerializer is in place, we are ready to add it to state. Back in the app/store/reducers/index.ts file, we'll add a new 'router' property to State and reducers:



```
import {
  StoreRouterConnectingModule,
  routerReducer,
  RouterReducerState,
  RouterStateSerializer
} from "@ngrx/router-store";

export interface State {
  auth: fromAuth.State;
  router: RouterReducerState<RouterStateUrl>;
}

export const reducers: ActionReducerMap<State> = {
  auth: fromAuth.reducer,
  router: routerReducer
};
```

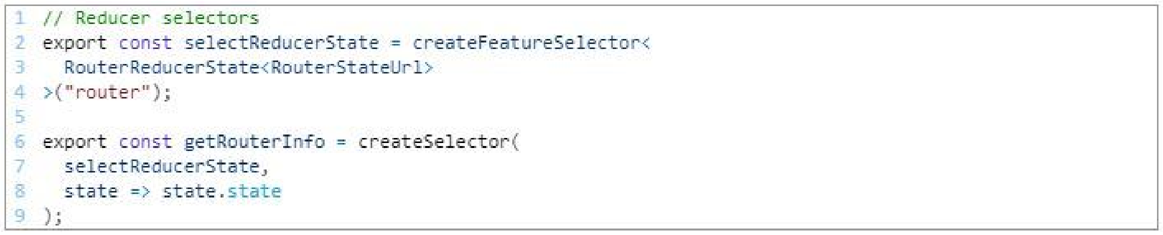
Notes:

- Add the “router” property to the State interface
    - RouterReducerState is imported from @ngrx/router-store
    - The generic piece here is our RouterStateUrl interface defined previously
- Add the “router” property to the ActionReducerMap
    - routerReducer is imported from @ngrx/router-store

Add Router Info Selectors

Lastly, I’m going to add selectors so we can access this router info in our application. Without this piece, at this point, we would still have a win by putting router info in the state because it gives us the ability to do time-traveling using the Redux DevTools in Chrome. I’ll show you this later on.

Here are the selectors in the app/store/reducers/index.ts file:



```
// Reducer selectors
export const selectReducerState = createFeatureSelector<
  RouterReducerState<RouterStateUrl>
>("router");

export const getRouterInfo = createSelector(
  selectReducerState,
  state => state.state
);
```

- selectReducerState creates our feature selector “router” which matches the property name on State and the reducers map
- getRouterInfo returns the RouterStateUrl interface we defined initially with the url, params and queryParams – which is what we care most about

And that’s all there is to getting router information into state for our application. Be patient on this, it seems like there is no reason for this but there is a big payoff for this coming (besides the time-traveling.)

Create a Ship Detail Component

Now that we have the router info stored in state, it’s time to build a component that will use it and get our big payoff that’s been promised.

To create the ship detail component, run this command:

```
ng generate container starships/ship-detail --state store/reducers/index.ts --stateInterface State
```

- This created a component with the store injected into the constructor nicely
- It also defined it in the starships module

I'll add this HTML code:

```html
<h1>Ship Detail</h1>
<div *ngIf="(starShip$ | async) != null">
  <h2>{{(starShip$ | async).name}}</h2>
  <table>
    <tr>
      <td width="20%"><strong>Manufacturer</strong></td>
      <td width="50%">{{(starShip$ | async).manufacturer}}</td>
    </tr>
    <tr>
      <td><strong>Cost In Credits</strong></td>
      <td>{{(starShip$ | async).cost_in_credits}}</td>
    </tr>
    <tr>
      <td><strong>Length</strong></td>
      <td>{{(starShip$ | async).length}}</td>
    </tr>
    <tr>
      <td><strong>Crew</strong></td>
      <td>{{(starShip$ | async).crew}}</td>
    </tr>
    <tr>
      <td><strong>Passengers</strong></td>
      <td>{{(starShip$ | async).passengers}}</td>
    </tr>
    <tr>
      <td><strong>Class</strong></td>
      <td>{{(starShip$ | async).starship_class}}</td>
    </tr>
  </table>
</div>
```

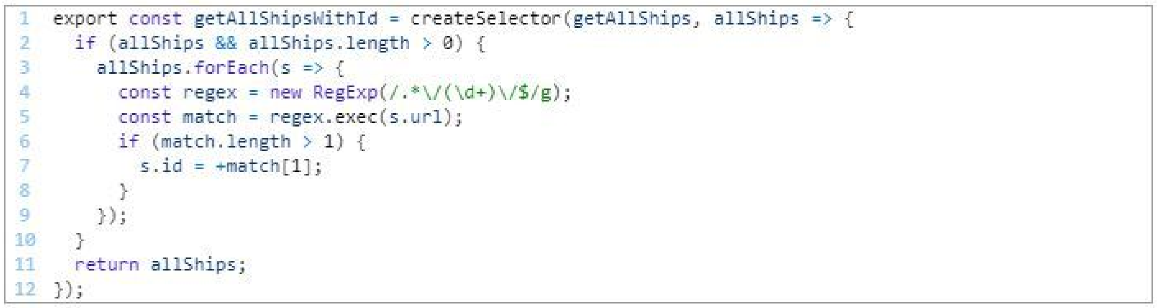
And the component TypeScript:

```typescript
import { Component, OnInit } from "@angular/core";
import { Store } from "@ngrx/store";
import { Observable } from "rxjs";

import * as fromStore from "../store/reducers";
import { StarShip } from "../../models/star-ship.model";

@Component({
  selector: "app-ship-detail",
  templateUrl: "./ship-detail.component.html",
  styleUrls: ["./ship-detail.component.css"]
})
export class ShipDetailComponent implements OnInit {
  starShip$: Observable<StarShip>;

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

  ngOnInit() {
    this.starShip$ = this.store.select(fromStore.getCurrentShip);
  }
}
```

This is all there is to the code! It works for any ship that is selected from the list of ships. The magic here is in the fromStore.getCurrentShip selector. Let's take a closer look at that.

The Starship Selectors

Before looking at the **getCurrentShip** selector, I have to explain the other one I created: the **getAllShipsWithId** selector.

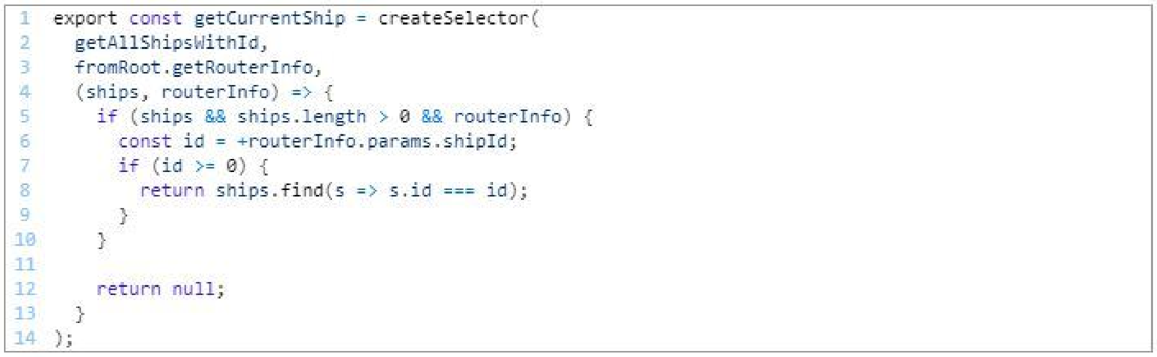


```
export const getAllShipsWithId = createSelector(getAllShips, allShips => {
  if (allShips && allShips.length > 0) {
    allShips.forEach(s => {
      const regex = new RegExp(/.*\/(\d+)\/$/g);
      const match = regex.exec(s.url);
      if (match.length > 1) {
        s.id = +match[1];
      }
    });
  }
  return allShips;
});
```

- The first arg in the createSelector function is the "getAllShips" selector – it returns all of the starships in the store
- The second arg "allShips" will take the ships passed in from "getAllShips" and use a regular expression to grab the starship id from the url property on the starship
- This is necessary so that it will be easy to find a starship by its id later

Until now, selectors have been pretty boring, but here you can see what else you can do with them

Here is the **getCurrentShip** selector and our payoff with the router information:



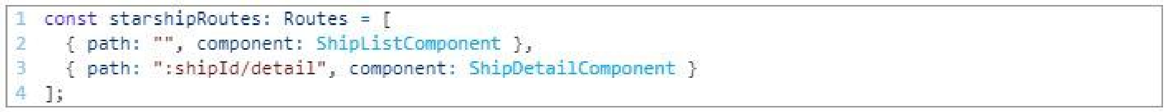
```
export const getCurrentShip = createSelector(
  getAllShipsWithId,
  fromRoot.getRouterInfo,
  (ships, routerInfo) => {
    if (ships && ships.length > 0 && routerInfo) {
      const id = +routerInfo.params.shipId;
      if (id >= 0) {
        return ships.find(s => s.id === id);
      }
    }

    return null;
  }
);
```

Notes:

- Note that this works because the list of ships we get from the Star Wars API has the details in it, so there is no need for another API call
    - If your code needs to get more detailed info, you can of course do that in an effect

- The first arg of the createSelector function is the "getAllShipsWithId" selector – so we can easily match the id with the id in the router info
- The second arg is the fromRoot.getRouterInfo selector which gives us the router info
- The third arg takes both of these and finds the ship by id using the routerInfo.params.shipId
    - Note that when the shipId changes, this also changes our selector and the component updates!
        - So for example, you could have a previous/next set of buttons that allowed you to navigate up and down the list w/o going back to the grid, this code would execute whenever the route was navigated to

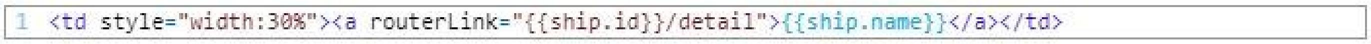
    - For this shipId parm to make sense, let's look at the Starships routing module:



```
const starshipRoutes: Routes = [
  { path: "", component: ShipListComponent },
  { path: ":shipId/detail", component: ShipDetailComponent }
];
```

- :shipId is added to our params object in the RouterStateUrl interface

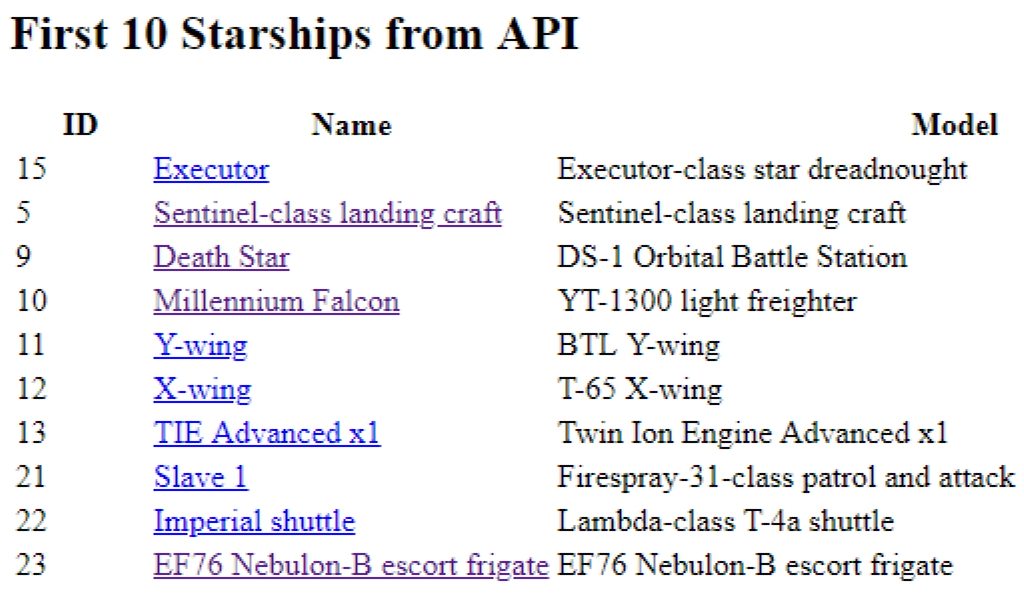
And we just need a link in our list of starships:



```
<td style="width:30%"><a routerLink="{{ship.id}}/detail">{{ship.name}}</a></td>
```

Results

Let's take a look at how the site looks now. First, the list of starships has changed to add the id (I guess you wouldn't have to show the id) and a link in the name of the starship:

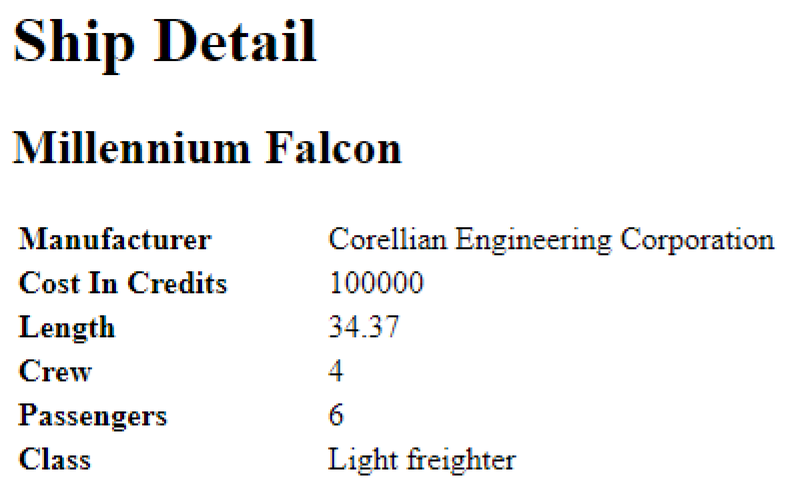


First 10 Starships from API

ID	Name	Model
15	Executor	Executor-class star dreadnought
5	Sentinel-class landing craft	Sentinel-class landing craft
9	Death Star	DS-1 Orbital Battle Station
10	Millennium Falcon	YT-1300 light freighter
11	Y-wing	BTL Y-wing
12	X-wing	T-65 X-wing
13	TIE Advanced x1	Twin Ion Engine Advanced x1
21	Slave 1	Firespray-31-class patrol and attack
22	Imperial shuttle	Lambda-class T-4a shuttle
23	EF76 Nebulon-B escort frigate	EF76 Nebulon-B escort frigate

If I click on the Millennium Falcon, here is our detail page:

URL=/ships/10/detail



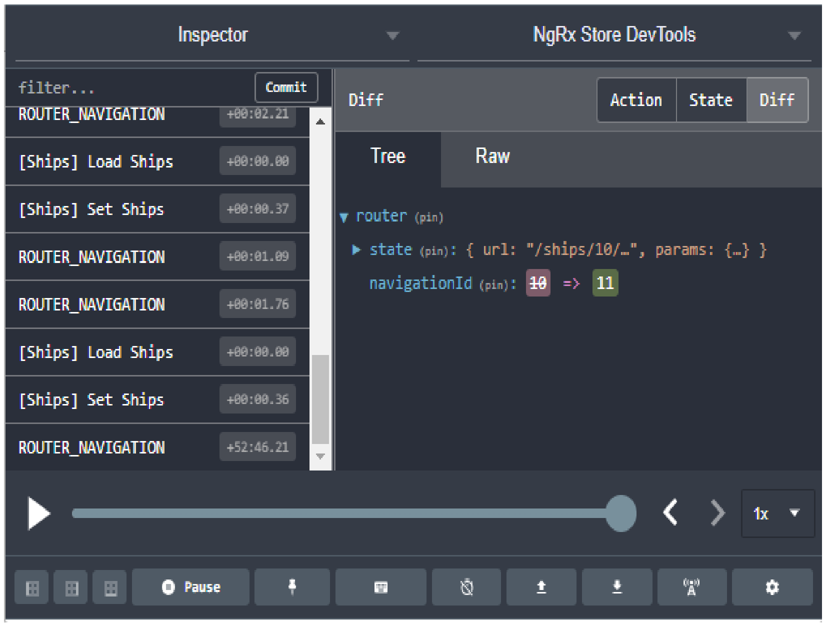
Ship Detail

Millennium Falcon

Manufacturer	Corellian Engineering Corporation
Cost In Credits	100000
Length	34.37
Crew	4
Passengers	6
Class	Light freighter

Time Traveling

Finally, here is a look at the Redux DevTools after clicking the "stopwatch icon" on the bottom row of buttons:



```
ROUTER_NAVIGATION     +00:02.21
[Ships] Load Ships    +00:00.00
[Ships] Set Ships     +00:00.37
ROUTER_NAVIGATION     +00:01.09
ROUTER_NAVIGATION     +00:01.76
[Ships] Load Ships    +00:00.00
[Ships] Set Ships     +00:00.36
ROUTER_NAVIGATION     +52:46.21
```

```
▼ router (pin)
  ▶ state (pin): { url: "/ships/10/…", params: {…} }
    navigationId (pin): 10 => 11
```

- Notice the ROUTER_NAVIGATION entries – those are added there (partly) so we can travel back in time to see the state at a given point, along with actual navigation happening on the page!

Conclusion

Let's recap:

1. Router information is now stored in state – the single source of truth now for our url, parameters and query parameters
2. The parameters can now easily be accessed inside our selectors (or anywhere else for that matter)
3. Our component now doesn't have to have code that gets the id and then retrieves the ship, it is all handled in the selector
4. We can do time-traveling using Redux DevTools now
5. We have some very clean code

It is my wish that you will take the information you've learned in these NgRx tutorials and use them on your job effectively. Happy coding!