Crash Course: Angular & NgRx Workshop

Exercise #1: Additional components

Use the banana component as your example!

Open repository: http://stackblitz.com/github/apaytonmn/fruitsaladapp

Note that this repository is our final banana app with a few additional components: apple & watermelon and a layout component

Apple component: The html is ready, so you need to create the state

1. Create the state folder inside your apple folder
2. In the state folder, create the apple.state.ts file and define the following properties for your apple: isWashed, bitesRemaining, variety
3. In the state folder, create the apple.actions.ts file and define an action to get a new apple
4. In the state folder, create the apple.reducer.ts file and handle the get new apple action to set the properties of your apple
5. In the state folder, create index.ts and export the pieces of your apple state
6. Open app.state.ts and add your apple slice of state to the application state
7. Open apple.component.ts and complete the following:
    a. Define an observable called apple$
    b. Add the store as a parameter on the constructor:
    private store: Store<AppState>
    c. In ngOnInit add a call to newApple() and assign the pipe select call to the apple$ observable
    d. In newApple(), dispatch the action GetNewApple
    e. You will need to add the imports for your getMyApple selector from '../app.state' and your GetNewApple action from './state'
8. When your apple state shows up on your UI, check out your Redux DevTools. When looking at the full state, you will be able to see the multiple slices of state - one for each fruit!

Watermelon component: The state is ready, so you need to create the html and set up your component

1. Open watermelon.component.ts and complete the following:
    a. Define an observable called watermelon$
    b. Add the store as a parameter on the constructor:
    private store: Store<AppState>

c. In ngOnInit add a call to newWatermelon() and assign the pipe select call to the watermelon$ observable

d. In newWatermelon, dispatch the action GetNewWatermelon

e. You will need to add the imports for your getMyWatermelon selector from '../app.state' and your GetNewWatermelon action from './state'

2. Create html to bind to your watermelon$ observable and display your state in text form.

   a. We did not walk through this in the step by step earlier, so use apple.component.html as your example to create the basic html to display the properties of your watermelon based on your observable.

   b. Check out watermelon.state.ts to see what properties are available for your watermelon

Exercise #2: Loading configuration data into the store

1. Under the app folder, create a new folder called config
2. In the config folder, create a folder called state
3. In the state folder, create two files: config.actions.ts and config.reducer.ts

config.actions.ts

In the actions file, we will create an action to load our application configuration

```
import { Action } from '@ngrx/store';
import { State } from './config.reducer';

export const LOAD_APP_CONFIG = 'Load Application Config Action';

// ----------------------------------
// Load app config action
// ----------------------------------
export class LoadAppConfigAction implements Action {
  readonly type: string = LOAD_APP_CONFIG;

  constructor(public payload: State) {
    console.log('ACTION ' + LOAD_APP_CONFIG);
  }
}

export type ConfigActions = LoadAppConfigAction;
```

config.reducer.ts

The reducer code looks much like what we did previously. However, instead of creating a separate file to define the shape of our state, we are defining the state for our config right in our reducer.

```
import { LOAD_APP_CONFIG, ConfigActions } from './config.actions';

export interface State {
  runtimeConfig: {
    programId: number;
    resource: string;
    url: string;
  };
  otherInfo: {
    email: string;
    sessionProperties: {
      format: string;
      value: string;
      sessionIndex: string;
    };
  };
}

export const initialState = {
  runtimeConfig: null,
  otherInfo: null
};

// --------------------
// Config reducer
// --------------------
export function reducer(state = initialState, {type, payload}: ConfigActions): State {
  switch (type) {
    case LOAD_APP_CONFIG: {
      console.log('REDUCER ' + LOAD_APP_CONFIG);
      return {
        ...state,
        ...payload
      };
    }
    default: {
      return state;
    }
  }
}
```

4. In the app folder, create a file called **config.service.ts**
    - We are keeping this very simple by hardcoding our config data. In a more realistic example, this service could make an http call to retrieve data from a config file and pass the returned data as the payload to LoadAppConfigAction.

- The payload needs to align with the State we mapped out in our reducer. As seen here in myJsonConfig, it can be a subset of the values we defined.

```
import { Injectable } from '@angular/core';
import { Store } from '@ngrx/store';
import { AppState } from './app.state';
import { State } from './config/state/config.reducer';
import { LoadAppConfigAction } from './config/state/config.actions';

@Injectable()
export class AppConfigService {

  constructor(private store: Store<AppState>) { }

  async load() {
    let myJsonConfig = {
      "runtimeConfig": {
        "programId": 76,
        "resource": "myResourceName",
        "url": "http://github.com"
      },
      "otherInfo": null
    }
    this.store.dispatch(new LoadAppConfigAction(myJsonConfig));
  }
}
```

5. Open app.state.ts and make the following updates:
    - Add this import:

```
import * as configStore from './config/state/config.reducer';
```

- Then add the config state to the existing AppState, initialState, and reducers following the same pattern as our fruit example. Similarly, add a selector for our config.

```
export interface AppState {
  banana: bananaStore.State;
  ...
  config: configStore.State;
}

export const initialState: AppState = {
  banana: bananaStore.initialState,
  ...
  config: configStore.initialState
}

export const reducers: ActionReducerMap<AppState> = {
  banana: bananaStore.reducer,
```

```
  ...
  config: configStore.reducer
}

// Selector to get config slice of state
export const getMyConfig = (s: AppState) => s.config;
```

6. Open app.module.ts
    - Add the following imports:

```
import { APP_INITIALIZER } from '@angular/core';
import { AppConfigService } from './config.service';
import { Store } from '@ngrx/store';
```

    - Add the hook into app initialization just before the @NgModule decorator:

```
export function initializeApp(appConfig: AppConfigService) {
  return () => appConfig.load();
}
```

    - Add a providers section to the @NgModule

```
  providers: [
    AppConfigService,
    { provide: APP_INITIALIZER, useFactory: initializeApp, deps: [AppConfigService, Store],
multi: true },
  ],
```

7. Now when the app starts up, you will see the config data we created immediately loaded into our state. Check out your Redux DevTools to see it in action.