Angular 6 Training Course

Exercise O-redux

- This exercise will use the **Redux** library to centralise management of application state in one place.
- This application state will be isolated into a **Redux store**.
- Components will use **Redux actions** and the **Redux getState** method to change state in the store.
- The basic Angular application **without Redux** has been built for you.

Setup

- Rebuild the **starter/p-redux** project.

```
npm install
ng serve --open
```

- We will now add Redux code to centralise state within the application.

Define state

- Create a new folder **src/app/redux**.
- Define state as a Typescript interface in **redux/redux.state.ts**

```
export interface Trip{
    airport:string;
    parking:boolean;
    hotel:string;
    hirecar:boolean;
}
```

Define actions

- We need to define **actions** to set the airport, parking, hotel and hirecar.
- Here are examples of the four types of action object:

```
    { type:"airport", data:"heathrow" }
    { type:"parking", data:false }
```

- Create a new file **redux/redux.actions.ts**.

- We can define the action types as constants to avoid bugs caused by case-sensitive text.

```
const AIRPORT : string = "airport" ;
const PARKING : string = "parking" ;
```

- We will define **action creators**. These functions take two arguments: the action type and any associated data.

```
setAirport( AIRPORT , "heathrow" );
// returns { type:"airport", data:"heathrow" }
```

- Add the action creators.

```
export let setAirport = ( airport:string ) => ({ type:
AIRPORT,data: airport }) ;
export let setParking = ( parking:boolean ) => ({ type:
PARKING,data: parking }) ;
```

Reducer

- Two arguments are passed in to the **reducer** function: the current state and a specific action object.
- The reducer is a **pure function**. It does not directly change the Redux store.
- Instead, it will make a **copy** of the current state. It will update that copy based on the action object passed to it.
- The reducer returns the updated object.
- The Redux store will update itself and broadcast its changed state to any listening **subscribers**.
- Create a reducer file **redux/redux.reducer.ts**.
- Make the reducer aware of the state interface you defined in **redux/redux.state.ts**

```
import { Trip } from './redux.state';
```

- Import the action types.

```
import { AIRPORT, PARKING } from './redux.actions';
```

- Define a simple reducer that does nothing but return the current state unchanged.
- We have defined the initial state as an empty object.

```
export let reducer = (state:Trip={}, action) => {
console.log( state,action );
return state;
}
```

Create the Redux store

- We will create the store using the Redux library **createStore** method. We pass it our **reducer** function.

```
createStore( reducer );
```

- We can use **dependency injection** to make all the components in all app aware of the store.
- Create **redux/redux.store.ts**
- Import the Redux library and your reducer function.

```
import { createStore } from 'redux';
import { reducer } from './redux.reducer';
```

- Write a function the creates the store.

```
export function makeStore() {
return createStore( reducer );
}
```

- Add code which allows Angular to pass around the store created by makeStore using dependency injection.

```
import { InjectionToken } from '@angular/core';

export const TripStore = new InjectionToken('Trip.store');

export const storeProvider = [
    { provide: TripStore, useFactory: makeStore }
];
```

- This provider needs to be defined in the app.module.ts

```
    import { storeProvider } from './redux/redux.store';

  providers: [storeProvider],
```

Inject the store into the main component.

- Import the built-in Inject annotation.

```
import { Inject } from '@angular/core';
```

- Import the Redux library.

```
import * as Redux from 'redux'
```

- Import the store as a DI token

```
import { TripStore } from '../redux/redux.store';
```

- Import the state defined as a Typescript interface

```
import { Trip } from './redux/redux.state';
```

- Import the Action types (copy code from reducer)

```
import { AIRPORT, PARKING } from './redux.actions';
```

- Create a constructor which passes in the DI token

```
    constructor(@Inject(TripStore) private
store:Redux.Store<Trip>) {}
```

- Your reducer function will start logging the current state and action objects to the browser console.

```
        undefined {type: "@@redux/INIT"}
```

- Add Observable code which logs when the store changes state.

```
    store.subscribe(() => console.log( "GETSTATE"
,store.getState());
```

- Currently we are not dispatching any actions, so we will not see this code running.

Inject the store into the airport component.

- Add import statements to **airport/airport.component.ts**
- Note the relative path names may differ from the main component.

```
import * as Redux from 'redux'
import { TripStore } from '../redux/redux.store';
import { Trip } from '../redux/redux.state';
import { setAirport, setParking } from '../redux/redux.actions';
```

- Pass the store into the constructor using DI.
- Remember to import the Inject annotation.

```
    import { ..Inject } from '@angular/core';

    constructor(@Inject(TripStore) private
store:Redux.Store<Trip>) {
store.subscribe(() => console.log( store.getState());
```

}

Listen for form changes and dispatch actions.

- Currently the code logs changes to the entire form.

```
this.fg.valueChanges.subscribe( data => console.log( data ))
```

- We can choose to log changes to specific form fields.

```
this.fg.controls.airport.valueChanges.subscribe( data =>
console.log( data ))
```

- Create a method that dispatches setAirport **actions**.

```
setAirport(s) {
    this.store.dispatch(setAirport(s));
}
```

- Call this function from the observable:

```
this.fg.controls.airport.valueChanges.subscribe( data =>
this.setAirport( data ))
```

- The Redux state is logged to the console when we select a different airport. But it does not appear to change.
- We need to add custom logic to the reducer to handle this case.
- Add a switch statement to the reducer.
- Note that Object.assign is used. The reducer is a pure function which returns a **changed copy** of the current state.

```
    switch (action.type) {
case AIRPORT:
    return Object.assign( {} , state, { airport: action.data });
default:
    return state;
```

}

- Change airports in the form. It should log changes of state.

Parking

- Add another case to the reducer for the parking.

```
        case PARKING:
    return Object.assign( {} , state, { parking: action.data });
```

- Create a method that dispatches parking actions.

```
    setParking(b) {
this.store.dispatch(setParking(b));
```

}

- Listen for changes to the parking checkbox.

```
this.fg.controls.parking.valueChanges.subscribe( data =>
this.setParking( data ))
```

Update the message in the main component

- Define an object and update this to reflect the store.

```
trip:object;

constructor(..) {
    store.subscribe(() => this.updateMessage());
    this.updateMessage()
}

updateMessage() {
    this.trip = this.store.getState()
}
```

- Update the main template

```
{{ trip.airport }}
{{ trip.parking ? "with" : "without" }}
{{ trip.hotel }}
{{ trip.hirecar ? "with" : "without" }}
```

O