**Section 24 Bonus Working with NgRx in our Project**

**Section 24: Lecture 307//Module Introduction**

1. In this section we will learn about state management in angular. State describes the state of our application.
2. Is the user authenticated, are the recipes loaded – these are the states of the application.
3. This far in this course we manage the state through services. We stored the authentication status in a services, we stored the recipes in a service. And we used services and methods in our services to interact with the state.
4. We also used the RxJS Subjects in our app as event emitters.
5. If the App grows bigger, then we may require this module for handling states.
6. NgRx is an additional package which we can use in angular.
7. It’s not part of angular itself, but we can install it.
8. It’s all about state management and forcing redux like pattern.

**Section 24: Lecture 308//Important: Angular 6, RxJs 6 and this Section**

Using Angular 6 and therefore RxJS 6+?

I recommend that you go through this module with rxjs-compat  installed.

There'll be a video at the end of the module where we together update everything to work with RxJS 6 without rxjs-compat. But to prevent unnecessary issues with the import path adjustments and pipe(), it's easier to simply use rxjs-compat  by running npm install --save rxjs-compat  in the project folder.

----------------------------

[OPTIONAL] Still want to go without that package?

You find detailed update instructions (Angular 5 => Angular 6) on this page: <https://www.academind.com/learn/angular/snippets/angular-6-whats-new-angular-upgrade>

You'll have to adjust a couple of imports, operator names and use pipe() in this case. Refer to the "Understanding Observables" section of this course (section 13) to learn how you may update your code to use RxJS 6 WITHOUT rxjs-compat .

Essentially, your imports have to change.

For example,

import { Observable } from 'rxjs/Observable';

becomes

import { Observable } from 'rxjs';

Additionally, you use operators differently.

1. import 'rxjs/add/operator/map';
2. import 'rxjs/add/operator/switchMap';
3. myObservable.map(...).subscribe(...)

becomes

1. import { map, switchMap } from 'rxjs/operators';
2. myObservable.pipe(map(...), switchMap(...)).subscribe(...);

Last but not least, you'll encounter operators in this module, for which the name changed:

do()  => tap()

catch()  => catchError()

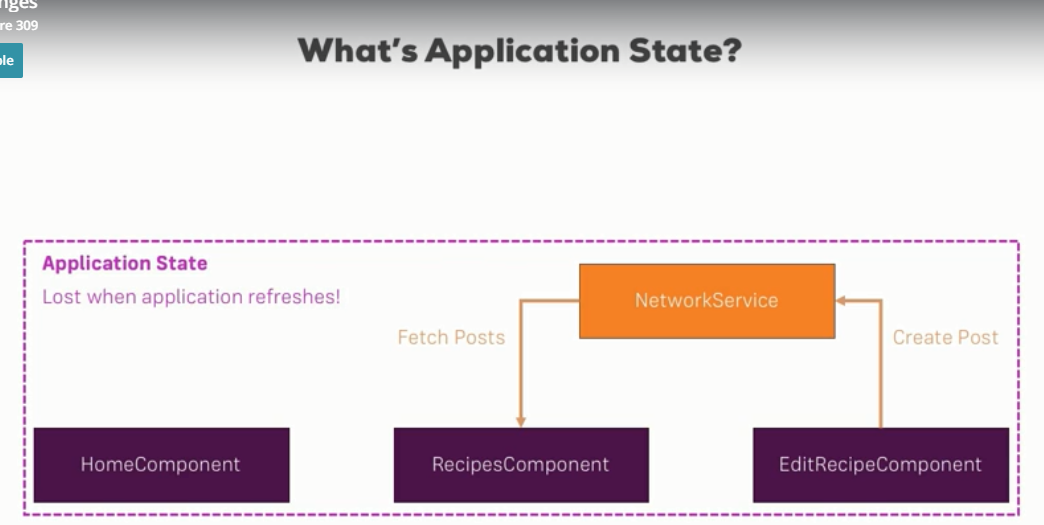
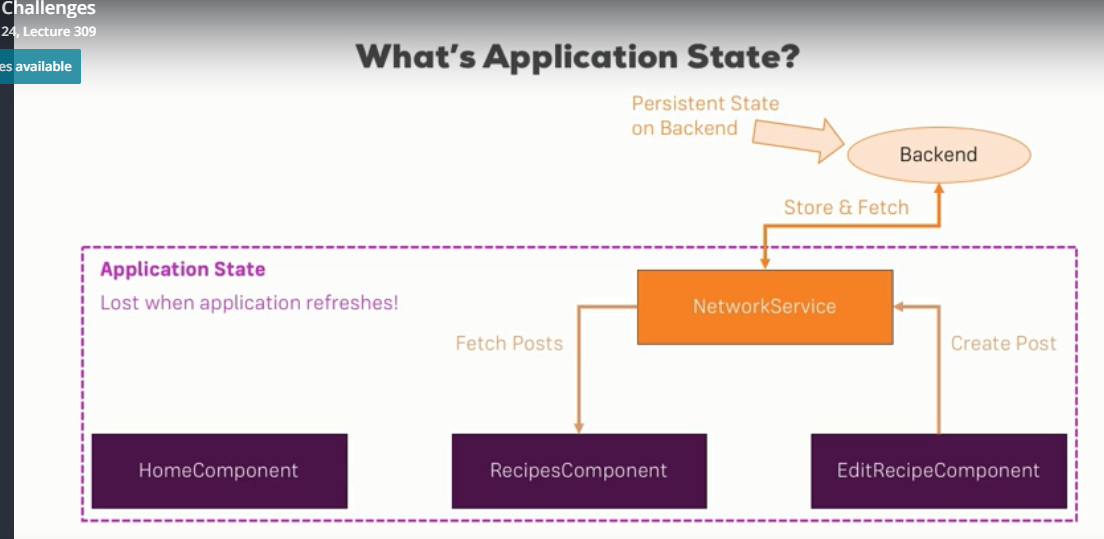
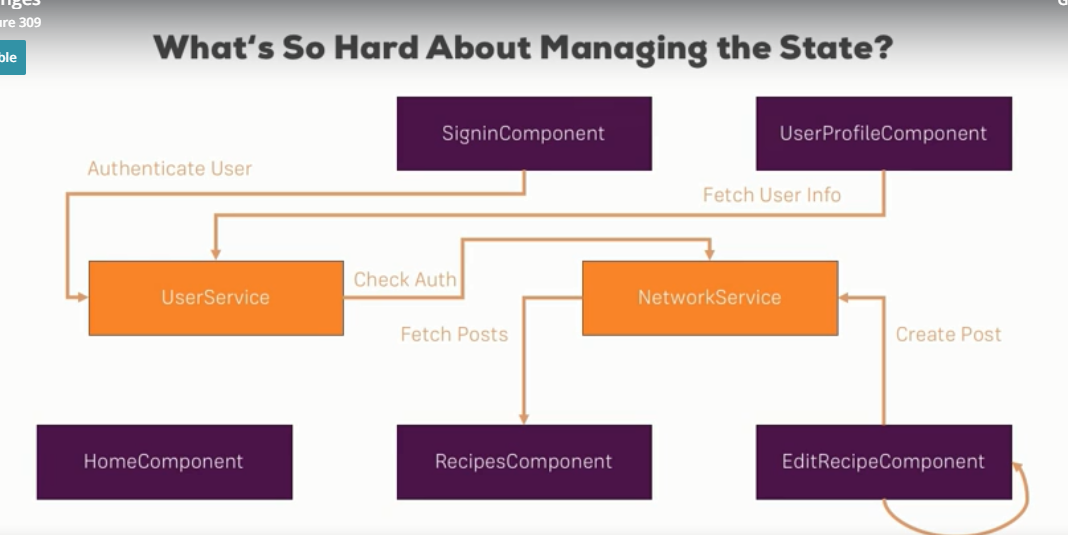
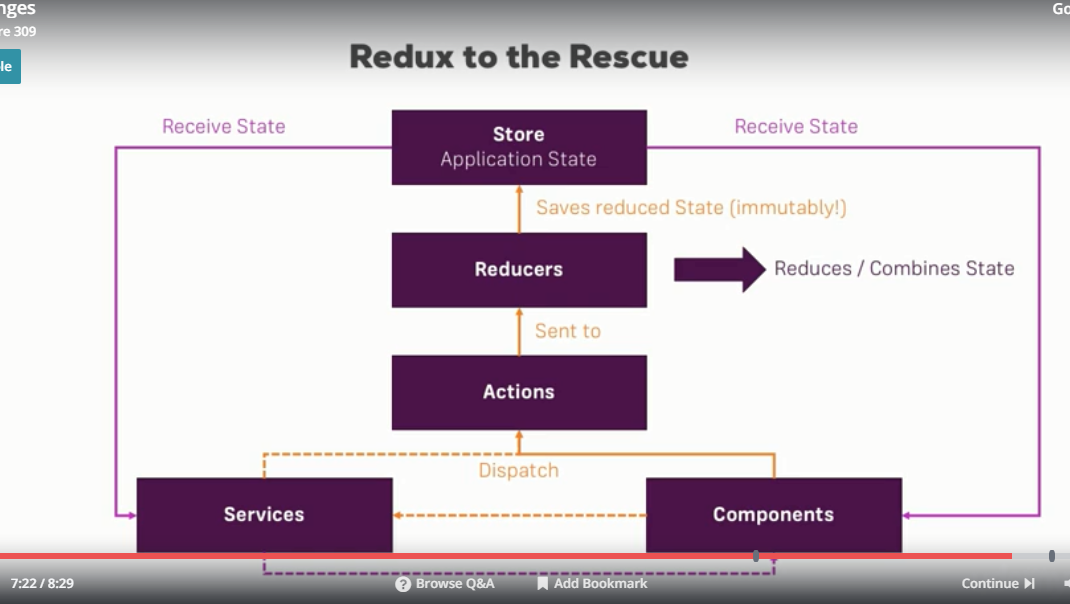
An observable-creation method (throw ) was also renamed:

1. import { Observable } from 'rxjs/Observable';
2. import 'rxjs/add/observable/throw';
3. ...
4. myObservable.pipe(catchError(error => Observable.throw(error));

becomes

1. import { throwError } from 'rxjs';
2. ...
3. myObservable.pipe(catchError(throwError(error));

**Section 24: Lecture 309//State Challenges**

1. 
2. All this application state is lost once we reset our application. Here we get a new index.html file in our app.
3. We can store the parts of the state in the local storage if we want to persist it.
4. We also have backend where we have possibility of storing and fetching data. The backend is a persistent state as it would be still there after we started.
5. 
6. State in general describes what the user sees, what the user does; it keeps track of what happened in our application.
7. There are some things of the state which we want to store at the backend - for example the recipes that the user created.
8. If there is longer period of inactivity, we may loose our state. This is state in general.
9. What can be difficult about managing the state
10. 
11. Above application is relatively simple application with some components. But there are lot of things going on with the state of the application. State is edited and queried from all over the app.
12. As our app grows it difficult to know where out app state is edited and which service could override the existing state.
13. In our recipe service we are using the Subject to inform our app whenever the state changes.
14. If our app is small the services and Subject are enough for the application.
15. When the application grows then using the subject approach with services may not be enough to keep the application state track issues in the application, enough.
16. So, in this scenario we can use a different pattern which is quite popular in the react community – Redux.
17. Let’s see how we can manage the application state when using the redux design patterns.
18. In the redux approach we have one central store in our application i.e. once central place where we manage our application state. With this approach there is one place where the state lives.
19. In this approach we dispatch Actions to define our state/ or change our state.
20. These actions call reducers.
21. 

**Section 24: Lecture 310//Getting Started with reducers**

1. npm install --save @ngrx/store
2. In the reducer we must return the new state of our application.
3. Reducers updates the state not by setting it but by updating it with the new onw.
4. ... is spread operator
5. Spread operator will pass all the properties of the old object will be copied to the new object we are returning.
6. ...state.ingredients
7. This will distribute all the elements of ingredients array into this new array.
8. shopping-list.reducers.ts:
9. //import { Action } from "rxjs/internal/scheduler/Action";
10. import { Action } from "@ngrx/store";
11. import { Ingredient } from "../shared/ingredient.model";
12. export const ADD\_INGREDIENT = 'ADD\_INGREDIENT';
13. const initialState = {
14. ingredients: [
15. new Ingredient('Apples', 5),
16. new Ingredient('Tomatoes', 10),
17. ]
18. };
19. export function ShoppingListReducer(state = initialState, action: Action){
20. switch(action.type){
21. case ADD\_INGREDIENT:
22. return {
23. ...state, //... is spread operator
24. ingredients: [...state.ingredients, action ]
25. }
26. }
27. return state;
28. }

**Section 24: Lecture 311//AddingActions**

1. shopping-list.actions.ts:
2. import { Action } from '@ngrx/store'
3. import { Ingredient } from '../../shared/ingredient.model';
4. export const ADD\_INGREDIENT = 'ADD\_INGREDIENT';
5. export class AddIngredient implements Action{
6. readonly type = ADD\_INGREDIENT;
7. payload: Ingredient;
8. }
9. export type ShoppingListActions = AddIngredient;

**Section 24: Lecture 312//Finishing the first Reducer**