CSM

breaking thought barriers

Angular Training

Session -23

Outlines

Statement Management using NgRX

State Management and NgRx

Problems of complex Angular applications

As we build more and more complex business applications using Angular, we eventually run into issues, such as:

- how to manage all the interaction between components
- how to persist data between sections of the application
- how to make one part of our application access data from another
- how changing from one part of the app affects another.

All of these problems come under the responsibility of state management.

How can we manage the state?

Luckily for us, as Angular developers, there are a few options regarding how we can manage state. We could roll our own solution using **RxJs**, or we could make use of **local storage** and a **service layer** to store data (but this isn’t a good approach).

Thankfully, some very smart people in the Angular community have come up with approaches to solving this problem. There are many frameworks available for us that help manage state in Angular, such as **NgXs, Akita, and NgRx**.

Why choose NgRx?

NgRx is used in some large enterprise-level applications.

What is state management?

State management is the approach we take to manage how data is synchronized, stored, and accessed throughout our application—making sure that the data displayed to the user is current and correct.

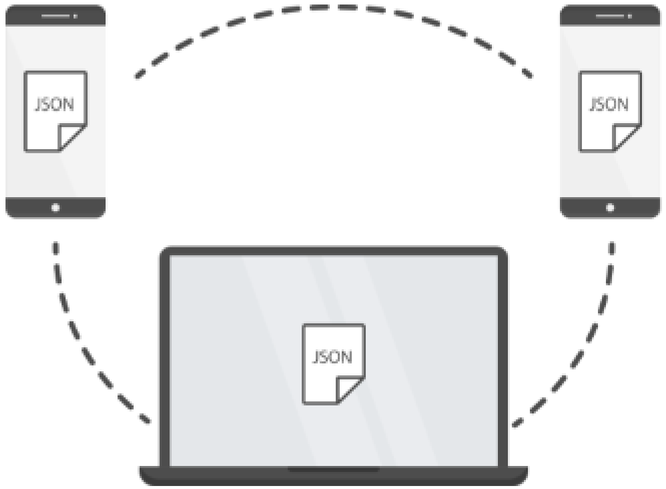
Modern web application and state management

A lot of aspects of a modern web application can come under the term state management. With modern web applications, we need to think about how we manage data from multiple sources, such as:

- How do we **pass data** around our application? Whether we use **Events** to pass data or **Services**.
- What data should we pass **within the URL**, or should we even pass data in a URL?

- How do we make sure that all the data we show in the application is in sync?

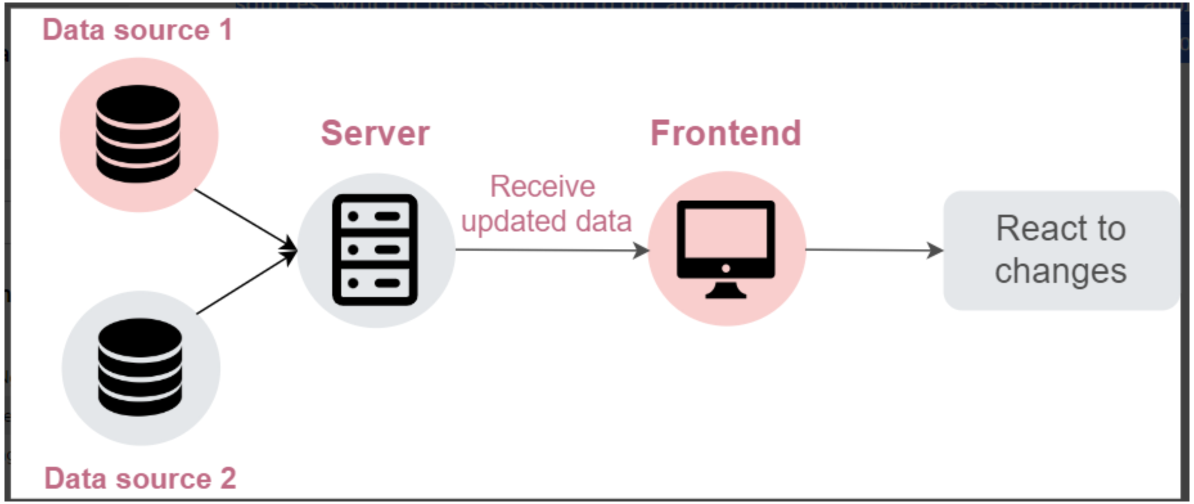
For example, if we're building a financial application. What if a user sees a monetary value on one screen and the same represented value is a different number because the data is not in sync, then the application appears broken.



Data is synced

•We also need to make sure that the *state of the application represents what the state is on a backend server*.

If our application loads data from a backend service and the backend gathers data from multiple sources, which it then sends out to our application, how do we make sure that our application reacts when this updated data is sent so that it shows the same data as the backend application?



Types of states

- The **Local User Interface state**.
- **URL-based state**, where values are persisted through query parameters attached to URLs in the application.
- **Server-side state**, which is state information that’s persisted on a backend/database and sent to the application.
- **Client-side state**, where the state information is stored on the client (the web browser) instead of a backend server.
- **Transient state**, where the state is stored on the client-side, but the user is not aware of it. They do not see values being passed in the URLs.
- **Persisted state**, where the state stored in a backend server is passed and stored on the client by using services and local data storage.

Methods for Implementing State Management

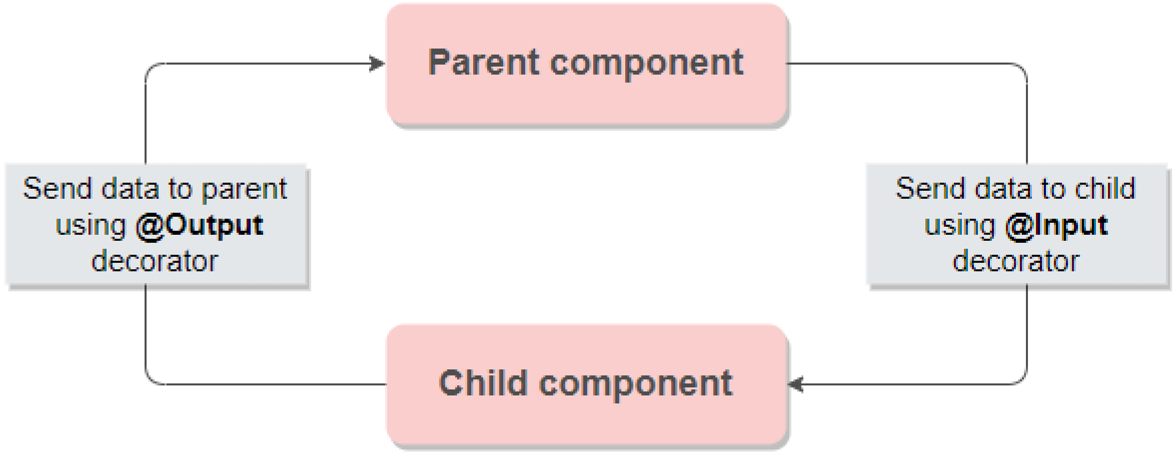
Approaches to state management

- ❑ Manage the passing of state information through **@Input and @Output** decorators in our Components, along with Events to send state information between components in the application.
- ❑ Make use of **Services and Dependency Injection** to pass state information between Components and other Services.
- ❑ We could pass state information through URLs via **Route Parameters**.
- ❑ We can pass data around by storing data in the **browser's local storage**.
- ❑ Use an **Observable-based approach**. For example, create Services where the state is stored and accessed through Observables that react to changes in the application.
- ❑ Using an **RxJs** approach based on the Redux pattern
- ❑ Use a third-party library like **NgRx**, which has been designed to solve this problem

Using @Input and @Output decorators

The first approach is a perfectly reasonable approach. The state is passed into and out of components using the @Input and @Output decorators.When this state data is passed into the Component, it can change the view in the Template, and if the user makes a change to this state data within that Template, this can be transmitted to other areas of the application through Events and the EventEmitter.

This approach is fine, but as the logic of the application becomes more and more complex, this approach can soon become a maintenance nightmare and lead to **spaghetti code**.

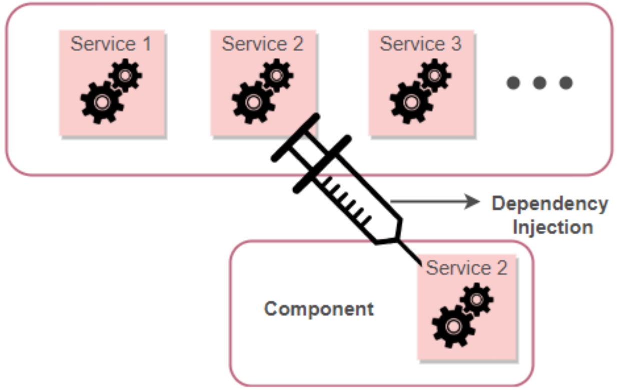

Parent component
Send data to parent using @Output decorator
Send data to child using @Input decorator
Child component

Input and Output decorators

Services and Dependency Injection

Using Services and Dependency Injection is a better approach since the state is stored and accessed through the Services, means that any Component or other Service can access this state data when a Service has been injected.

However, again, this could lead to a very complex code base, and you would have to manage the state being passed through Promises. However, making use of Observables would make the state more reactive to changes.



Dependency Injection

Pass state information through route parameters

We can also make use of what the browser provides us with. We could pass state information through URLs via Route Parameters, which we learned about in the Routing and Navigation chapter. Route parameters allow us to attach data to the end of a URL. This data can be accessed as route parameters in different sections of the application.

While this approach is fine for a small amount of data, as the data grows in size, parameters can become complex, thus making it difficult to keep track of the data being passed.

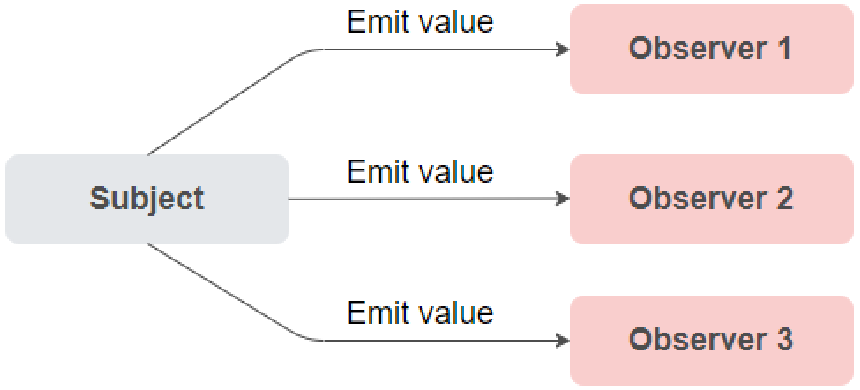
Another problem with this approach is that this data can be seen in the browser's address bar, which could be a security risk, depending on the data being passed.

Store data in the browser’s local storage

One other approach we could use to pass data around the application is by storing data in the browser’s local storage. All modern browsers have local and session storage, which we can write to using JavaScript. So, if we wanted to store some sort of state, we could write it to either local storage or session storage. Both are great options.
There are a few TypeScript libraries we can add to our Angular applications that make writing to these storage options really easy.

Observable-based approach

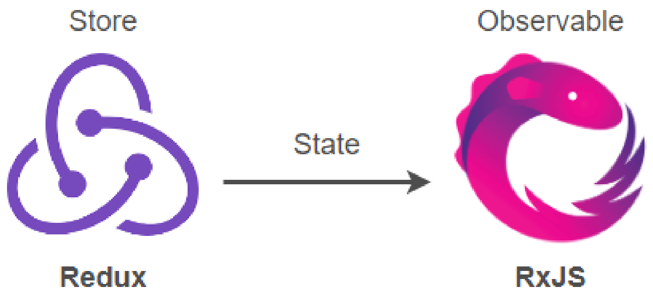
An Observable-based approach is a good approach because we can Subscribe to Observables, which then emit state changes to all the Subscribed Observers. Using Subject Observables such as **ReplaySubject** and **BehaviorSubject** provides the ability to send out state-based changes to multiple Observers, which is a better approach.



Subjects in RxJS

Redux pattern and RxJs

Using the Redux pattern and RxJs is a good approach because we are making use of Observables to broadcast state changes within our application, and following a design pattern like Redux gives us a roadmap to follow when implementing state management in our applications.

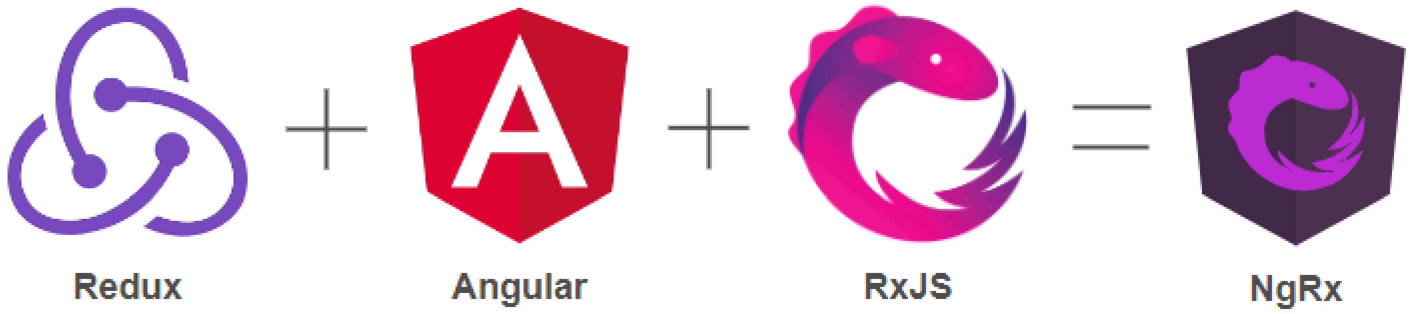


Redux and RxJS

This approach's drawback is that a less experienced developer may implement this approach differently compared to an experienced developer who has learned about the best approaches for implementing state management following the Redux pattern.

NgRx

This is why using a third-party library like NgRx is a great way forward. Not only are we following the principles set out in the Redux pattern, but the approach set out in NgRx has been written by experienced Angular developers who have solved this complex issue many times and add solutions to NgRx based on their knowledge.

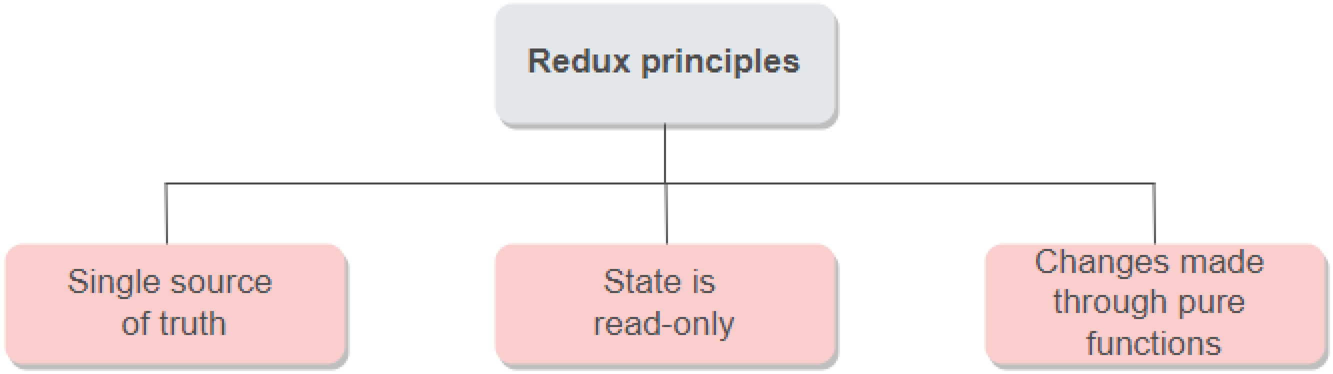


NgRx

What is Redux

Redux is a state container for JavaScript apps.

The principles of Redux



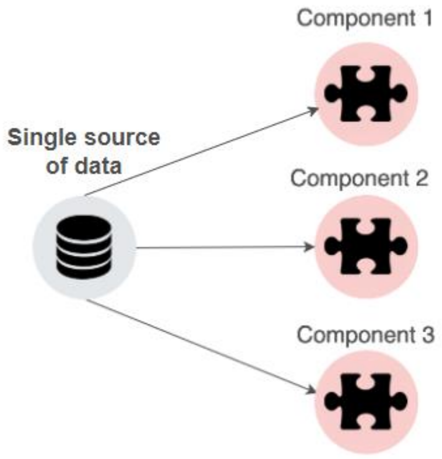
Principles of Redux

Single source of truth

In the Redux approach, the state of the application is known as the single source of truth.

This means that the state of the application is stored in one place so that whenever we get some state information for our application, we only get it from a single place.

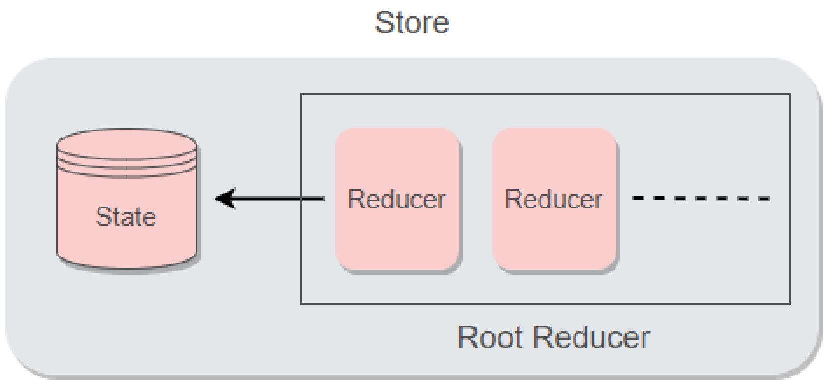
This solves the problem where we may have two components that are showing the state information. They are both getting it from the same place, which means that both components will show the same data.



The state is read-only

The second principle is that the state is read-only. This means that the state store can only be read directly. It can't be changed directly. All changes must be made through our Reducers, which use pure functions to amend the state.

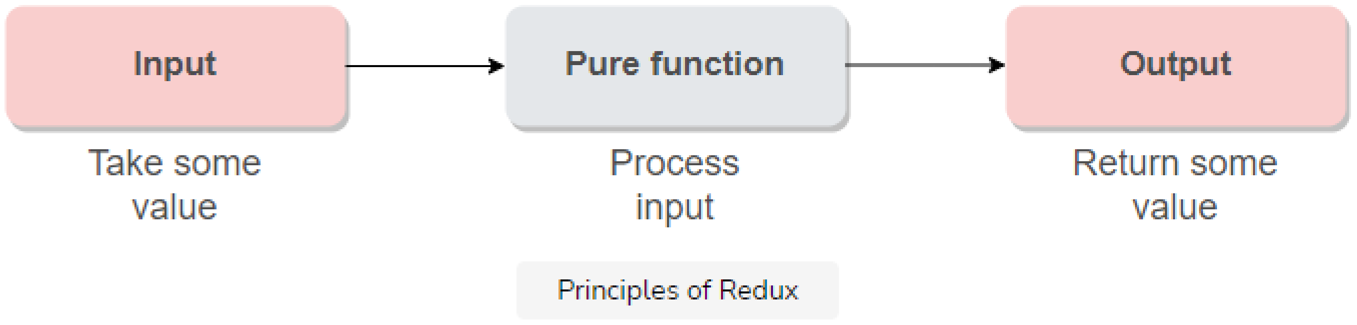
Using this Reducer approach allows all changes to be centralized and happen one after the other in order. This means there is never an issue with the data stored in the state being changed by one part of an application before another part of the application can update the state. We never get what is called a race condition, where one change goes through before a previous change has happened.

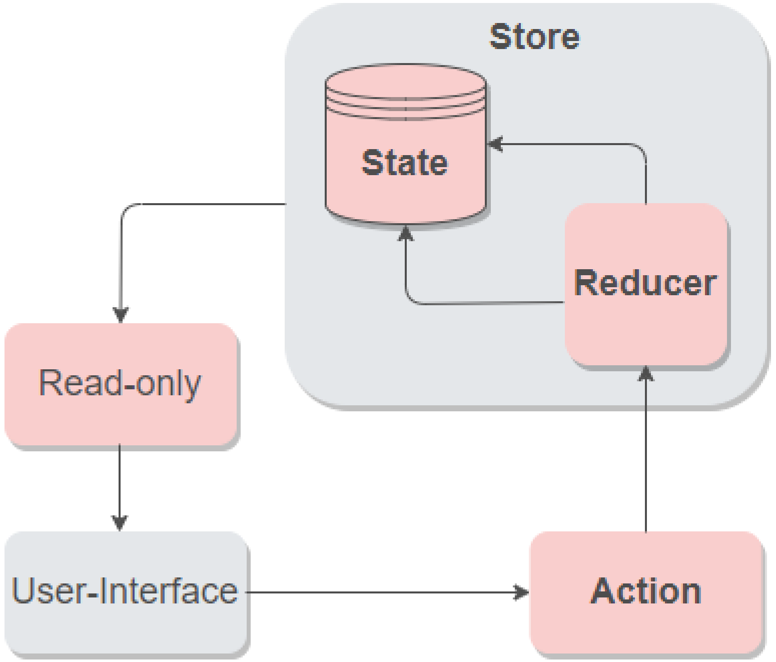


The state is changed through Reducer

Changes are made through pure functions

A **pure function** is a function that takes in an argument and always returns a value.



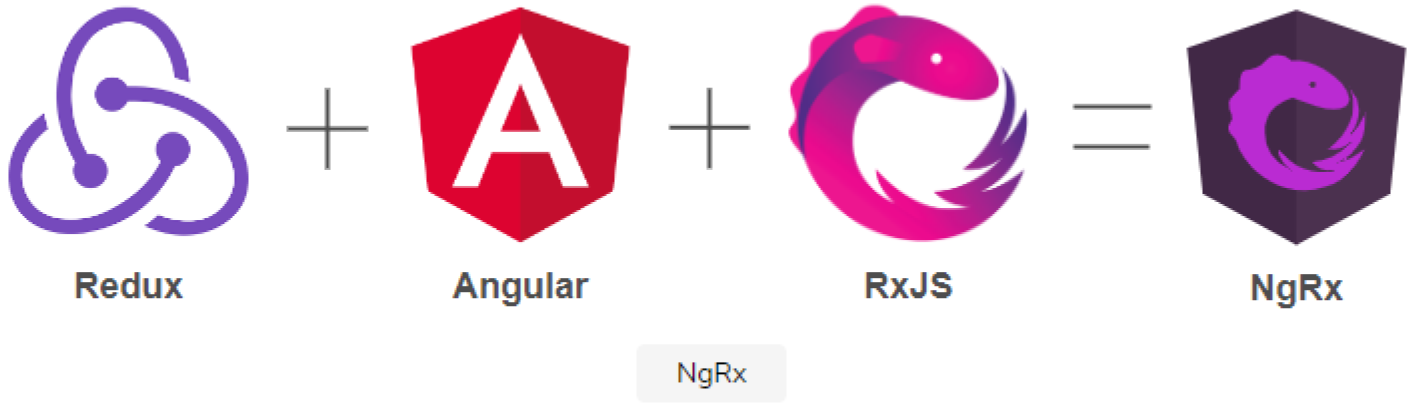


How Reducer changes the state

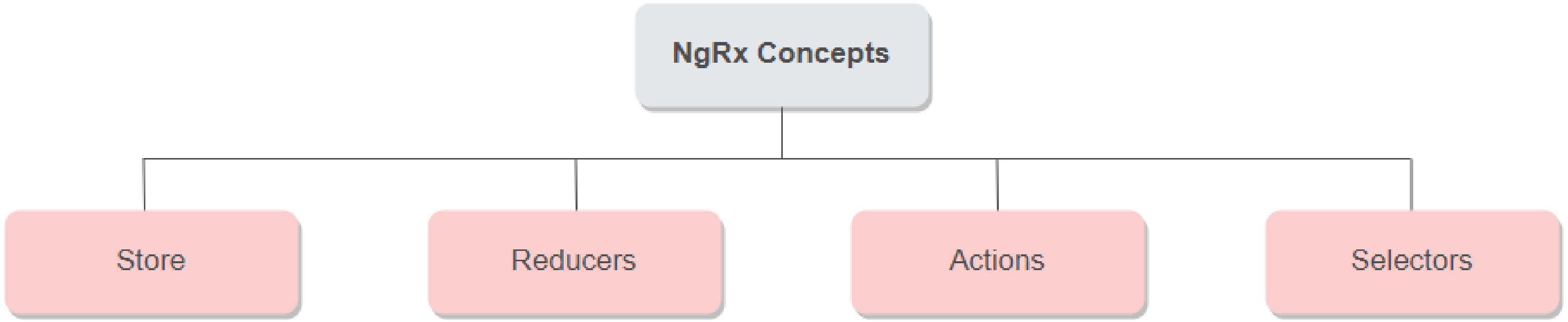
What is NgRx?

The best place to start when looking into NgRx is on the official website.

NgRx looks different from the official Angular website. This is because while NgRx is used with Angular, it isn’t officially part of Angular. It’s a library that we can use with Angular.



The core concepts of NgRx



Key concepts in NgRx

Store

- The store is the state container of our application, which is an Observable object.
- This object contains all the actual state information for our application.
- The Actions and Reducers interact with this store to add this state management to the application.

Reducers

- Reducers are the pure functions we mentioned earlier when looking at Redux.

- Reducers handle the functionality where we go from one state to the next.

- A Reducer function will take in the current state plus an Action and then return a new state.

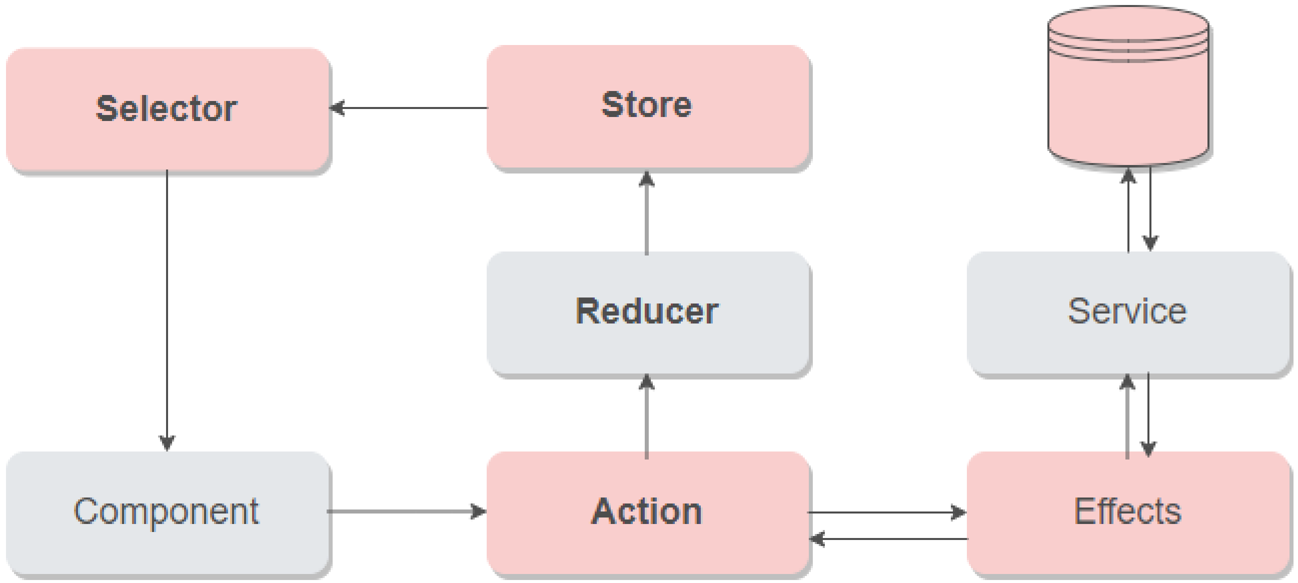
- For example, if we have an Action that adds a new item to the state, a Reducer would return a new state that contains this new item as one single state object.

Actions

- Actions are the events we dispatch from our Components and Services.
- These Action events or Actions are unique events that happen in our application.
- They cover everything from a user clicking a button to a Service making an API call. All these different things are triggered by Actions.

Selectors

- Selectors are also pure functions that return a selection of the State store object.
- They allow us to get information for the State store without having to return the entire store.
- These are helpful when getting state information for our views, where we don’t want to have the entire State Store just to show a small amount of data in the view level.



How NgRx manages the state

Using NgRx concepts in Angular

In an NgRx based Angular application, we write Actions, Reducers, and Selectors in order to access the Store object that contains all the state information of our application.

Example: Save form data

Suppose we want to write some data to the store when the user clicks on a button, for example, a Save button. We write an Action that contains a payload of the form data. This gets passed to a Reducer, which looks at the action and performs this action on the state, which in this case will be an ADD action.

Note: A payload is an object that is passed to the Reducers, along with the action from the action event. It can contain any data that is needed as part of the action.

What is Action?

An Action is dispatched when a message needs to be sent.

Action interface

All Actions are based on the Action interface, which defines what an Action must have. The interface is like the signature of the class. It defines what the class should have without setting the implementation.

A class that implements an interface has to implement what the interface says it should have, but how the class implements that feature is up to the individual class.

```
{
  type: 'Add Apples'
  payload: {
    numberOfApples: number
  }
}
```

Any class that implements the Action interface must have a type.

We can create separate TypeScript files for each Action if we want and have them kept in an actions folder. Or, we could also have all our actions grouped by functionality in a single TypeScript file.

What is a Reducer ?

Reducers handle the functionality where we go from one State to the next.

A Reducer doesn’t implement an interface as the Action does, but it does return a State object. The Reducer function takes in two arguments, that is, the State and the Action.

Step 1 : Define the interface of our State model

Step 2 : Create an initial state object, along with some default values

Step 3 : Create a Reducer function that handles different Actions

What is a Store?

A Store is an Object from NgRx that contains the State of the application. This Store object has an API that allows us to manage the State contained within the Store.

The State is the data we create for our application, which is specific to the application we’re building.

Creating a State tree of Reducers

In an NgRx-based application, we probably have many Reducer files that all contain state information, but we want a way to combine all the Reducers into one state tree.

A state tree is a mapping of all the State objects contained within our Reducer files. This idea of a tree is similar to how we think of Components within Angular and how they all branch out of a module.

Step 1 : We are creating an interface of our State,

```
export interface State {
  apples: applesProductState.ApplesState,
  oranges: orangesProductState.OrangeState
}
```

This interface can grow to have all the State information

Step 2 : Pass this interface to an exported constant variable called a Reducer, which is using a utility from NgRx called **ActionReducersMap**.

This mapper helps us build up a list of the Reducers (and their State). But in order to make use of type checking, we pass the State interface as the type of the Reducer's Object being exported. This means that we can't add new Reducers that are not part of the State interface.

```
export const reducers: ActionReducerMap<State> = {
    apples: applesProductState.reducer,
    oranges: orangesProductState.reducer,

    // not part of theState interface
    banana: bananaProductState.reducer
}
```

Registering the State

Methods for registering the State

There are two levels where we can register the State:

1. at the global application level
2. at the feature level.

Registering the State at the global level

The global level means that the entire application can access the state tree made from the Reducers.

This is fine for a small application, but if we have a more complex application, we could use the feature level module to register the state tree for that feature.

To register at the global level, we use the StoreModule class from the NgRx Store API to make NgRx aware of the Reducers file we’ve just created.

Registering the State at the feature level

```
import { NgModule } from '@angular/core';
import { StoreModule } from '@ngrx/store';
import { appleReducer } from './reducers/ fruit.reducer';
@NgModule({
  imports: [
    StoreModule.forFeature(‘ appleOrdering ', appleReducer)
  ],
})
export class AppleOrderingModule {}
```

```
import { NgModule } from '@angular/core';
import { StoreModule } from '@ngrx/store';
import { FruitOrderingModule } from './ordering/ fruit- ordering. module';
@NgModule({
  imports: [
    StoreModule.forRoot({}),
    FruitOrderingModule
  ],
})
export class AppModule {}
```

1. Create an Actions file defining the Events we can perform on the State
2. Create a Reducers file containing an interface that defines the structure of the State
3. Add an initial state object to the Reducer file of the State.
4. Add a switch statement to the Reducer file, with a case for each Action that can be performed on the State.
5. Always return the State from the Reducer file.
6. Create an interface defining the structure of the map of Reducers for a feature.
7. Create an ActionReducerMap of all the Reducers in a feature of the application.
8. Create a global level Store using the NgRx StoreModule's forRoot() method in app.module.ts.
9. Register a feature level Store using the NgRx StoreModule's forFeature() method.
10. Register the feature level Module Store with the global level Store.

What is the Selector?

The **Selector** allows us to get information for the State store without having to return the entire store.

Selectors are pure functions in NgRx, similar to Reducers.

NgRx helper functions

NgRx provides two helper functions: **createSelector()** and **createFeatureSelector()**

The **createFeatureSelector()** method gets the feature state (the state of a feature module),

The **createSelector()** method returns a selection of that State information.

Getting State using a Selector

Step 1: Create a Selector

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

import { ApplesState } from '../state/ apples.state';

const selectApples = (state: ApplesState) => state;

export const selectApplesFromStore = createSelector(
  selectApples,
  (state: ApplesState) => state.applesCount
);
```

Step 2: Subscribe to Results

To use these Selectors in a component, we import these Selectors into our component and then subscribe to the results using the Async pipe in the app.component.html