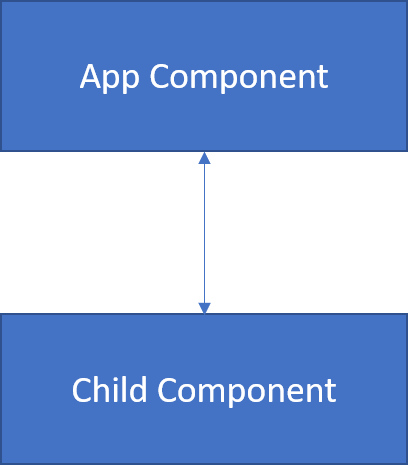
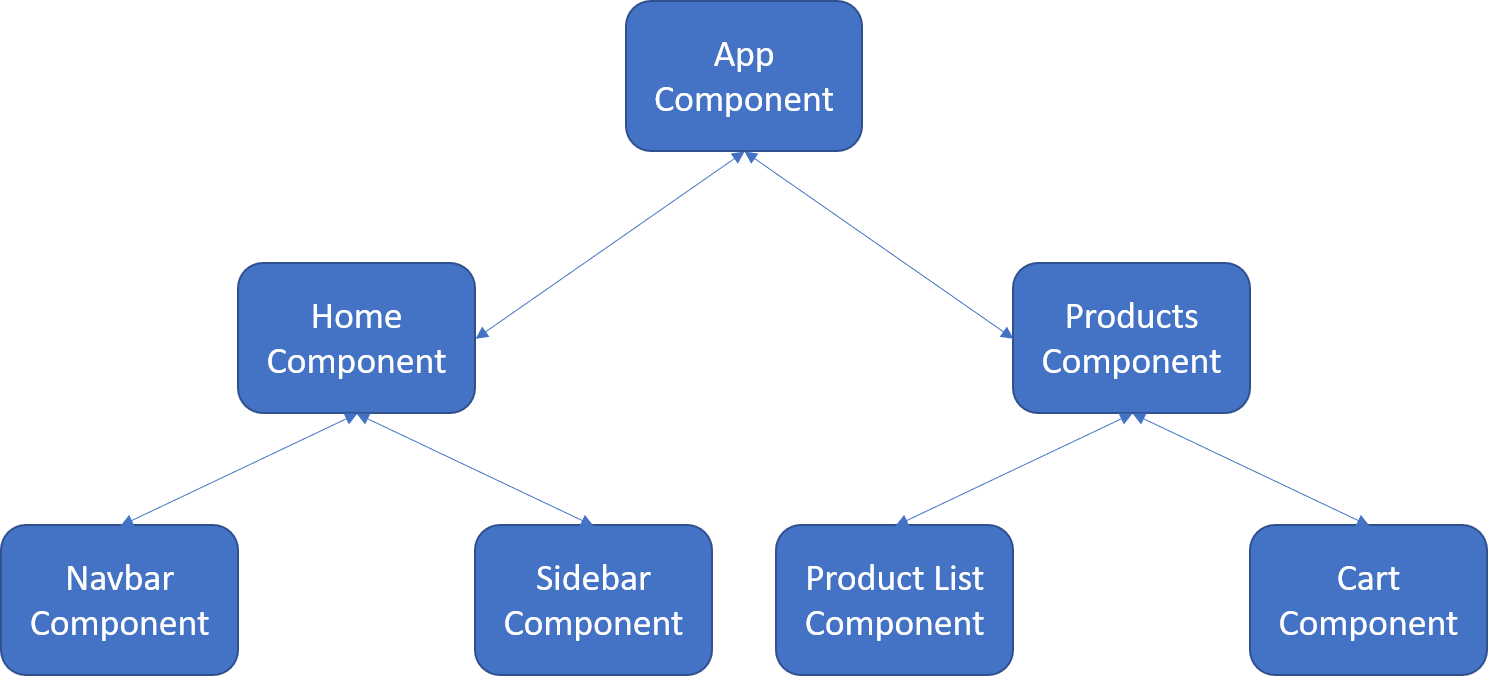
We have already covered most of the topics with respect to developing SPAs using angular, but we never used the term **state management** anywhere in the application. So, before we learn how to manage state, its important to know what is state ?

We will find a lot of definitions for State,  but in simple words, ***“State is the data or the behavior of the app at a given instant”.*** It includes both, the **state of the UI**as well as the**state of the variables** in the code. So, any change in either of these is a change in state of the application.

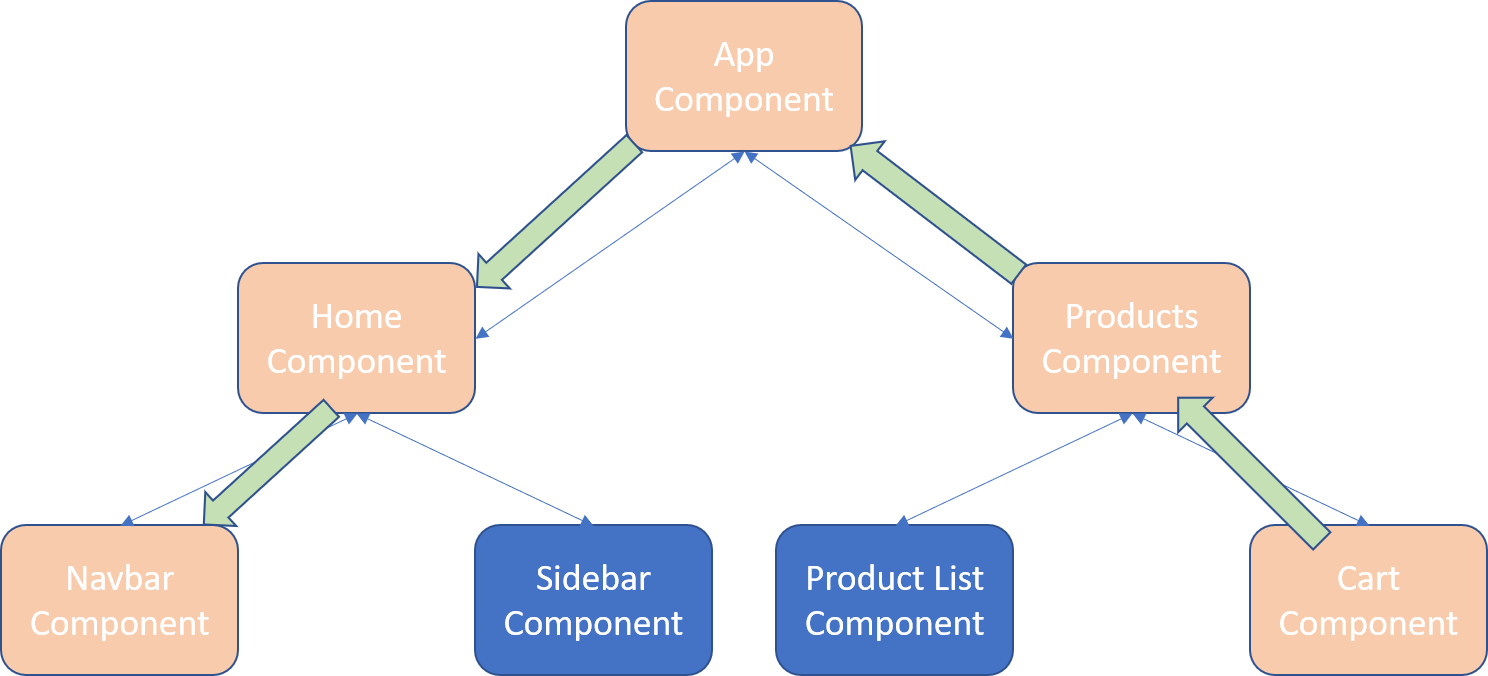
In angular, the application has a **component-based**architecture and each component maintains its own state and one component does not have any information about the state of other components until it is passed from one component to other using **@Input**or **@Output**. It is easier to exchange data between components in this way for simple apps as shown.

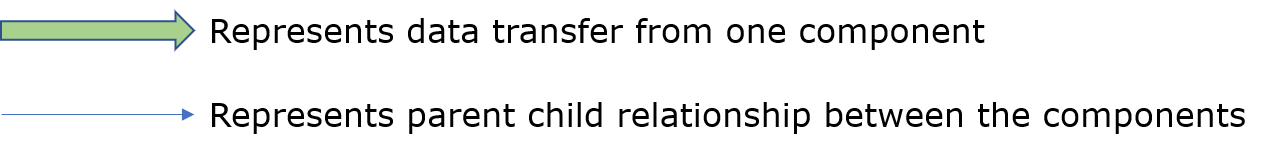


But with complex app architecture as shown, it gets very difficult and painful to exchange data between components. Consider if we had to pass some data from cart component to the navbar component.



If we use **@Input/@Output**, we will jump across 4 different components to do so:





No Worries, Here comes **State Management** to rescue.

**State management** is a technique that we use to handle the state of the application effectively to ensure the consistency of data across the application. We will be using **ngRx**for state management in angular.

Next we will learn about **ngRx**in detail.

ngRx stands for **Angular Reactive Extensions**.

It is an open source library that helps to build **reactive applications** using angular. It is a great design pattern for predictable state management, which is inspired by Redux (a state management library for React). It aims to bring reactive extensions to angular and redux like a single store for the state in an application.



Some of the features provided by ngRx are:

* State management
* Isolation of side effects
* Router bindings
* Entity collection management

In this course, the focus will be on **state management using ngRx**.

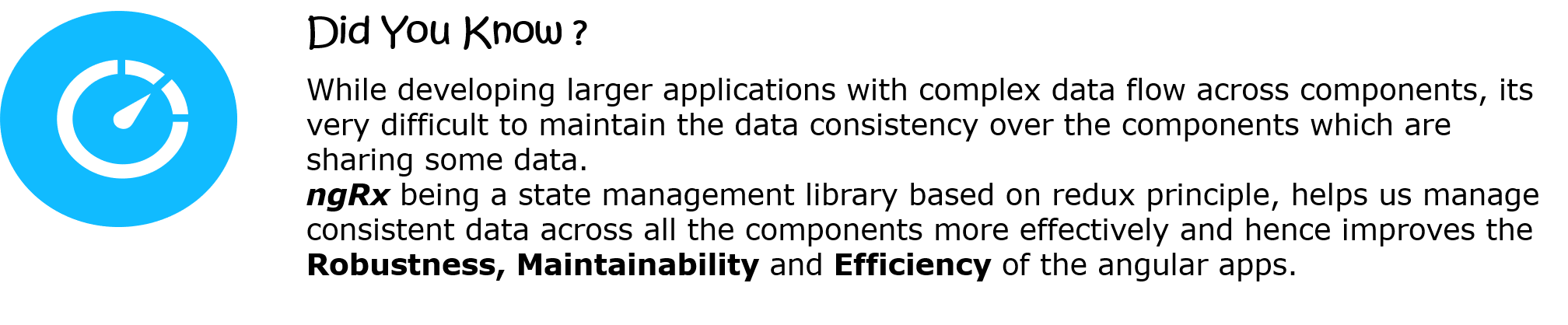
**Why should you use ngRx for state management ?**

**ngRx**should be used in medium/complex applications which involve a lot of user interactions and multiple data sources.

A good way to decide whether we need ngRx in our angular applications is **SHARI principle**of ngRx.

* **Shared**: state that is accessed by many components and services.
* **Hydrated**: state that is persisted and rehydrated from external storage.
* **Available**: state that needs to be available when re-entering routes.
* **R*etrieved***: state that must be retrieved with a side-effect.
* **Impacted**: state that is impacted by actions from other sources

Next we will see an analogy to understand ngRx furthermore.



**An Analogy: A visit to the Library**

Let’s consider an event that we all are likely to be familiar with — visiting a library to borrow a few books.

Let's say, one morning you wake up and want to read some books.

Rather than buying them, you decide to visit your local library to issue them instead. While going to the library there’s just one intention or action you’ve got in mind i.e. **BORROW**(issue/borrow a book).

Here’s where things will get interesting, in the above scenario, there are three main tasks that are happening:

* A **Library**which has all the books stored for not only you, but many others.
* The **Customer**(you) wants to access the books in library without much hassle
* You relied on someone else (a **Librarian**), to smoothly and efficiently let you borrow some books.
* And definitely not to forget the **Book**itself

You are being the customer of the Library, looking to borrow some books. You won't go directly to the Library's book racks and take a book on your own. Rather you need to rely upon the library system that includes a librarian to handover your required books to you.

Thats where Redux pattern plays a role in ngRx for state management.

The previous scenario is very common, and is faced by almost all of us.

Your **issued books** in the library signifies your**'state'** with respect to the **Library**. Library as institution manages everyone else's books along with managing your books specifically. For which, the Library has created different accounts (or library cards) to preserve the 'state' of each of its customers.

The library company can be thought of as **ngRx** as a whole. This scenario has four main actors involved:

* **Book racks:** This is the only single source of books, which is being used for everyone's transactions (borrow/returning/extension etc).
* **You:** You are the one who want to borrow (or return) the books. The library has your existing state with it, and you are relying on the library system to get your book out of the pool of books sitting inside the library.
* **A Librarian:** They are the agents for the actual management of money. A librarian will validate if you have the authority and the necessary availability of books for your transaction.
* **The Book:** The book rack is going to have a lot of books placed. We don't need the entire book rack, rather we want only the book of our choice.

A library transaction results in the manipulation of books or the 'state' in the Library. Hence, the librarian is the person who is responsible for changing from one 'state' to another.

Likewise, **ngRx** is similar to the library company. **ngRx** also has four main parts:

* **Store -** Store is the place where all the possible states of all the components of your application are stored. **Store**in **ngRx**can be understood as the book racks of a **library**where the books for all the customers are kept.
* **Actions -** Action is the one which initiates a request to change the 'state'. In our example, Action is analogous to you. As Action initiates state change, similarly, you are the one who initiates any book borrow or book return from your account.
* **Reducers -**These are the agents that change the state of app, in a smooth and predictable manner. Reducers are analogous to the librarian who are responsible for any borrow or return of books happening in the library.
* **Selectors -**When the customer borrows a book, the library does not provide the entire book rack to the customer. It just gives the required book to customer. The same job is done by the **Selector** in ngRx. It provides only the required states from the store.

Hence, the main actors in ngRx for any state management are Store, Reducers, Selectors and Actions.

Next, lets see the principles of redux based on which ngRx works.

**ngRx**is based on the **three**fundamental **principles**of **Redux**:

* Single source of truth
* State is immutable
* Changes are made with pure functions

Lets understand these three principles.

**1. Single source of truth**

The state of complete application is stored in a single state object tree within a store. Having single store in a application makes it easy to create, debug and test. It also means components should read data from this single source and not keep their own version of the same state separately. The state of whole application is centralized and stores in a single state object. In ngRx, store is the single source of truth.

**2. State is immutable**

State is immutable. The only way to change a state is by dispatching an action. Since whole application is stored in single state tree, having immutable state make sure that no views or network callbacks wrote directly to state. Instead, they dispatch an action, which have type property which indicates the  type of action being performed.

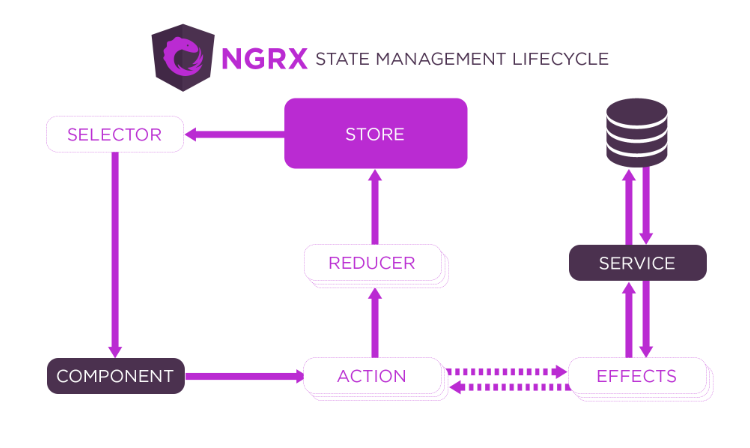
**3. Changes are made with pure functions**

Pure functions are those functions which do not change input value and always provide same output for same input. In Redux/ngRx, reducers are pure functions which takes previous state and an action as parameter and return new state.Since reducers are pure functions, it will never change the previous state, it will always return new state object.

Next, lets see the working of ngRx in angular.

Now that we know what is ngRx all about. Let’s see how an angular app with ngRx works and the key terms used.

The Following diagram represents the flow of logic and data inside the angular-ngRx application.



We have already learnt a lot about angular components and know that it consists of an HTML template which represents data to the user dynamically, helping them to see stuffs and interact as well.

We already know how storing the data across the component tree of the application, can soon become tedious for us to handle. Hence, ngRx says, instead of having the data scattered across components, store and manage them externally outside the component tree, which can then be passed back to the components where it is needed.

The entire application’s state is maintained at a single place called as **‘Store’. Store** more often, can be considered as the client-side database and represents the state of the application. This is where the entire state of the application will be stored.

As the data is stored externally, we need a way to effectively connect the components to the store.

By now, we already know, any change in the UI of the application happens as a result user interaction (events) like click of a button or form submission, etc...

While using ngRx, whenever an event occurs, we **create an action** such as **ADD\_TO\_CART** or **REMOVE\_FROM\_CART**.

Then, based on the action we create, the changes must be applied into the state available in the **ngRx Store**. To apply those changes into the state based on the action, we use **Reducers**.

Once the new **application state** is produced, we need to select the state/data from the **store**that are required in the components and then pass it on to them, this job of selecting the data from the store and then passing on to the components is done by a **Selector**.

So, In Short, we can say, A component creates an **action**based on user **events**, which then goes to the **reducer**, based on the action the reducer updates the **store**and then **selector**passes on the required state to the respective components.

Now, the application might not be that simple always, some user events might also involve asynchronous http calls. To manage such asynchronous operations, we use **Effects**. These ***Effects***will help us invoke the required services to interact with the database and hence bring back appropriate response which will be later used to update the store.

So that was how the angular app works with ngRx. We saw a lot of new terminologies. Next we will see each one of them in detail.

**Action**

**Actions**are one of the main building blocks in NgRx. Actions represent unique events in ngRx based angular application and each action updates the state differently. There are many ways in which an action can be created such as:

* user interaction through clicks, mouseovers, form submissions, etc....
* external interaction through network requests,
* direct interactions with device APIs, etc..

An action is made up of a simple interface containing a property **type**.

1. interface Action {
2. type: string;
3. }

The property **type**describes the action to be dispatched. The value for **type**property will be in the form '**[Source] Event'**where we use **Source**to specify source from where action was created. A simple of action is:

1. { type: '[Counter Component] Increment' }

We can also pass additional data **(also known as payload)** with the action as given below

1. { type: '[Login Page] Login', username: string; password: string;}

***payload:***payload is not compulsory to be passed with every action. The data type of the **payload**depends on the type of data that the actions needs to send to the reducer.

**Action Creators**

Actions can be created using the **createAction** method available from the **@ngrx/store** library. Following are few examples for **createAction**method:

**Example 1: (without payload)**

1. import { createAction } from '@ngrx/store';
2. export const increment = createAction('[Counter Component] Increment');
3. */\* increment() returns { type: '[Counter Component] Increment'} \*/*

**Line 3: increment**is a function that returns an **action**, hence it is called as ***Action creator***

**Example 2: (with payload)**

1. import { createAction, props } from '@ngrx/store';
2. export const login = createAction(
3. '[Login Component] Login',
4. props<{ username: string; password: string }>()
5. );
6. */\* login('user1', 'user1') returns {type: '[Login Component] Login', username:'user1', password:'user1' } \*/*

**createAction** function will return a function that when called will return an object as shown before in this page. props() method helps to **payload** data.

**createAction**just returns an action object which then must be dispatched based on any event that occurs due to user interaction. An action can be dispatched from the components using the **dispatch**method of store as given below:

1. this.store.dispatch(increment()); */\* Without Payload \*/*
3. this.store.dispatch(login("user1", "user1")); */\* With Payload \*/*

Now that we have understood how to create an dispatch actions. Next we will see what are Reducers and how do they work.

**Reducers**are the pure function that handles the state of the application based on the **type**of the action triggered. It takes the previous state and an action as an argument and returns a new state and hence, it transitions the application from one state to another.

There are a few consistent parts of every state managed by a reducer which includes:

* An Interface or a data type to define the type of state
* It will always have two parameters i.e. initial/current state and current action
* function to handle the state changes for the the current action

Lets see this with an example:

First thing we need to do is define the **type of the State** using an interface as given below:

1. export interface State {
2. counter : number;
3. }

Once the state's type has been defined, we also need to **create an initial state** object of the type ***State.***

1. export const initialState: State = {
2. counter: 0
3. };

After defining the initial state, We need to define the reducer function that will handle the different actions.

**Reducers** can be created using the **createReducer**method available from the **@ngrx/store** library. Following is an example to use **createReducer**method:

1. import { createReducer, on } from '@ngrx/store';
2. import { increment, decrement, reset } from './counter.actions'; */\* This file contains all the actions \*/*
3. const \_counterReducer = createReducer(initialState,
4. on(increment, state => { return { ...state, counter: state.counter + 1 } }),
5. on(decrement, state => { return { ...state, counter: state.counter - 1 } }),
6. on(reset, state => initialState),
7. );
8. export function counterReducer(state, action) {
9. return \_counterReducer(state, action);
10. }

**Note:-** The exported **counterReducer**function is necessary as function calls are not supported by the AOT compiler.

In the above example, **counterReducer**handles 3 actions: **'[Counter Component] Increment', '[Counter Component] Decrement'**and **'[Counter Component] Reset'.**Each of these actions are strongly typed and handle the state changes immutably i.e. state transition does modify the original state instead it returns a new state object using spread operator.

When an action is dispatched, ngRx will go through all the reducers by passing the previous state and the Action till it finds a reducer to handle the action. Whether the reducer handles the action or not, is determined by the **on**function.

Next, we will see how to configure the global store for the application.

**Store**is a container that holds the entire application’s state.

The ***store***helps us to keep the application data in one place, and also which enables us to use the store as a single source of truth, i.e. we can reliably access the application's state from the store rather than components of the app holding their own state and passing data between then on need. This reduces the communication between the components, which helps to scale the application without adding more complexity.

To access and manipulate the global **store,**we must first register it within our angular application along with the reducer function that should change the state of the application. To register the global store within our application, we use ***StoreModule.forRoot()*** method with a map of key-value pairs that define our state.

Let us see how to set up an application store:

We can configure the **app.module** with **@ngrx/store** module and reducer as follows, by importing the application **reducer**into the **StoreModule**:

**app.module.ts:**

1. import { BrowserModule } from '@angular/platform-browser';
2. import { NgModule } from '@angular/core';
3. import { AppComponent } from './app.component';
4. import { StoreModule } from '@ngrx/store';
5. import { counterReducer } from './counter.reducer';
7. @NgModule({
8. declarations: [AppComponent],
9. imports: [
10. BrowserModule,
11. StoreModule.forRoot({ count : counterReducer })
12. ],
13. providers: [],
14. bootstrap: [AppComponent],
15. })
16. export class AppModule {}

Registering states with ***StoreModule.forRoot()***ensures that the states are defined upon application startup. In general, we register root states that always need to be available to all areas of your application immediately.

We can also configure the reducers and state for lazy loaded feature modules using **forFeature()**method.

The store also provide us with a method **dispatch**which is used to dispatch **actions**to update the state.

**Advantages of Store based application:**

The primary advantages of a store based application are:

**1. Centralized and Immutable State:**As all the relevant data/state is kept at a single place, it makes the tracking of issues/bugs in the applications easier.

**2. Performance:**As the entire state of the application is centralized, the components can access the updated data directly from there and this improves the performance of the application as the data exchange is much simpler and faster.

**3. Testability:**Reducer functions are pure functions and they are the ones which handle all the state updates. Pure functions are also easier to test as it simply involves input in, assert against output.

**4. Tooling and Ecosystem:**As the state is centralized and immutable, it also empowers us with better tooling. for example, **ngRx developer tools,**which maintain a history of all the dispatched actions and state changes done which allows time travel during development.

Next we will see the use of **Selectors**in ngRx.

***Selectors***are pure functions used for obtaining slices of store state. @ngrx/store provides a few helper functions for optimizing this selection. Selectors provide many features when selecting slices of state such as **Portability, Memoization, Composition, Testability**and **Type Safety**.

We can create selectors using the functions ***createSelector***or ***createFeatureSelector***.

The **createSelector**function can be used to select only the required data for a component from all the entire state. We can pass upto 8 selector functions as parameter to createSelector method for better state selection.

Lets see a simple example with a single selector function:

1. import { createSelector } from '@ngrx/store';
2. export interface FeatureState {
3. counter: number;
4. }
5. export interface AppState {
6. count: FeatureState;
7. }
8. export const selectFeature = (state: AppState) => { return state.count };
9. export const selectFeatureCount = createSelector(
10. selectFeature,
11. (state: FeatureState) => state.counter
12. );

Then to fetch the value of counter function in respective components, we need to use **select**method by passing the selector method i.e. ***selectFeatureCount***as given below:

1. this.count$ = this.store.pipe(select(fromRoot.selectFeatureCount))

When these functions are used to create the selectors, @ngrx/store keeps track of the latest arguments in which the selector function was invoked. As Selectors are pure functions, it can return the last result without re invoking the selector function, if the arguments match with the previous one. This can help in boosting the performance of the application and is also called as ***Memoization***.

Now that we have understood most of the core concepts of ngRx. Lets see a complete Demo for the same.

**What are Meta Reducers?**

@ngrx/store helps us to compose all the reducers together into a single reducer. Meta Reducer is a higher order reducer function that accepts reducers and returns a new reducer.

As developers, we can consider meta reducer as a hook between **action --> reducer** pipeline and it helps pre process the action before the normal reducers are invoked.

Some of the common use cases for ***meta reducers*** are:

* Reset State once a user session terminates
* Loggers to help developers in debugging
* Rehydrating once the app restarts

Meta reducers are the reducer functions which should be executed irrespective of the **action type**.

Meta reducers are very similar to the **middlewares**in redux.

Lets see an example to log all the actions and state changes in the ngrx counter application.