# **NGRX IN ANGULAR APPLICATIONS**

* **Why NgRx**

The Angular application which we have created so far is simple and can be maintained easily.

 But when the complexity of the application grows, with more components, services, which interacts with each other, and when component depends on the other components for the data, managing and handling the complex state across multiple components would become difficult.

And this can lead to inconsistency.

 In this case, we can use an architectural pattern for developing and maintaining large scale applications to streamline the development process.

Here, NgRx is a great design for developing an angular application, which provides state management for creating a maintainable application where unidirectional dataflow is ensured. As the data flow is unidirectional it would be easy to track the event flow.

* **What is NgRx**

NgRx is a great design pattern for predictable state management, which is inspired by Redux.

NgRx aims to bring reactive extensions to Angular and Redux like a single store for the state in an application.

And it helps to write performant, consistent and robust applications by using RxJS which brings this powerful pattern to Angular application.

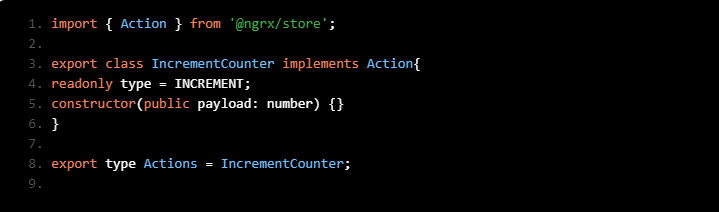
* **Core concepts of NgRx**

1. Actions
2. Reducer
3. Store
4. Selector

**Actions:**

Actions encapsulate specific event which occurs within the application. Actions are created as classes that contain type of action and an optional payload. This action will be passed to the reducer.

Action can be created as follows by implementing the Action interface from @ngrx/store



**Reducers:**

Reducers are the pure function that handles the state of the application based on the action triggered. It takes the previous state and an action as an argument and returns a new state.

Reducer specifies how application’s state changes for the given action type, a switch statement is written that switches through all the action types and does the changes for the mapped action type and return new state as shown below:



In the above code snippet:

1. counterReducer() is a reducer function which is taking 2 parameters, they are

* initialState, which is the state of the application
* action object, which holds action type and payload

2. If the action type is INCREMENT\_COUNT, it takes the copy of state from the store, modifies it and returns the new state. Else it returns the previous state

As reducers are pure function , it will not mutate the state directly.

**Store:**

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.

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:**

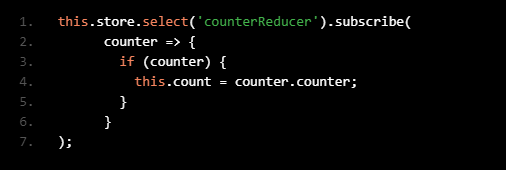


In the above code snippet, **StoreModule.forRoot()** method registers the counterReducer with ngrx/store in the application.

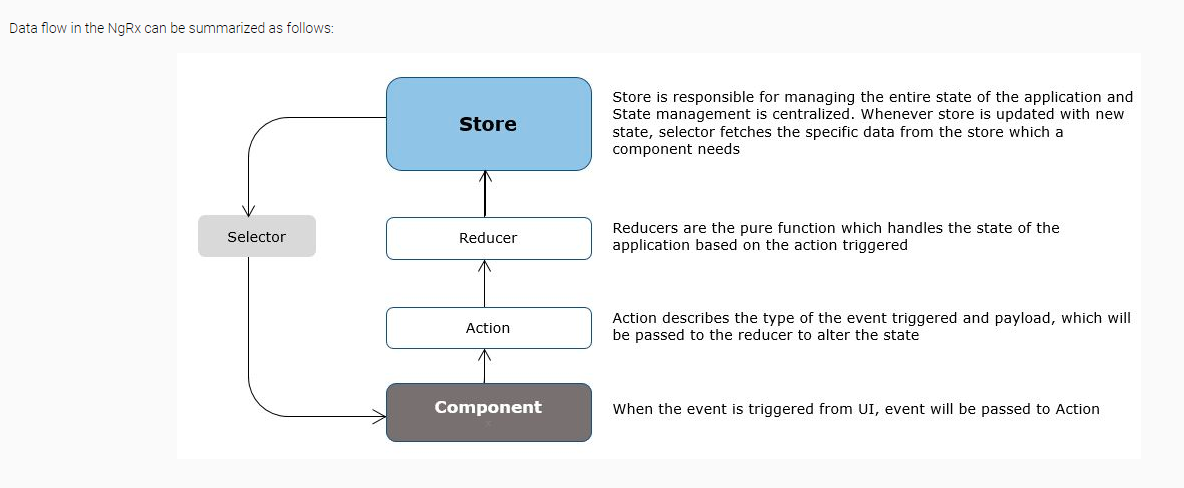
**Selector:**

We have seen how to configure reducer and store in the application, now we can inject ngrx's store service into component to dispatch the actions and to select a slice of application state using Selectors.

Selectors are pure functions that help to fetch slices of the store state. **store.select()** method returns an observable that is bound to a specific property in the application state, as shown below:



In the above code snippet, **counterReducer** represents the name of the slice of state in the store, and data corresponding to **counterReducer** will be returned.



**EXAMPLE DEMO:**

