# **State Management with NgRx**

Using [NgRx](https://ngrx.io/) for state management in an Angular application allows you to build out application flows that track unique events and manage the state of shared data in a reactive, explicit, and consistent way.

## **Overview**

Nx provides a schematic to build out a new NgRx feature area that manages shared state.

The **@nrwl/angular** package has an ngrx schematic to generate files that implement best practices when using NgRx for state management. This schematic generates source files that include enhancements to NgRx for data persistence strategies, and simplified testing.

The ngrx schematic generates an NgRx feature set containing the following files:

* actions - Express unique events throughout your application.
* reducer - Handle state changes from dispatched actions to perform state changes in an immutable way.
* effects - Handle side effects for isolating external interactions from UI components.
* selectors - Composable functions that select pieces of state and update when their inputs change.
* facade - Optional class that provides further encapsulation of NgRx from your component.

The ngrx schematic only provides a sub-set of schematics for the NgRx libraries. See [@ngrx/schematics](https://ngrx.io/guide/schematics) for the full set of available schematics.

## **Command**

The following command is used to run the ngrx schematic:

ng g @nrwl/angular:ngrx <featurename> --module=<path-to-module> --defaults [options]

Copy

Note: the name and the --module= arguments are required. The defaults option chooses the recommended defaults for the schematic, unless you override them.

The most common additional options are:

* root - Set up the initial NgModule imports for NgRx Store, Effects, Router-Store, and Store DevTools.
* syntax - NgRx introduced new creator functions for actions, reducers, and effects that provide the same type-safety with less code than action classes.
* facade - Optional. If you prefer to further encapsulate NgRx from your components, add an injectable facade. See the blog [Better State Management with Facades](https://blog.nrwl.io/nrwl-nx-6-2-angular-6-1-and-better-state-management-e139da2cd074" \l "cb93) for details.

See the [API Docs](https://nx.dev/latest/angular/plugins_angular_schematics/ngrx) for detailed descriptions of all the available options. Also visit the [NgRx](https://ngrx.io/) website for more guides and documentation about the libraries.

## **Initial Setup**

To get started with NgRx in an Angular application, you set up the root level store. As your application grows, you add feature level states, ensuring that your code follows a common pattern each time.

The example below shows you how to setup NgRx in the root of your application.

ng g @nrwl/angular:ngrx app --module=apps/<appname>/src/app/app.module.ts --root

Copy

The above command applies the following changes to the provided module:

* Registers StoreModule.forRoot({}) in the imports array for state management, with recommended runtime checks enabled for maintaining immutable actions and state.
* Registers EffectsModule.forRoot([]) in the imports array for isolation of side effects.
* Registers StoreRouterConnectingModule.forRoot() in the imports array for integration with the [Angular Router](https://angular.io/guide/router).
* Registers StoreDevtools.instrument() in the imports array for integration with the [Redux Devtools browser extension](http://extension.remotedev.io/).

You manage separate slices of state using libraries and feature states.

## **Feature Workflow**

When building new features using NgRx, you want to manage the state from within a separate library. This allows your state to be easily shared across other libraries and applications. The steps below go through the workflow of using NgRx within the context of a library.

The example below generates a library to begin a new feature. For this example, products is used as the library name.

ng g @nrwl/angular:lib products

Copy

To manage the feature state:

* Use the ngrx schematic with the feature name in plural form, such as products.
* Provide a path to the products library module.

ng g @nrwl/angular:ngrx products --module=libs/products/src/lib/products.module.ts --directory +state/products --defaults

Copy

Use the --facade option to generate an injectable Facade class along with the feature.

The following files are created, or updated:

myorg  
├── apps  
└── libs  
 └── products  
 └── src  
 └── lib  
 │ └── +state  
 │ │ ├── products.actions.ts  
 │ │ ├── products.effects.ts  
 │ │ ├── products.effects.spec.ts  
 │ │ ├── products.facade.ts # optional  
 │ │ ├── products.facade.spec.ts # optional  
 │ │ ├── products.models.ts  
 │ │ ├── products.reducer.ts  
 │ │ ├── products.reducer.spec.ts  
 │ │ ├── products.selectors.ts  
 │ │ └── products.selectors.spec.ts  
 │ ├── products.module.spec.ts  
 │ └── products.module.ts  
 └── index.ts

Copy

The above command also does the following changes:

* Updates the feature module and registers StoreModule.forFeature() with the name of your feature state in the imports array.
* Updates the feature module and registers EffectsModule.forFeature() in the imports array.

The feature library's barrel index.ts is also updated to export the updated public API for the state including:

* The NgRx selectors.
* The NgRx feature reducer.
* The optional facade class for the NgRx feature.