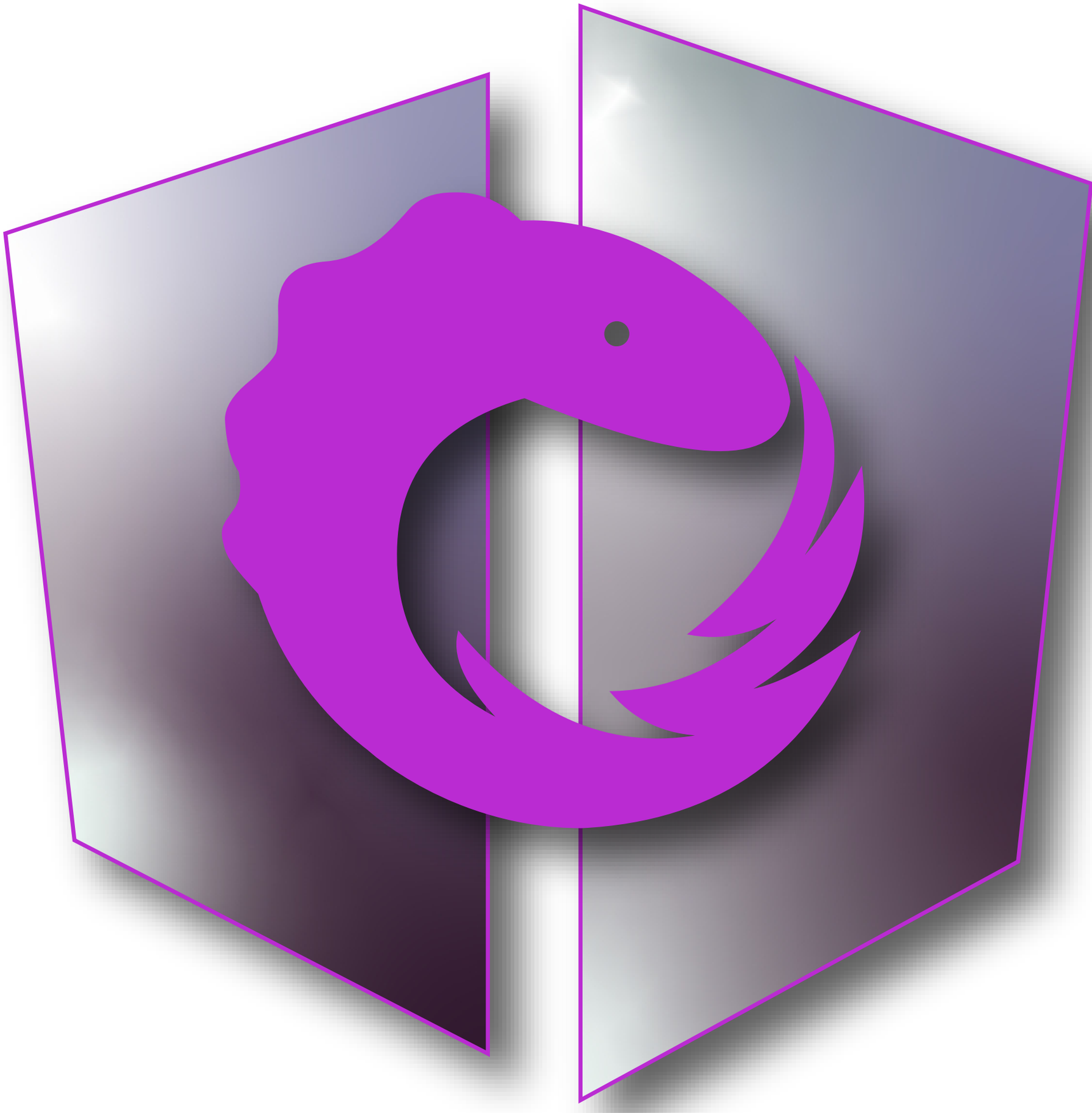
NGRX



NGRX: The Basics

Introduction

Redux Pattern

Getting Started

Store

State

Initialize State

Feature State Management

Strong Typing

Creating Feature Selectors

Actions

Reducer

Utilizing Feature Selectors

Effects

Dev Tools

The Basics

Introduction

What is NgRx:

NgRx is a framework for building reactive applications in Angular. Ngrx provides libraries for:

- Managing global and local state.

- Isolation of side effects to promote a cleaner component architecture

- Entity Collection management

- Integration with the Angular Router

- Developer tooling that enhances developer experience when building many different types of applications

In this NgRx kick starter we will try to help you become acquainted with the basic operation of RgRx. We will focus on the highlighted functionality listed above and the advantages of NgRx implementation. As your application's complexity increases
you will the need for @ngrx/entity and @ngrx/router-store.

How is NgRx pronounced?
Simple, it is just spelled out "N G R X"

What Does NgRx mean?
Like stated above NgRx is a framework for building reactive applications in angular. This means some very smart developers combined the Redux Pattern with Angular and created a host of libraries and tools to manage your application's state.

Redux

State / Store

Reducers

Actions

Payloads

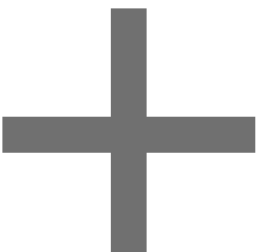
Effects

Angular

Component

View

NgRx =



What are the advantages of NgRx?

‘Redux is not great for making simple things quickly. It is great for making really hard things simple’

– Jani Evakallio

NgRx simplifies a complex problem by standardizing the process of dealing with a shared state. A shared state grows in complexity as the application grows as more components become dependent on the shared state and can effect the shared state at any time. Implementing NgRx can

- Simplify code

- Easier to track changes

- Easier to debug and test

- Implement an angular change detection strategy in components -improve performance

- Testing Reducers are easier to test

Tooling Advantages

- History of state changes

- Visualize state tree

- Debug by undoing and redoing state changes

Advanced Logging

Redux Pattern

The redux pattern is very handy for updating and tracking changes to a data store that multiple resources rely on for up-to-date information simultaneously. How does this pattern keep all components dependent on a single source synchronized to one data source?

1. It uses selectors to get the latest information that is stored in the store/state.
2. This information is gathered in the component and displayed in the view.
3. When the user interacts with view and changes the state of the application this information and new state will be packaged into actions in the Component and sent to the reducers.
4. The reducers are listening for specific actions that carry payloads of new information about the changes to the state
5. The reducers copy the existing data and augment it with the payload from the actions carrying the new changes to state using a spread operator, creating the new state.
6. The new state is saved to the store
7. The selectors that have been listening for changes to the state, notify all the components subscribed simultaneously of the new state.

Actions
Payloads
Effects

Redux Pattern

The redux pattern is very handy for updating and tracking changes to a data store that multiple resources rely on for up to date information simultaneously. How does this pattern keep all components dependent on a single source synchronized to one data source?

1. It uses selectors to get the latest information that is stored in the store/state.
2. This information is gathered in the component and displayed in the view.
3. When the user interacts with view and changes the state of the application this information and new state will be packaged into actions in the Component and sent to the reducers.
4. The reducers are listening for specific actions that carry payloads of new information about the changes to the state
5. The reducers copy the existing data and augment it with the payload from the actions carrying the new changes to state using a spread operator, creating the new state.
6. The new state is saved to the store
7. The selectors that have been listening for changes to the state, notify all the components subscribed simultaneously of the new state.

The process happens over and over again. The store cannot be changed directly and must follow this strict process.

The process happens over and over again. The store cannot be changed directly and all changes to the store must follow this strict process in order to protect the integrity of the data.

Getting Started

Lets install NgRx:

Prerequisites: Node, NPM, Angular

```
> npm install @ngrx/store --save
```

Import StoreModule into the project imports in your app.module.ts file :

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

@NgModule({
imports: [
                ...
                StoreModule.forRoot({ }),
                ]
})
```

app > app.module.ts

Congratulations you have now added NgRx to your project.

State - Immutable Store

Every angular application must manage its' state. As your application grows so does the size of the state that the application has to manage. Larger state means more application to manage more information, more state to manage, more actions and reactions to state changes to track and test. Very powerful when all the pieces of application state is held in one place the store because it will help with reasoning about user interaction, debugging, performance and avoiding race conditions.

The store is what we call an immutable store. Immutability is paramount to the integrity of the store. Immutability insures the data is accurate across all components by insuring there is only one path to changing the store. For the redux pattern to work you must follow the redux principals of immutability:

- If you need to change state of the store always replace the whole state object and not mutate part of it

- Only use actions to change the state of the store

- Data that never changes is easier to reason about, track, and test.

What kind of information can I store in my State/Immutable Store?

Do's

Don'ts

View State

Unshared State

User Information

Angular form state

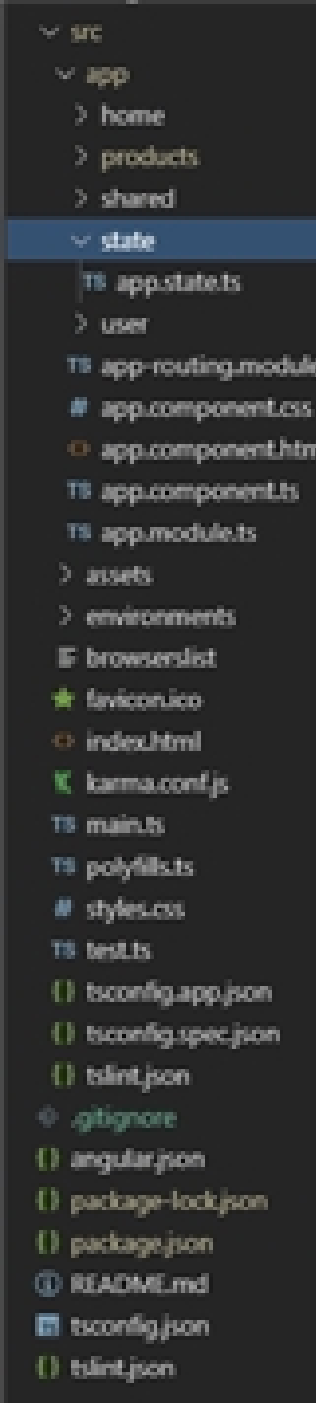
Entity Data

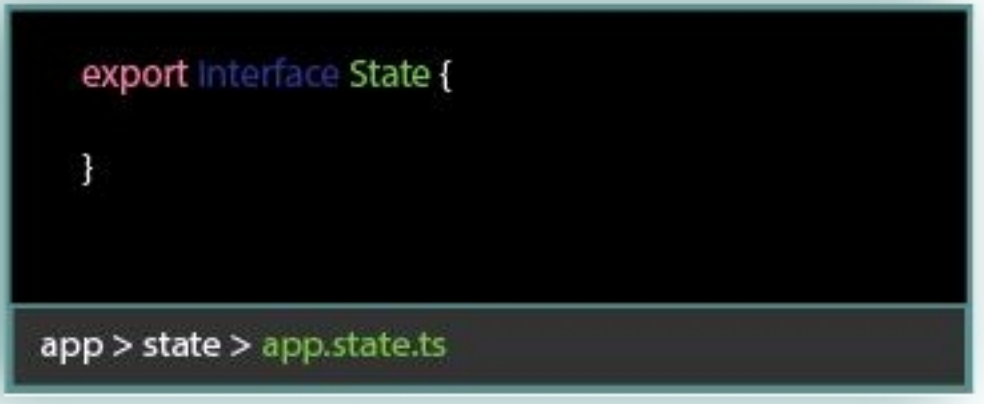
Non-serializable state - Router State

User Selection and Input

Initialize State

Now that you have NgRx running and we need to add initialize our store with some data. Create app.state.ts file in the state folder at the root of your project. In there you will initialize your State

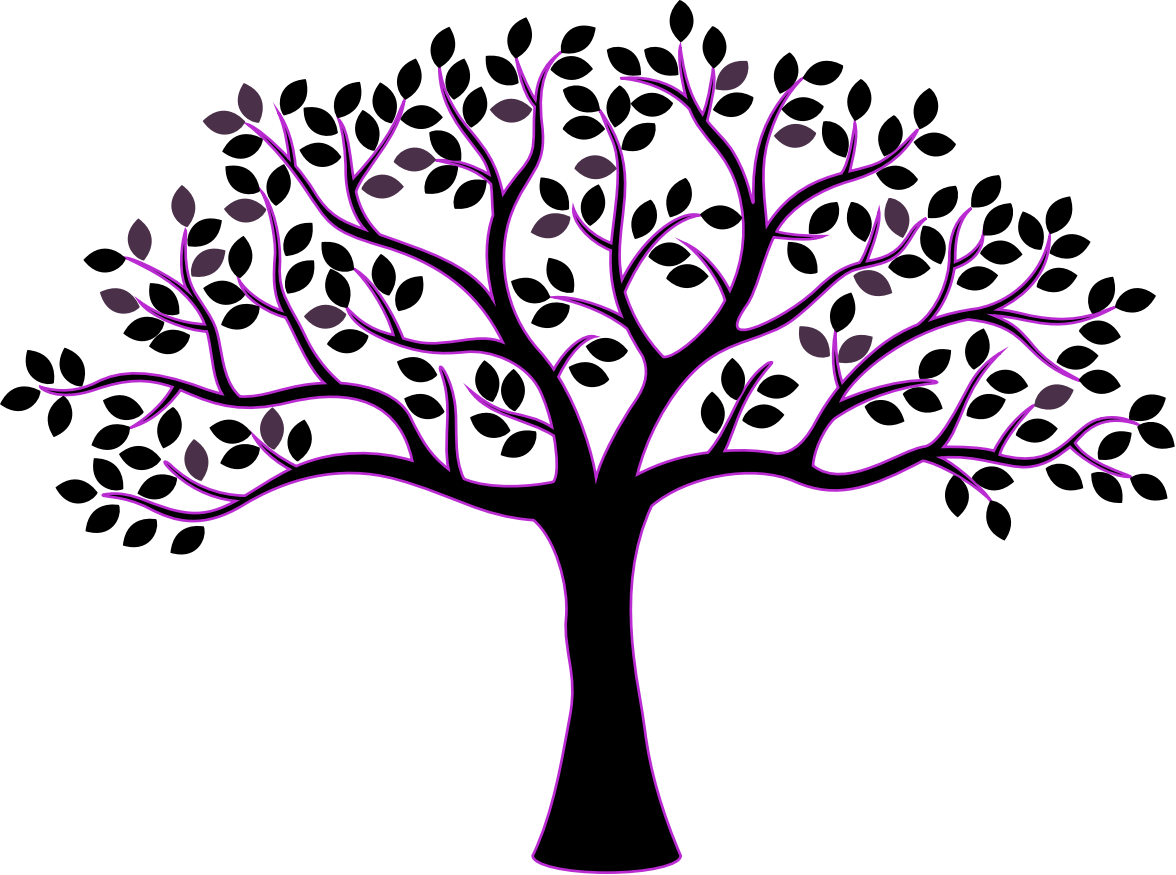

src
app
home
products
shared
state
app.state.ts
user
app-routing.module.ts
app.component.css
app.component.html
app.component.ts
app.module.ts
assets
environments
browserslist
favicon.ico
index.html
karma.conf.js
main.ts
polyfills.ts
styles.css
test.ts
tsconfig.app.json
tsconfig.spec.json
tslint.json
.gitignore
angular.json
package-lock.json
package.json
README.md
tsconfig.json
tslint.json


export Interface State {
}
app > state > app.state.ts

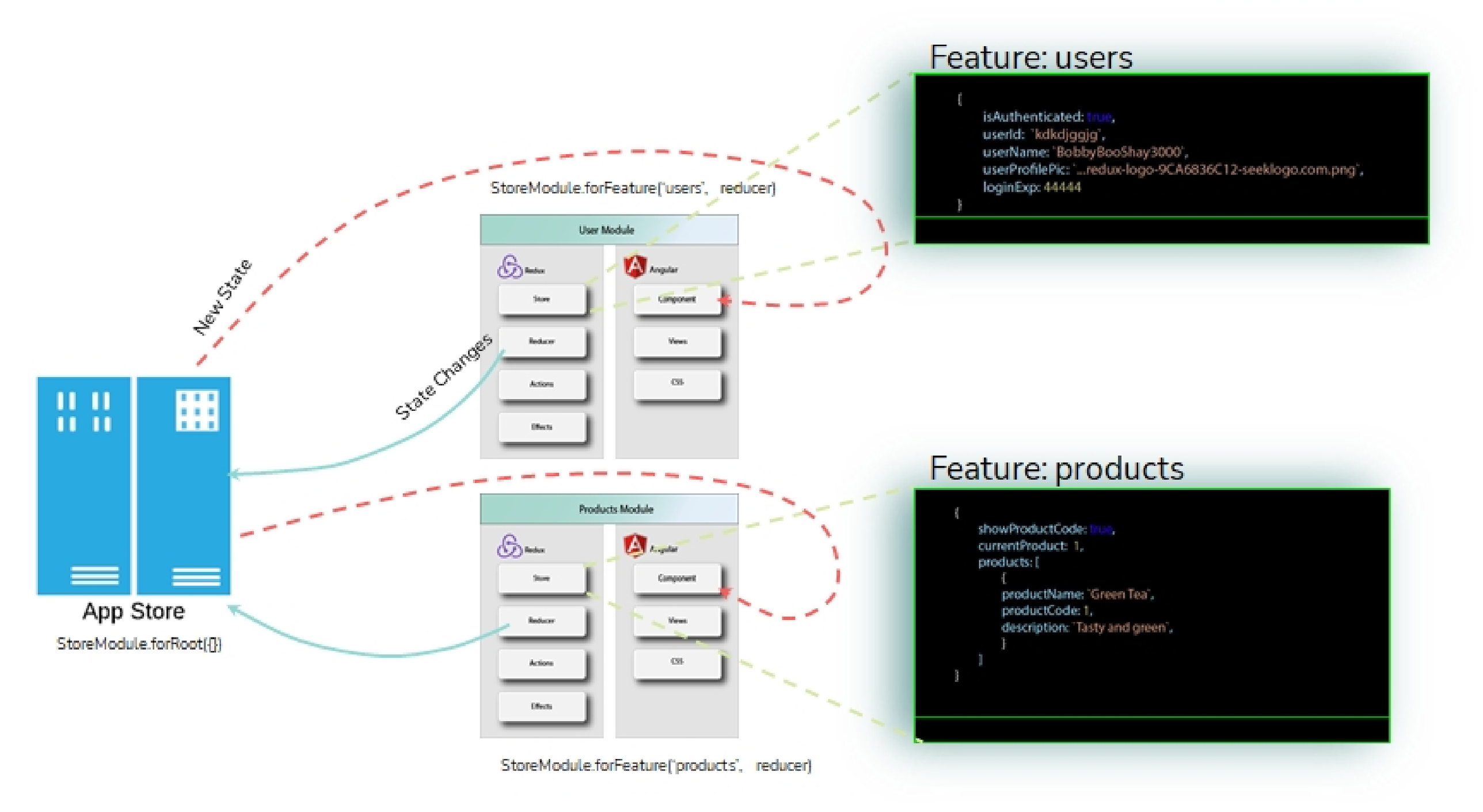
When the StoreModule initializes the store it will implement the content in this folder at the root This is the content that is loaded that will operate across all modules and features.

Feature State Management

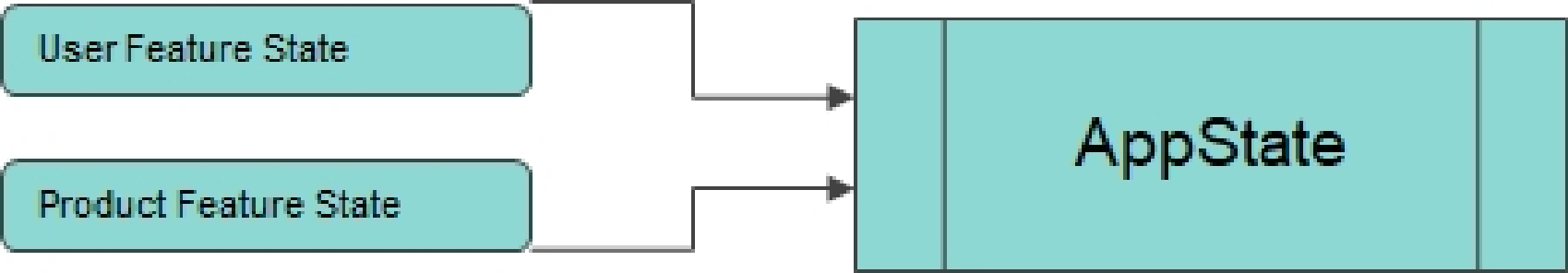
Your store is a JavaScript object that is destroyed when the application is restarted. It operates as a client side database that holds the state of the application. Each bit of state is defined by a property. This state will grow as the application grows and become unmanageable. But when married to the logical angular feature module scheme the state can grow along with the application and maintain functionality. Therefore a portion of the state that corresponds with a feature will be considered a slice and named according to its respective slice as in ‘products slice’ or ‘user slice’. Store operates like an in-memory database and is destroyed when the browser is refreshed



The application will function as a tree of features and functionality. The branches of the tree represent independent features. Each feature will implement its own Redux pattern, complete with its own store, reducer, actions, selectors and effects. The features manage their own slice of the store, which helps simplify management of large applications that's complexity would otherwise be unmanageable.


Feature: users
{
isAuthenticated: true,
userId: `kdkdjggjg`,
userName: `BobbyBooShay3000`,
userProfilePic: `...redux-logo-9CA6836C12-seeklogo.com.png`,
loginExp: 44444
}
StoreModule.forFeature('users', reducer)
User Module
Redux
Angular
Store
Reducer
Actions
Effects
Component
Views
CSS
New State
State Changes
App Store
StoreModule.forRoot({})
Products Module
Redux
Angular
Store
Reducer
Actions
Effects
Component
Views
CSS
StoreModule.forFeature('products', reducer)
Feature: products
{
showProductCode: true,
currentProduct: 1,
products: [
{
productName: `Green Tea`,
productCode: 1,
description: `Tasty and green`,
}
]
}

This redux pattern will work hand in hand with the feature's Components and Views. In the figure above you can see product information and user information is managed in their respective feature module. For the application to recognize these branches of the state they must be imported into their feature modules. The definition of feature state resides in the feature's reducer file and they are imported into their feature modules. And the each module feature state subsequently imported into the AppState. As seen below...


User Feature State
Product Feature State
AppState


import { StoreModule } from '@ngrx/store';
import { reducer } from './state/user.reducer';
@NgModule({
imports: [
...
StoreModule.forFeature('users', reducer),
]
})
app > users > user.module.ts


import { StoreModule } from '@ngrx/store';
import { reducer } from './state/producer.reducer';
@NgModule({
imports: [
...
StoreModule.forFeature('products', reducer),
]
})
app > products > product.module.ts

Strong Typing - Feature State Management