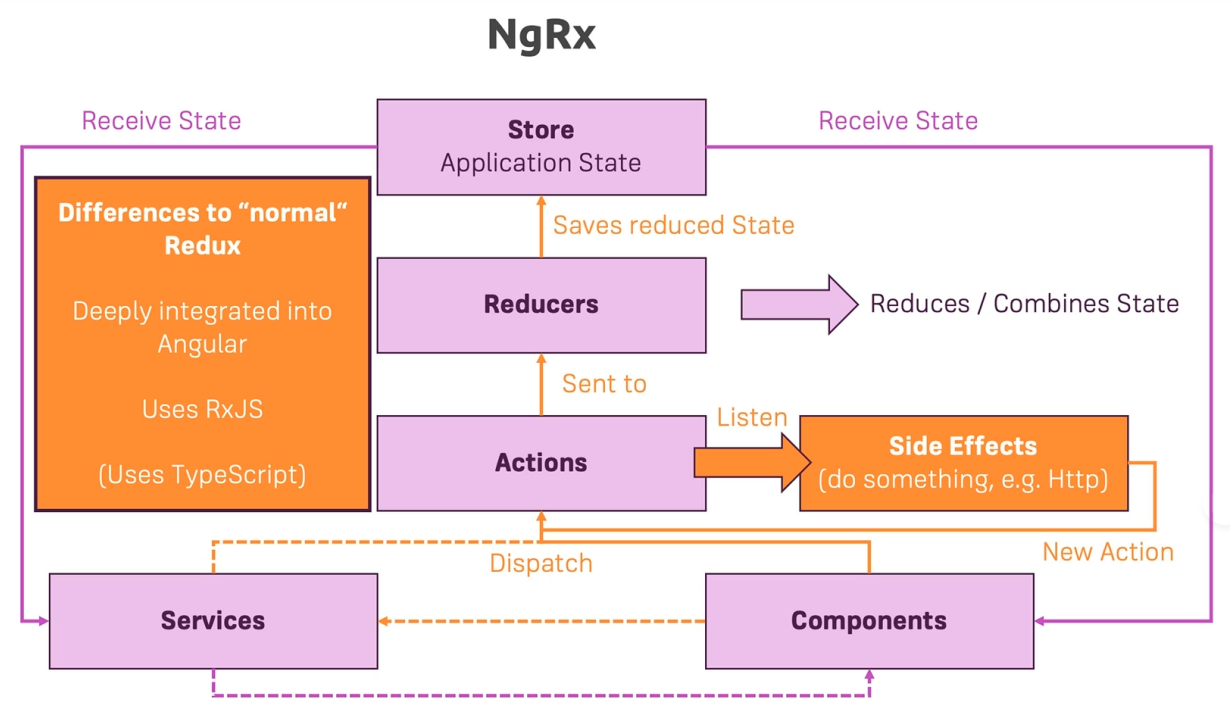
Working with NgRx in Our Project

What is Application State?

- The **current** state of the application
  - Includes the **current** component, fetching/storing, etc
  - Regards whatever data is **present** on the **screen**
- Is **lost** whenever the application **refreshes**
  - We use a **back-end** to **retain** data
  - **Back-Ends** are a **persistent state**
- The **bigger** the **application**, the bigger the **state**
  - This becomes a **nightmare**
  - Partly remedied with **RxJS**
    - Let's us **react** to **specific events** carefully

What is NgRx?

- **Issues** with **RxJS** Approach
  - The state can be **updated anywhere**
  - The state is (possibly) **mutable**
  - Handling **side effects** (e.g. Http calls) is **unclear**
  - Therefore, **no specific pattern** is **enforced**
- **Redux** is a state-management pattern/library
  - Has **one central store** maintaining the application state
  - **Services** and **components** may still **interact** with each other, but they each **receive** their **states** from the **store**
  - We **update** the **state** by dispatching **actions** from the services and components
    - Actions include **identifiers** and **payloads**
  - **Actions** are sent to **reducers**, which are JavaScript functions that **merge** the **current state** with the **action**



Getting Started with Reducers

- Within your logical component sphere, add a **<file-name>.reducer.ts** file
- **Export** a reducer function
  - Contains **state** and **action** parameters
- This file may also contain an **initial state**
  - Written as a **JavaScript object**
  - Would be the **default argument** in the **reducer function**

Adding Logic to the Reducer

- Can use **if**s or a **switch** statement to read the **action's type**
- **States** are **immutable**
  - Instead, we **return** a **new object representing** the new state
  - Use the **spread operator** to fetch existing state information, and include new data separately

Understanding and Adding Actions

- We should handle actions inside a **<file-name>.actions.ts** file
- This **action** file contains the **action-types** and their string values, which we reference in the **if/switch** statements within the **reducer function**

- **Action-types** are also accompanied by a **class**
  - This class implements **Action** from **NgRx**
  - Contains a **readonly type** field, sharing the **type's name**
  - Contains a **payload** field, which should be referenced with the **action** in the **reducer**

Setting Up the NgRx Store

- Must import **StoreModule** within **AppModule**
- Inside the **imports** array, apply the **forRoot** method to **StoreModule**
- This method takes a **JavaScript Object** containing key/value pairs
  - **Keys** are how we reference **identifiers**
  - Values are **reducers**
- Ensure that your version of **NgRx** matches your version of **Angular**

Selecting State

- Within your **component**, inject a **store** field
  - This is of type **Store<{}>**
  - The object contains a **key** that matches your desired **key** from **AppModule's StoreModule**, and the **value** is the type of whatever data is changing
- In **ngOnInit**, invoke and store an **observable** with **this.store.select('<id-name>')**
- Don't forget to provide a **default case** in the **reducer**

Dispatching Actions

- Similar to **select**, we can invoke **this.store.dispatch(<action-shenanigans>)**
- We pass a **new Action** as a parameter, appropriate one from our **<file-name>.actions.ts** file

Multiple Actions

- Add more **fields** and **classes** to the **reducer file** as necessary

- Within the **Actions** file, export a **type** equal to all **field identifiers piped** together
- Set the **reducer's** original **action** to the new **type**
- Add new **if's/cases** for the new **types**

Expanding the State

- You should **export interfaces** depicting your desired **states** when referencing them in other files

One Root State

- It's good practice to keep a **global state** that ties in **all reducers**
- At **root**, create a new **store** directory with an **app.reducer.ts** file
- File Contents
  - This file should contain an **AppState** interface with references to the **State** interfaces of all **reducers**
  - Instead of listing all **AppStates** in **AppModule.imports.StoreModule.forRoot**, only insert **appReducer**
    - This entails applying a wildcard import with an alias
- All **components** that originally referenced their own **AppStore** objects should now be referencing this new **AppState**

An Important Note on Actions

- We also return **state** by **default** in our **reducers**
  - This is important for **initializing** the **state** → Our **initialState** model
- **All dispatches** reach **all reducers**
- You must always **copy** the **original state** within each **case**
- We must ensure that all **action identifiers** are **unique** as they reach everywhere
  - It's a good idea to **prefix identifiers** → **'[Shopping List] Add Ingredient'**

Exploring NgRx Effects

- **Side effects** are components of code that, while important, aren't critical for the **current state's update** when you **execute logic**

- **@ngrx/effects** let's us **handle side effects** between dispatches

Defining the First Effect

- Within your **component-specific store** folder, add **<file-name>.effects.ts**
- Within the file, **export** a **class** and **inject Actions** from **@ngrx/effects**
- Here, we can **execute** any other **code** when an **action** is **dispatched**
  - You can then **dispatch** another **action** when that code is done
- We list our **effects** at the **top** of the **class**
  - Assign them to the **injected actions**
  - Use **pipe** off the assignment
    - Apply **ofType()** within the pipe → Defines the **types** of **effects** you with to **apply** within the **pipe**
    - **Actions** handles subscription automatically

Effects and Error Handling

- This came up late, but add the **@Effect** ()**decorator** to each **effect**
- After **ofType** (from before), add **switchMap**
  - This allows us to **create** a **new observable** by taking **another observable's data**
- All **effects** should **return** an **action** (since we're in an **observable chain**, we're returning an **observable**) upon **completion**

Login via NgRx Effects

- Wiring up **effects**
  - Add the **@Injectable() decorator** to the **AuthEffects** class
  - Within **AppModule**, import **EffectsModule**
    - This should be referenced in the **imports** array with a **forRoot** method that contains an **array of effects**

Using the Store DevTools

- Browsers have a special **extension** called **Redux DevTools**

    - This extension gives us **updates** on our application's **state** at the **time of change**
- We pair it with Angular using the **@ngrx/store/devtools** npm package
    - Within **AppModule**, import **StoreDevtoolsModule** and call the **instrument** method
    - Pass in an **object** → **{ logOnly: environment.production }**

The Router Store

- Within **AppModule**, import **StoreRouterConnectingModule** and call the **forRoot** method
    - This requires the **@ngrx/router-store** npm package
    - This method takes **no arguments**
- This provides us a straightforward way of seeing **routing events** and their **data**