Actions

Overview

Actions parcel up information that is used to add to or modify the stored data in our store. This app state data is created by user events in the views. The action operates as a discreet mechanism to transport changes in app state data from the components to be the reducer. Once the information is in the reducer it is processed through a switch statement and modifications to the store will be applied. The NgRx Action is not just the critical path in change control of the app state but the only path to state manipulation. This insures the state integrity across the entire application.

How and When Do I Create Actions
Actions are created based upon how the store's data will be modified. Actions are declared along side the state properties because they define how the values of the state properties can be changed. Actions are generally defined in their own file and exported so that they can be accessed from the components and anywhere the feature module is imported.

An Action is a simple JavaScript object consisting of two properties, the type and the payload.

Type: is a unique string describing what the action does, what information the action is responsible for manipulating
Payload: is the object that carries the change of state.

This is an example of a basic action:

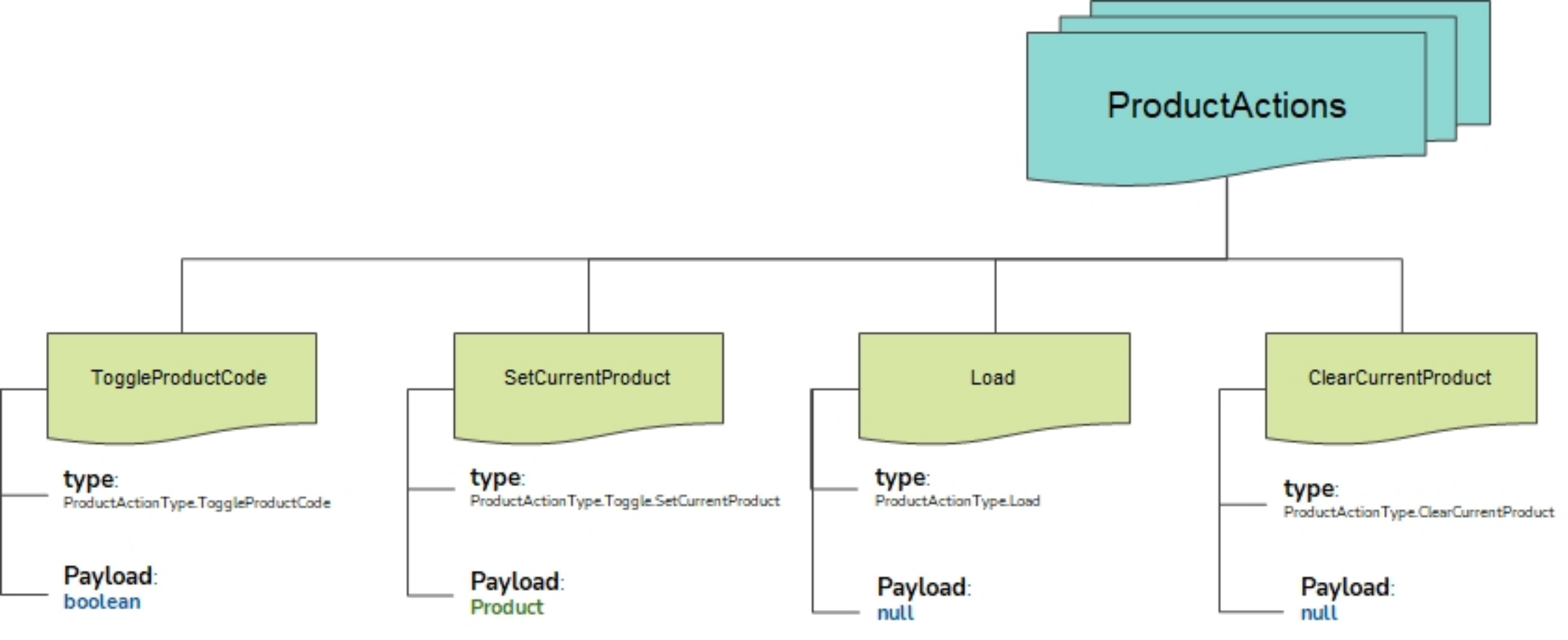
```
{    type: 'LOGIN',
     payload: {
                  username; 'apexBugFinder',
                  password: 'secret'
                }
}
```

Actions are recognized by the reducer via the type property in a switch statement. Hence the requirement for them to be unique. It is best practice to define your type properties via enum, so that you define them once and you can bypass any problems with typos that can occur. This enum listing will function as an index and action definition listing (or what i like to refer to as an action dictionary). Declaration type via enum immediately transforms this fragile process into a strongly typed dependable process.

```
export enum ProductActionTypes {
        ToggleProductCode =
                '[Product] Toggle Product Code',
        SetCurrentProduct =
                '[Product] Set Current Product',
        ClearCurrentProduct =
                '[Product] Clear Current Product'
}
```

app > products > state > product.reducer.ts

As we are building Typescript we can protect our State but strongly typing our payload to add another layer of protection to the integrity of our state data. Allowing for an error to be thrown if the data being passed does not match the defined type for that payload.


ProductActions
ToggleProductCode
type:
ProductActionType.ToggleProductCode
Payload:
boolean
SetCurrentProduct
type:
ProductActionType.Toggle.SetCurrentProduct
Payload:
Product
Load
type:
ProductActionType.Load
Payload:
null
ClearCurrentProduct
type:
ProductActionType.ClearCurrentProduct
Payload:
null

```
export class ToggleProductCode implements Action {
        readonly type =
                ProductActionTypes.ToggleProductCode;
        constructor (public payload: boolean) { }
}

export class SetCurrentProduct implements Action {
        readonly type =
                ProductActionTypes.SetCurrentProduct;
        constructor (public payload: Product) { }
}

export class ClearCurrentProduct implements Action {
        readonly type =
        ProductActionTypes.ClearCurrentProduct;

}
```

app > products > state > product.reducer.ts

As we are building Typescript we can protect our State but strongly typing our payload to add another layer of protection to the integrity of our state data. Allowing for an error to be thrown if the data being passed does not match the defined type for that payload.

Once our Type definitions and actions have been defined we then can export them as a union type. The naming convention is that you use the

```
export type ProductActions =
        ToggleProductCode =
        | SetCurrentProduct =
        | ClearCurrentProduct
        ;
}
```

app > products > state > product.reducer.ts

Dispatching Actions

Now that we have actions defined we can now dispatch them in the components with ease and with very little amount of code needed.

```
checkChanged(value: boolean): void {
  this.store.dispatch(
        new productActions.ToggleProductCode(value));
}



newProduct ( product: Product): void {
  this.store.dispatch(
        new productActions.InitializeCurrentProduct());
}



productSelected(product: Product): void {
  this.store.dispatch(
        new productActions.SetCurrentProduct(product));
}
```

app > products > product-shell > product.-shell.component.ts

Reducers