An opinionated but pragmatic approach to writing large-scale Angular 2 applications.

We welcome new ideas and perspectives on improving it. Feel free to contribute by opening issues, forking, cloning, branching, committing, pushing, PR-ing, etc…

Table of Contents

1: Directory structure

```
src
├── actions
│   ├── cart-actions.ts
│   ├── order-history-actions.ts
│   ├── profile-actions.ts
│   ├── store-front-actions.ts
│   └── ...
├── assets
│   ├── brand-logo.svg
│   └── ...
├── components
│   ├── search-bar
│   │   ├── search-bar.css
│   │   ├── search-bar.html
│   │   ├── search-bar.test.ts
│   │   └── index.ts
│   ├── product-card
│   │   ├── product-card.css
│   │   ├── product-card.html
│   │   ├── product-card.test.ts
│   │   └── index.ts
│   └── button
│       ├── button.css
│       ├── button.html
│       ├── button.test.ts
│       └ index.ts
├── constants
```

```
│   ├── cart-constants.ts
│   ├── order-history-constants.ts
│   ├── profile-constants.ts
│   ├── store-front-constants.ts
│   ├── ...
│   └── index.ts
├── containers
│   ├── app.ts
│   ├── cart.ts
│   ├── order-history.ts
│   ├── profile.ts
│   ├── store-front.ts
│   └── ...
├── reducers
│   ├── cart-reducer.ts
│   ├── order-history-reducer.ts
│   ├── profile-reducer.ts
│   ├── store-front-reducer.ts
│   ├── ...
│   └── index.ts
├── store
│   ├── configure-store.ts
│   ├── ...
│   └── index.ts
├── services
│   ├── api-service.ts
│   ├── models-service.ts
│   ├── order-management-service.ts
│   └── ...
├── pipes
│   ├── currency-pipe.ts
│   └── ...
└── utils
    ├── api-client.ts
    ├── api-client.test.ts
    ├── models.ts
    ├── models.test.ts
    ├── currency.ts
    ├── currency.test.ts
    ├── order-management.ts
    ├── order-management.test.ts
    └── …
```

✓ **utils:** contains the majority of business logic, implemented as plain JavaScript libraries, and is the initial source of application functionality (see Packaging for advice on managing growth and changes in this folder).

- ✓ **actions:** contains Redux action helpers that heavily utilize /utils libraries in order to dispatch actions.
- ✓ **assets:** contains non-code assets to be bundled with the application.
- ✓ **components:** contains state-less view components that take data "from above" and present a UI.
- ✓ **containers:** contains stateful, and often routable, components that pass down data and behaviour-encapsulating callbacks to presentational components (and sometimes other containers).
- ✓ **constants:** contains Redux constants which are typically action types.
- ✓ **reducers:** contains Redux reducers that respond to dispatched actions and leverage models and data manipulation functions from /utils to execute application state transitions.
- ✓ **store:** contains helpers that build a Redux store.
- ✓ **services:** contains /utils libraries wrapped in Angular 2 services to take advantage of dependency injection, where appropriate.
- ✓ **pipes:** contains /utils libraries wrapped in Angular 2 pipes to be executed directly on templates, where appropriate.

2: Key Points:

- ✓ Follow the **syntax and conventions** provided by angular, it makes your code easier to understand. Link: https://angular.io/guide/styleguide
- ✓ **Do:** Always use **directives** to interact with **dom**, like filtering of data using input box.
  Why: Sometimes people uses components in place of directives, which increasing application logic processing and time of dom rendering.
- ✓ **Do:** Always write module level logic when needed, modules are easy to share from one place to another.
- ✓ **Do:** Make use of **Immutable objects**: They are thread safe, also it will not change once it is created.
  NOTE: Objects and Arrays are pass by reference.
  Why: Mutation doesn't guaranty that something will stay unchanged. For example, if the same object is used in separate parts of the application, then any mutation(change) in one component, can create a bug in the other one. Such bug is really hard to track, because the source of the problem lies out of the scope of the failure.

```
const person = {
        name: 'Pritam',
        age: 28
}

const newPerson = person

newPerson.age = 30

console.log(newPerson === person) // true
console.log(person) // { name: 'John', age: 30 }
```

Instead of passing the object and mutating(change) it, it is better to create a completely new object:

```
const person = {
  name: 'John',
  age: 28
}

const newPerson = Object.assign({}, person, {
  age: 30
})

console.log(newPerson === person) // false
console.log(person) // { name: 'John', age: 28 }
console.log(newPerson) // { name: 'John', age: 30 }
```

✓ **Do:** Write **Pure Functions**: Always returns the same set of values for the same inputs.

Why: Pure Functions guaranties that state of any object will stay unchanged. A pure function has no side effects.

For example: **without pure function**

```
const x  = {
        val: 2
}
const x1 = () => x.val += 1;
const x2 = () => x.val *= 2;


x1();
```

```
x2();
```

It will output : 3, 6

Now if we change the order of calling the function like x2(); x1()
The output will change to 4, 5

Using pure function:

```
const x  = {
        val: 2
}
const x1 = x => Object.assign({}, x, { val: x.val + 1});
const x2 = x => Object.assign({}, x, {val: x.val * 2});


x1(x).val;
x2(x).val;
```

It will output : 3, 4

- ✓ **Do:** Always **unsubscribe** your subscribers.
  Why: Avoid memory leaks
- ✓ **Don't:** Avoid writing **jquery or javascripts** in your components, instead we can go with directives.
- ✓ **Do:** Write **Micro Components**: Writing micro components allow reusability of components, also reduces writing or copy-pasting code again and again.
- ✓ **Do:** Avoid writing **promises**, instead we can use rxjs features.
  Why:
  a. Promise emits single value, where as an observable emits multiple values over a period of time.

```
const promise = new Promise(resolve => {

  setInterval(() => {
        console.log("Promise resolved");
        resolve('Hello from a Promise!');
  }, 2000);
});
```

```
promise.then(value => console.log(value));
```

```
const observable = new Observable(observer => {

setInterval(() => {
   console.log(“Observable resolved”);
   observer.next('Hello from a Observable!');
  }, 2000);
});

observable.subscribe(value => console.log(value));
```

b. Promise are not cancellable, where as an observable can be cancelled any time.