Introduction to Redux with TypeScript

> In this chapter and the next we'll be looking at a data-architecture pattern called Redux. In this chapter we're going to discuss the ideas behind Redux, build our own mini version, and then hook it up to Angular. In the next chapter we'll use Redux to build a bigger application.

In most of our projects so far, we've managed state in a fairly direct way: We tend to grab data from services and render them in components, passing values down the component tree along the way.

Managing our apps in this way works fine for smaller apps, but as our apps grow, having multiple components manage different parts of the state becomes cumbersome. For instance, passing all of our values down our component tree suffers from the following downsides:

Intermediate property passing - In order to get state to any component we have to pass the values down through `inputs`. This means we have many intermediate components passing state that it isn't directly using or concerned about

Inflexible refactoring - Because we're passing `inputs` down through the component tree, we're introducing a coupling between parent and child components that often isn't necessary. This makes it more difficult to put a child component somewhere else in the hierarchy because we have to change all of the new parents to pass the state

State tree and DOM tree don't match - The "shape" of our state often doesn't match the "shape" of our view/component hierarchy. By passing all data through the component tree via `props` we run into difficulties when we need to reference data in a far branch of the tree

State throughout our app - If we manage state via components, it's difficult to get a snapshot of the total state of our app. This can make it hard to know which component "owns" a particular bit of data, and which components are concerned about changes

Pulling data out of our components and into services helps a lot. At least if services are the "owners" of our data, we have a better idea of where to put things. But this opens a new question: what are the best practices for "service-owned" data? Are there any patterns we can follow? In fact, there are.

In this chapter, we're going to discuss a data-architecture pattern called *Redux* which was designed to help with these issues. We'll implement our own version of Redux which will store **all of our state in a single place**. This idea of holding **all** of our application's state in one place might sound a little crazy, but the results are surprisingly delightful.

Redux

If you haven't heard of Redux yet you can read a bit about it on the official website[98]. Web application data architecture is evolving and the traditional ways of structuring data aren't quite adequate for large web apps. Redux has been extremely popular because it's both powerful and easy to understand.

Data architecture can be a complex topic and so Redux's best feature is probably its simplicity. If you strip Redux down to the essential core, Redux is fewer than 100 lines of code.

We can build rich, easy to understand, web apps by using Redux as the backbone of our application. But first, let's walk through how to write a minimal Redux and later we'll work out patterns that emerge as we work out these ideas in a larger app.

There are several attempts to use Redux or create a Redux-inspired system that works with Angular. Two notable examples are:

- ngrx/store[99] and
- angular2-redux[100]

`ngrx` is a Redux-inspired architecture that is heavily observables-based. `angular2-redux` uses Redux itself as a dependency, and adds some Angular helpers (dependency-injection, observable wrappers).

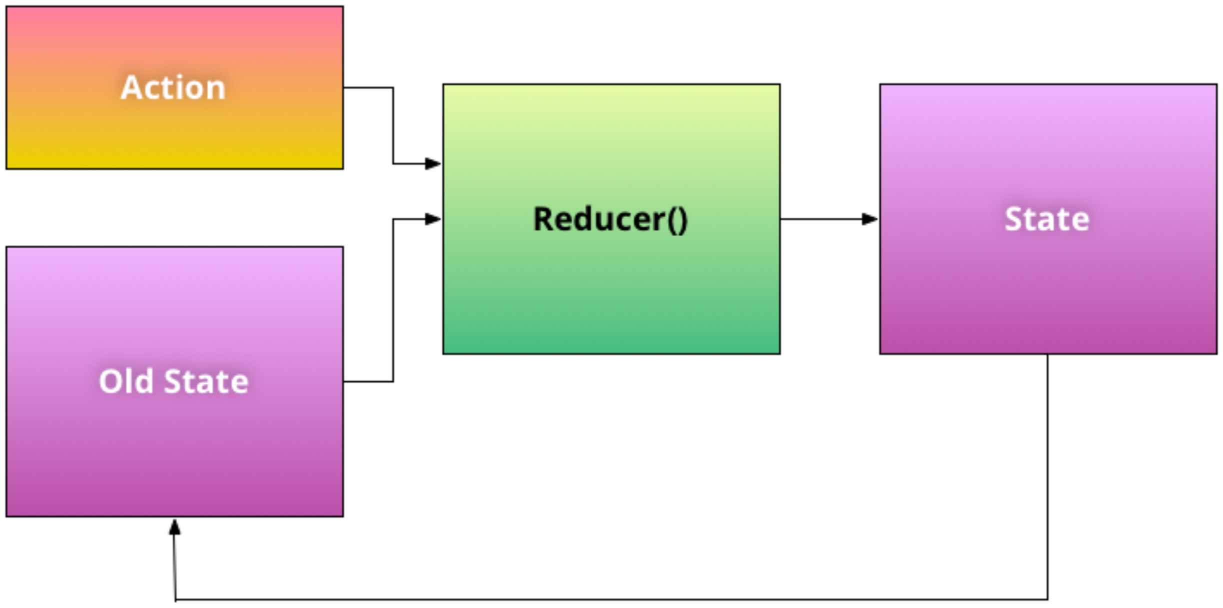
Here we're not going to use either. Instead, we're going to use Redux directly in order to show the concepts without introducing a new dependency. That said, both of these libraries may be helpful to you when writing your apps.

Redux: Key Ideas

The key ideas of Redux are this:

- All of your application's data is in a single data structure called the *state* which is held in the *store*
- Your app reads the **state** from this **store**
- This **store** is never mutated directly
- User interaction (and other code) fires *actions* which describe what happened
- A *new state* is created by combining he **old state** and the **action** by a function called the *reducer*.

[98]http://redux.js.org/
[99]https://github.com/ngrx/store
[100]https://github.com/InfomediaLtd/angular2-redux



Redux Core

If the above bullet list isn't clear yet, don't worry about it - putting these ideas into practice is the goal of the rest of this chapter.

Core Redux Ideas

What's a *reducer*?

Let's talk about the *reducer* first. Here's the idea of a *reducer*: it takes the *old state* and an *action* and returns a *new state*.

A reducer must be a **pure function**[101]. That is:

1. It must not mutate the current state directly
2. It must not use any data outside of its arguments

Put another way, a pure function will always **return the same value, given the same set of arguments**. And a pure function won't call any functions which have an effect on the outside world, e.g. no database calls, no HTTP calls, and no mutating outside data structures.

Reducers should always treat the current state as **read-only**. A reducer **does not change the state** instead, it **returns a new state**. (Often this new state will start with a copy of old state, but let's not get ahead of ourselves.)

Let's define our very first reducer. Remember, there are three things involved:

[101]https://en.wikipedia.org/wiki/Pure_function

1. An `Action`, which defines what to do (with optional arguments)
2. The `state`, which stores *all* of the data in our application
3. The `Reducer` which takes the `state` and the `Action` and returns a new state.

Defining `Action` and `Reducer` Interfaces

Since we're using TypeScript we want to make sure this whole process is typed, so let's setup an interface for our `Action` and our `Reducer`:

The `Action` Interface

Our `Action` interface looks like this:

code/redux/redux-chat/tutorial/01-identity-reducer.ts

```
interface Action {
  type: string;
  payload?: any;
}
```

Notice that our `Action` has two fields:

1. `type` and
2. `payload`

The `type` will be an identifying string that describes the action like `INCREMENT` or `ADD_USER`. The `payload` can be an object of any kind. The `?` on `payload?` means that this field is optional.

The `Reducer` Interface

Our `Reducer` interface looks like this:

code/redux/redux-chat/tutorial/01-identity-reducer.ts

```
interface Reducer<T> {
  (state: T, action: Action): T;
}
```

Our `Reducer` is using a feature of TypeScript called *generics*. In this case type `T` is the type of the `state`. Notice that we're saying that a valid `Reducer` has a function which takes a `state` (of type `T`) and an `action` and returns a new `state` (also of type `T`).

Creating Our First `Reducer`

The simplest possible reducer returns the state itself. (You might call this the *identity* reducer because it applies the identity function[102] on the state. This is the default case for all reducers, as we will soon see).

code/redux/redux-chat/tutorial/01-identity-reducer.ts

```
let reducer: Reducer<number> = (state: number, action: Action) => {
  return state;
};
```

Notice that this `Reducer` makes the generic type concrete to `number` by the syntax `Reducer<number>`. We'll define more sophisticated states beyond a single number soon.

We're not using the `Action` yet, but let's try this `Reducer` just the same.

Running the examples in this section

You can find the code for this chapter in the folder `code/redux`. If the example is runnable you will see the filename the code is from above each code box.

In this first section, these examples are run **outside of the browser and run by node.js**. Because we're using TypeScript in these examples, you should run them using the commandline tool `ts-node`, (instead of `node` directly).

You can install `ts-node` by running:

```
npm install -g ts-node
```

Or by doing an `npm install` in the `code/redux/redux-chat` directory and then calling `./node_modules/.bin/ts-node [filename]`

For instance, to run the example above you might type (not including the `$`):

```
$ cd code/redux/redux-chat/tutorial
$ npm install
$ ./node_modules/.bin/ts-node 01-identity-reducer.ts
```

Use this same procedure for the rest of the code in this chapter until we instruct you to switch to your browser.

Running Our First `Reducer`

Let's put it all together and run this reducer:

[102] https://en.wikipedia.org/wiki/Identity_function

code/redux/redux-chat/tutorial/01-identity-reducer.ts

```
interface Action {
  type: string;
  payload?: any;
}

interface Reducer<T> {
  (state: T, action: Action): T;
}

let reducer: Reducer<number> = (state: number, action: Action) => {
  return state;
};

console.log( reducer(0, null) ); // -> 0
```

And run it:

```
$ cd code/redux/redux-chat/tutorial
$ ./node_modules/.bin/ts-node 01-identity-reducer.ts
0
```

It seems almost silly to have that as a code example, but it teaches us our first principle of reducers:

By default, reducers return the original state.

In this case, we passed a state of the number `0` and a `null` action. The result from this reducer is the state `0`.

But let's do something more interesting and make our state change.

Adjusting the Counter With *actions*

Eventually our state is going to be **much more** sophisticated than a single number. We're going to be holding the **all** of the data for our app in the `state`, so we'll need better data structure for the state eventually.

That said, using a single number for the state lets us focus on other issues for now. So let's continue with the idea that our `state` is simply a single number that is storing a counter.

Let's say we want to be able to change the `state` number. Remember that in Redux we do not modify the state. Instead, we create *actions* which instruct the *reducer* on how to generate a *new state*.

Let's create an `Action` to change our counter. Remember that the only required property is a `type`. We might define our first action like this:

```
let incrementAction: Action = { type: 'INCREMENT' }
```

We should also create a second action that instructs our reducer to make the counter smaller with:

```
let decrementAction: Action = { type: 'DECREMENT' }
```

Now that we have these actions, let's try using them in our reducer:

code/redux/redux-chat/tutorial/02-adjusting-reducer.ts

```
let reducer: Reducer<number> = (state: number, action: Action) => {
  if (action.type === 'INCREMENT') {
    return state + 1;
  }
  if (action.type === 'DECREMENT') {
    return state - 1;
  }
  return state;
};
```

And now we can try out the whole reducer:

code/redux/redux-chat/tutorial/02-adjusting-reducer.ts

```
let incrementAction: Action = { type: 'INCREMENT' };

console.log( reducer(0, incrementAction )); // -> 1
console.log( reducer(1, incrementAction )); // -> 2

let decrementAction: Action = { type: 'DECREMENT' };

console.log( reducer(100, decrementAction )); // -> 99
```

Neat! Now the new value of the state is returned according to which action we pass into the reducer.

Reducer `switch`

Instead of having so many `if` statements, the common practice is to convert the reducer body to a `switch` statement:

code/redux/redux-chat/tutorial/03-adjusting-reducer-switch.ts

```
let reducer: Reducer<number> = (state: number, action: Action) => {
  switch (action.type) {
  case 'INCREMENT':
    return state + 1;
  case 'DECREMENT':
    return state - 1;
  default:
    return state; // <-- dont forget!
  }
};

let incrementAction: Action = { type: 'INCREMENT' };
console.log(reducer(0, incrementAction)); // -> 1
console.log(reducer(1, incrementAction)); // -> 2

let decrementAction: Action = { type: 'DECREMENT' };
console.log(reducer(100, decrementAction)); // -> 99

// any other action just returns the input state
let unknownAction: Action = { type: 'UNKNOWN' };
console.log(reducer(100, unknownAction)); // -> 100
```

Notice that the `default` case of the `switch` returns the original `state`. This ensures that if an unknown action is passed in, there's no error and we get the original `state` unchanged.

Q: **Wait, all of my application state is in one giant `switch` statement?**

A: Yes and no.

If this is your first exposure to Redux reducers it might feel a little weird to have all of your application state changes be the result of a giant `switch`. There are two things you should know:

1. Having your state changes centralized in one place can help a **ton** in maintaining your program, particularly because it's easy to track down where the changes are happening when they're all together. (Furthermore, you can easily locate what state changes as the result of any action because you can search your code for the token specified for that action's `type`)
2. You can (and often do) break your reducers down into several sub-reducers which each manage a different branch of the state tree. We'll talk about this later.

Action "Arguments"

In the last example our actions contained only a `type` which told our reducer either to increment or decrement the state.

But often changes in our app can't be described by a single value - instead we need parameters to describe the change. This is why we have the `payload` field in our `Action`.

In this counter example, say we wanted to add `9` to the counter. One way to do this would be to send `9` `INCREMENT` actions, but that wouldn't be very efficient, especially if we wanted to add, say, `9000`.

Instead, let's add a `PLUS` action that will use the `payload` parameter to send a number which specifies how much we want to add to the counter. Defining this action is easy enough:

```
let plusSevenAction = { type: 'PLUS', payload: 7 };
```

Next, to support this action, we add a new `case` to our reducer that will handle a `'PLUS'` action:

code/redux/redux-chat/tutorial/04-plus-action.ts

```
let reducer: Reducer<number> = (state: number, action: Action) => {
  switch (action.type) {
  case 'INCREMENT':
    return state + 1;
  case 'DECREMENT':
    return state - 1;
  case 'PLUS':
    return state + action.payload;
  default:
    return state;
  }
};
```

`PLUS` will add whatever number is in the `action.payload` to the `state`. We can try it out:

code/redux/redux-chat/tutorial/04-plus-action.ts

```
console.log( reducer(3, { type: 'PLUS', payload: 7}) );    // -> 10
console.log( reducer(3, { type: 'PLUS', payload: 9000}) ); // -> 9003
console.log( reducer(3, { type: 'PLUS', payload: -2}) );   // -> 1
```

In the first line we take the state `3` and `PLUS` a payload of `7`, which results in `10`. Neat! However, notice that while we're passing in a `state`, it doesn't really ever *change*. That is, we're not storing the result of our reducer's changes and reusing it for future actions.

Storing Our State

Our reducers are pure functions, and do not change the world around them. The problem is, in our app, things *do* change. Specifically, our state changes and we need to keep the new state somewhere.

In Redux, we keep our state in the *store*. The store has the responsibility of **running the reducer and then keeping the new state**. Let's take a look at a minimal store:

code/redux/redux-chat/tutorial/05-minimal-store.ts

```
class Store<T> {
  private _state: T;

  constructor(
    private reducer: Reducer<T>,
    initialState: T
  ) {
    this._state = initialState;
  }

  getState(): T {
    return this._state;
  }

  dispatch(action: Action): void {
    this._state = this.reducer(this._state, action);
  }
}
```

Notice that our `Store` is generically typed - we specify the type of the *state* with generic type `T`. We store the state in the private variable `_state`.

We also give our `Store` a `Reducer`, which is also typed to operate on `T`, the state type this is because **each store is tied to a specific reducer**. We store the `Reducer` in the private variable `reducer`.

In Redux, we generally have 1 store and 1 top-level reducer per application.

Let's take a closer look at each method of our `State`:

- In our `constructor` we set the `_state` to the initial state.
- `getState()` simply returns the current `_state`

- `dispatch` takes an action, sends it to the reducer and then **updates the value of `_state`** with the return value

Notice that **`dispatch` doesn't return anything**. It's only *updating* the store's state (once the result returns). This is an important principle of Redux: dispatching actions is a "fire-and-forget" maneuver. **Dispatching actions is not a direct manipulation of the state, and it doesn't return the new state**.

When we dispatch actions, we're sending off a notification of what happened. If we want to know what the current state of the system is, we have to check the state of the store.

Using the Store

Let's try using our store:

code/redux/redux-chat/tutorial/05-minimal-store.ts

```
// create a new store
let store = new Store<number>(reducer, 0);
console.log(store.getState()); // -> 0

store.dispatch({ type: 'INCREMENT' });
console.log(store.getState()); // -> 1

store.dispatch({ type: 'INCREMENT' });
console.log(store.getState()); // -> 2

store.dispatch({ type: 'DECREMENT' });
console.log(store.getState()); // -> 1
```

We start by creating a new `Store` and we save this in `store`, which we can use to get the current state and dispatch actions.

The state is set to `0` initially, and then we `INCREMENT` twice and `DECREMENT` once and our final state is `1`.

Being Notified with `subscribe`

It's great that our `Store` keeps track of what changed, but in the above example we have to *ask* for the state changes with `store.getState()`. It would be nice for us to know immediately when a new action was dispatched so that we could respond. To do this we can implement the Observer pattern - that is, we'll register a callback function that will *subscribe* to all changes.

Here's how we want it to work:

1. We will register a *listener* function using `subscribe`
2. When `dispatch` is called, we will iterate over all listeners and call them, which is the notification that the state has changed.

Registering Listeners

Our listener callbacks are a going to be a function that takes *no arguments*. Let's define an interface that makes it easy to describe this:

code/redux/redux-chat/tutorial/06-store-w-subscribe.ts

```
interface ListenerCallback {
  (): void;
}
```

After we subscribe a listener, we might want to unsubscribe as well, so lets define the interface for an *unsubscribe* function as well:

code/redux/redux-chat/tutorial/06-store-w-subscribe.ts

```
interface UnsubscribeCallback {
  (): void;
}
```

Not much going on here - it's another function that takes no arguments and has no return value. But by defining these types it makes our code clearer to read.

Our store is going to keep a list of `ListenerCallbacks` let's add that to our `Store`:

code/redux/redux-chat/tutorial/06-store-w-subscribe.ts

```
class Store<T> {
  private _state: T;
  private _listeners: ListenerCallback[] = [];
```

Now we want to be able to add to that list of `_listeners` with a `subscribe` function:

code/redux/redux-chat/tutorial/06-store-w-subscribe.ts

```
  subscribe(listener: ListenerCallback): UnsubscribeCallback {
    this._listeners.push(listener);
    return () => { // returns an "unsubscribe" function
      this._listeners = this._listeners.filter(l => l !== listener);
    };
  }
```

`subscribe` accepts a `ListenerCallback` (i.e. a function with no arguments and no return value) and returns an `UnsubscribeCallback` (the same signature). Adding the new listener is easy: we `push` it on to the `_listeners` array.

The return value is a function which will update the list of `_listeners` to be the list of `_listeners` without the `listener` we just added. That is, it returns the `UnsubscribeCallback` that we can use to remove this listener from the list.

Notifying Our Listeners

Whenever our state changes, we want to call these listener functions. What this means is, whenever we `dispatch` a new action, whenever the state changes, we want to call all of the listeners:

code/redux/redux-chat/tutorial/06-store-w-subscribe.ts

```
  dispatch(action: Action): void {
    this._state = this.reducer(this._state, action);
    this._listeners.forEach((listener: ListenerCallback) => listener());
  }
```

The Complete Store

We'll try this out below, but before we do that, here's the complete code listing for our new `Store`:

code/redux/redux-chat/tutorial/06-store-w-subscribe.ts

```
class Store<T> {
  private _state: T;
  private _listeners: ListenerCallback[] = [];

  constructor(
    private reducer: Reducer<T>,
    initialState: T
  ) {
    this._state = initialState;
  }

  getState(): T {
    return this._state;
  }

  dispatch(action: Action): void {
    this._state = this.reducer(this._state, action);
    this._listeners.forEach((listener: ListenerCallback) => listener());
  }

  subscribe(listener: ListenerCallback): UnsubscribeCallback {
    this._listeners.push(listener);
    return () => { // returns an "unsubscribe" function
      this._listeners = this._listeners.filter(l => l !== listener);
    };
  }
}
```

Trying Out `subscribe`

Now that we can `subscribe` to changes in our store, let's try it out:

code/redux/redux-chat/tutorial/06-store-w-subscribe.ts

```
let store = new Store<number>(reducer, 0);
console.log(store.getState()); // -> 0

// subscribe
let unsubscribe = store.subscribe(() => {
  console.log('subscribed: ', store.getState());
});

store.dispatch({ type: 'INCREMENT' }); // -> subscribed: 1
store.dispatch({ type: 'INCREMENT' }); // -> subscribed: 2

unsubscribe();
store.dispatch({ type: 'DECREMENT' }); // (nothing logged)

// decrement happened, even though we weren't listening for it
console.log(store.getState()); // -> 1
```

Above we subscribe to our store and in the callback function we'll log `subscribed: ` and then the current store state.

Notice that the listener function is **not** given the current state as an argument. This might seem like an odd choice, but because there are some nuances to deal with, it's easier to think of *the notification of state changed* as separate from *the current state*. Without digging too much into the weeds, you can read more about this choice here[103], here[104], and here[105].

We store the `unsubscribe` callback and then notice that after we call `unsubscribe()` our log message isn't called. We can still dispatch actions, we just won't see the results until we ask the store for them.

If you're the type of person who likes RxJS and Observables, you might notice that implementing our own subscription listeners could also be implemented using RxJS. You could rewrite our `Store` to use Observables instead of our own subscriptions.

In fact, we've already done this for you and you can find the sample code in the file `code/redux/redux-chat/tutorial/06b-rx-store.ts`.

Using RxJS for the `Store` is an interesting and powerful pattern if you're willing to us RxJS for the backbone of our application data.

Here we're not going to use Observables very heavily, particularly because we want to discuss Redux itself and how to think about data architecture with a single state tree. Redux itself is powerful enough to use in our applications without Observables.

Once you get the concepts of using "straight" Redux, adding in Observables isn't difficult (if you already understand RxJS, that is). For now, we're going to use "straight" Redux and we'll give you some guidance on some Observable-based Redux-wrappers at the end.

The Core of Redux

The above store is the essential core of Redux. Our reducer takes the current state and action and returns a new state, which is held by the store.

There are obviously many more things that we need to add to build a large, production web app. However, all of the new ideas that we'll cover are patterns that flow from building on this simple idea of an immutable, central store of state. If you understand the ideas presented above, you would be likely to invent many of the patterns (and libraries) you find in more advanced Redux apps.

There's still a lot for us to cover about day-to-day use of redux though. For instance, we need to know:

- How to carefully handle more complex data structures in our state
- How to be notified when our state changes without having to poll the state (with subscriptions)
- How to intercept our dispatch for debugging (a.k.a. middleware)

[103] https://github.com/reactjs/redux/issues/1707
[104] https://github.com/reactjs/redux/issues/1513
[105] https://github.com/reactjs/redux/issues/303

- How to compute derived values (with *selectors*)
- How to split up large reducers into more manageable, smaller ones (and recombine them)
- How to deal with asynchronous data

We'll explain on each of these issues and describe common patterns over the rest of this chapter and the next.

Let's first deal with handling more complex data structures in our state. To do that, we're going to need an example that's more interesting than a counter. Let's start building a chat app where users can send each other messages.

A Messaging App

In our messaging app, as in all Redux apps, there are three main parts to the data model:

1. The state
2. The actions
3. The reducer

Messaging App `state`

The `state` in our counter app was a single number. However in our messaging app, the `state` is going to be **an object**.

This state object will have a single property, `messages`. `messages` will be an array of strings, with each string representing an individual message in the application. For example:

```
// an example `state` value
{
  messages: [
    'here is message one',
    'here is message two'
  ]
}
```

We can define the type for the app's state like this:

code/redux/redux-chat/tutorial/07-messages-reducer.ts

```
interface AppState {
  messages: string[];
}
```

Messaging App `actions`

Our app will process two actions: `ADD_MESSAGE` and `DELETE_MESSAGE`.

The `ADD_MESSAGE` action object will always have the property `message`, the message to be added to the state. The `ADD_MESSAGE` action object has this shape:

```
{
  type: 'ADD_MESSAGE',
  message: 'Whatever message we want here'
}
```

The `DELETE_MESSAGE` action object will delete a specified message from the state. A challenge here is that we have to be able to specify *which message* we want to delete.

If our messages were objects, we could assign each message an `id` property when it is created. However, to simplify this example, our messages are just simple strings, so we'll have to get a handle to the message another way. The easiest way for now is to just use the index of the message in the array (as a proxy for the ID).

With that in mind, the `DELETE_MESSAGE` action object has this shape:

```
{
  type: 'DELETE_MESSAGE',
  index: 2                // <- or whatever index is appropriate
}
```

We can define the types for these actions by using the `interface ... extends` syntax in TypeScript:

code/redux/redux-chat/tutorial/07-messages-reducer.ts

```
interface AddMessageAction extends Action {
  message: string;
}

interface DeleteMessageAction extends Action {
  index: number;
}
```

In this way our `AddMessageAction` is able to specify a `message` and the `DeleteMessageAction` will specify an `index`.

Messaging App `reducer`

Remember that our reducer needs to handle two actions: `ADD_MESSAGE` and `DELETE_MESSAGE`. Let's talk about these individually.

Reducing `ADD_MESSAGE`

code/redux/redux-chat/tutorial/07-messages-reducer.ts

```
let reducer: Reducer<AppState> =
  (state: AppState, action: Action): AppState => {
  switch (action.type) {
  case 'ADD_MESSAGE':
    return {
      messages: state.messages.concat(
        (<AddMessageAction>action).message
      ),
    };
```

We start by switching on the `action.type` and handling the `ADD_MESSAGE` case.

TypeScript objects already have a type, so why are we adding a `type` field?

There are many different ways we might choose to handle this sort of "polymorphic dispatch". Keeping a string in a `type` field (where `type` means "action-type") is a straightforward, portable way we can use to distinguish different types of actions and handle them in one reducer. In part, it means that you don't *have* to create a new `interface` for every action.

That said, it would be more satisfying to be able to use reflection to switch on the concrete type. While this might become possible with more advanced type guards[106], this isn't currently possible in today's TypeScript.

Broadly speaking, types are a compile-time construct and this code is compiled down to JavaScript and we can lose some of the typing metadata.

That said, if switching on a `type` field bothers you and you'd like to use language features directly, you could use the decoration reflection metadata[107]. For now, a simple `type` field will suffice.

Adding an Item Without Mutation

When we handle an `ADD_MESSAGE` action, we need to add the given message to the state. As will all reducer handlers, we need to **return a new state**. Remember that our reducers must be *pure* and not mutate the old state.

What would be the problem with the following code?

```
case 'ADD_MESSAGE':
  state.messages.push( action.message );
  return { messages: messages };
  // ...
```

The problem is that this code **mutates** the `state.messages` array, which changes our old state! Instead what we want to do is create a *copy* of the `state.messages` array and add our new message to the copy.

[106]https://basarat.gitbooks.io/typescript/content/docs/types/typeGuard.html
[107]http://blog.wolksoftware.com/decorators-metadata-reflection-in-typescript-from-novice-to-expert-part-4

code/redux/redux-chat/tutorial/07-messages-reducer.ts

```
  case 'ADD_MESSAGE':
    return {
      messages: state.messages.concat(
        (<AddMessageAction>action).message
      ),
    };
```

The syntax `<AddMessageAction>action` will cast our `action` to the more specific type. That is, notice that our reducer takes the more general type `Action`, which does not have the `message` field. If we leave off the cast, then the compiler will complain that `Action` does not have a field `message`.

Instead, we know that we have an `ADD_MESSAGE` action so we cast it to an `AddMessageAction`. We use parentheses to make sure the compiler knows that we want to cast `action` and not `action.message`.

Remember that the reducer **must return a new `AppState`**. When we return an object from our reducer it must match the format of the `AppState` that was input. In this case we only have to keep the key `messages`, but in more complicated states we have more fields to worry about.

Deleting an Item Without Mutation

Remember that when we handle the `DELETE_MESSAGE` action we are passing the index of the item in the array as the faux ID. (Another common way of handling the same idea would be to pass a real item ID.) Again, because we do not want to mutate the old `messages` array, we need to handle this case with care:

code/redux/redux-chat/tutorial/07-messages-reducer.ts

```
  case 'DELETE_MESSAGE':
    let idx = (<DeleteMessageAction>action).index;
    return {
      messages: [
        ...state.messages.slice(0, idx),
        ...state.messages.slice(idx + 1, state.messages.length)
      ]
```

Here we use the `slice` operator twice. First we take all of the items up until the item we are removing. And we concatenate the items that come after.

There are four common non-mutating operations:

- Adding an item to an array
- Removing an item from an array
- Adding / changing a key in an object
- Removing a key from an object

The first two (array) operations we just covered. We'll talk more about the object operations further down, but for now know that a common way to do this is to use `Object.assign`. As in:

```
Object.assign({}, oldObject, newObject)
             // <-------<-------------
```

You can think of `Object.assign` as merging objects in from the right into the object on the left. `newObject` is merged into `oldObject` which is merged into `{}`. This way all of the fields in `oldObject` will be kept, except for where the field exists in `newObject`. Neither `oldObject` nor `newObject` will be mutated.

Of course, handling all of this on your own takes great care and it is easy to make a mistake. This is one of the reasons many people use Immutable.js[108], which is a set of data structures that help enforce immutability.

Trying Out Our Actions

Now let's try running our actions:

code/redux/redux-chat/tutorial/07-messages-reducer.ts

```
let store = new Store<AppState>(reducer, { messages: [] });
console.log(store.getState()); // -> { messages: [] }

store.dispatch({
  type: 'ADD_MESSAGE',
  message: 'Would you say the fringe was made of silk?'
} as AddMessageAction);

store.dispatch({
  type: 'ADD_MESSAGE',
  message: 'Wouldnt have no other kind but silk'
} as AddMessageAction);
```

[108] https://facebook.github.io/immutable-js/

```

store.dispatch({
  type: 'ADD_MESSAGE',
  message: 'Has it really got a team of snow white horses?'
} as AddMessageAction);

console.log(store.getState());
// ->
// { messages:
//    [ 'Would you say the fringe was made of silk?',
//      'Wouldnt have no other kind but silk',
//      'Has it really got a team of snow white horses?' ] }
```

Here we start with a new store and we call `store.getState()` and see that we have an empty `messages` array.

Next we add three messages[109] to our store. For each message we specify the `type` as `ADD_MESSAGE` and we cast each object to an `AddMessageAction`.

Finally we log the new state and we can see that `messages` contains all three messages.

Our three `dispatch` statements are a bit ugly for two reasons:

1. we manually have to specify the `type` string each time. We could use a constant, but it would be nice if we didn't have to do this and
2. we're manually casting to an `AddMessageAction`

Instead of creating these objects as an object directly we should create a *function* that will create these objects. This idea of writing a function to create actions is so common in Redux that the pattern has a name: *Action Creators*.

Action Creators

Instead of creating the `ADD_MESSAGE` actions directly as objects, let's create a function to do this for us:

[109] https://en.wikipedia.org/wiki/The_Surrey_with_the_Fringe_on_Top

code/redux/redux-chat/tutorial/08-action-creators.ts

```
class MessageActions {
  static addMessage(message: string): AddMessageAction {
    return {
      type: 'ADD_MESSAGE',
      message: message
    };
  }
  static deleteMessage(index: number): DeleteMessageAction {
    return {
      type: 'DELETE_MESSAGE',
      index: index
    };
  }
}
```

Here we've created a class with two static methods `addMessage` and `deleteMessage`. They return an `AddMessageAction` and a `DeleteMessageAction` respectively.

You definitely don't *have* to use static methods for your action creators. You could use plain functions, functions in a namespace, even instance methods on an object, etc. The key idea is to keep them organized in a way that makes them easy to use.

Now let's use our new action creators:

code/redux/redux-chat/tutorial/08-action-creators.ts

```
let store = new Store<AppState>(reducer, { messages: [] });
console.log(store.getState()); // -> { messages: [] }

store.dispatch(
  MessageActions.addMessage('Would you say the fringe was made of silk?'));

store.dispatch(
  MessageActions.addMessage('Wouldnt have no other kind but silk'));

store.dispatch(
  MessageActions.addMessage('Has it really got a team of snow white horses?'));

console.log(store.getState());
// ->
// { messages:
//    [ 'Would you say the fringe was made of silk?',
//      'Wouldnt have no other kind but silk',
//      'Has it really got a team of snow white horses?' ] }
```

This feels much nicer!

An added benefit is that if we eventually decided to change the format of our messages, we could do it without having to update all of our `dispatch` statements. For instance, say we wanted to add the time each message was created. We could add a `created_at` field to `addMessage` and now all `AddMessageActions` will be given a `created_at` field:

```
class MessageActions {
  static addMessage(message: string): AddMessageAction {
    return {
      type: 'ADD_MESSAGE',
      message: message,
      // something like this
      created_at: new Date()
    };
  }
  // ....
```

Using Real Redux

Now that we've built our own mini-redux you might be asking, "What do I need to do to use the *real* Redux?" Thankfully, not very much. Let's update our code to use the real Redux now!

If you haven't already, you'll want to run `npm install` in the `code/redux/redux-chat/tutorial` directory.

The first thing we need to do is import `Action`, `Reducer`, and `Store` from the `redux` package. We're also going to import a helper method `createStore` while we're at it:

code/redux/redux-chat/tutorial/09-real-redux.ts

```
import {
  Action,
  Reducer,
  Store,
  createStore
} from 'redux';
```

Next, instead of specifying our initial state when we create the *store* instead we're going to let the *reducer* create the initial state. Here we'll do this as the default argument to the reducer. This way if there is no state passed in (e.g. the first time it is called at initialization) we will use the initial state:

code/redux/redux-chat/tutorial/09-real-redux.ts

```
let initialState: AppState = { messages: [] };

let reducer: Reducer<AppState> =
  (state: AppState = initialState, action: Action) => {
```

What's neat about this is that the rest of our reducer stays the same!

The last thing we need to do is create the store using the `createStore` helper method from Redux:

code/redux/redux-chat/tutorial/09-real-redux.ts

```
let store: Store<AppState> = createStore<AppState>(reducer);
```

After that, everything else just works!

code/redux/redux-chat/tutorial/09-real-redux.ts

```
let store: Store<AppState> = createStore<AppState>(reducer);
console.log(store.getState()); // -> { messages: [] }

store.dispatch(
  MessageActions.addMessage('Would you say the fringe was made of silk?'));

store.dispatch(
  MessageActions.addMessage('Wouldnt have no other kind but silk'));

store.dispatch(
  MessageActions.addMessage('Has it really got a team of snow white horses?'));

```

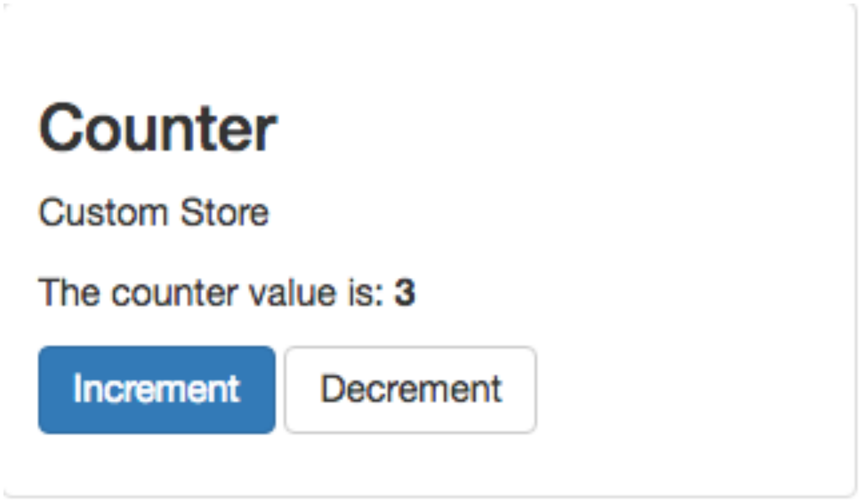
```
console.log(store.getState());
// ->
// { messages:
//    [ 'Would you say the fringe was made of silk?',
//      'Wouldnt have no other kind but silk',
//      'Has it really got a team of snow white horses?' ] }
```

Now that we have a handle on using Redux in isolation, the next step is to hook it up to our web app. Let's do that now.

Using Redux in Angular

In the last section we walked through the core of Redux and showed how to create reducers and use stores to manage our data in isolation. Now it's time to level-up and integrate Redux with our Angular components.

In this section we're going to create a minimal Angular app that contains just a counter which we can increment and decrement with a button.



Counter App

By using such a small app we can focus on the integration points between Redux and Angular and then we can move on to a larger app in the next section. But first, let's see how to build this counter app!

Here we are going to be integrating Redux directly with Angular without any helper libraries in-between. There are several open-source libraries with the goal of making this process easier, and you can find them in the references section below.

That said, it can be much easier to use those libraries once you understand what is going on underneath the hood, which is what we work through here.

Planning Our App

If you recall, the three steps to planning our Redux apps are to:

1. Define the structure of our central app state
2. Define actions that will change that state and
3. Define a reducer that takes the old state and an action and returns a new state.

For this app, we're just going to increment and decrement a counter. We did this in the last section, and so our actions, store, and reducer will all be very familiar.

The other thing we need to do when writing Angular apps is decide where we will create components. In this app, we'll have a top-level `AppComponent` which contains the view we see in the screenshot.

At a high level we're going to do the following:

1. Create our `Store` and make it accessible to our whole app via dependency injection
2. Subscribe to changes to the `Store` and display them in our components
3. When something changes (a button is pressed) we will dispatch an action to the `Store`.

Enough planning, let's look at how this works in practice!

Setting Up Redux

Defining the Application State

Let's take a look at our `AppState`:

code/redux/redux-chat/redux-counter/src/app/app.state.ts

```
export interface AppState {
  counter: number;
};
```

Here we are defining our core state structure as `AppState` - it is an object with one key, `counter` which is a `number`. In the next example (the chat app) we'll talk about how to have more sophisticated states, but for now this will be fine.

Defining the Reducers

Next lets define the reducer which will handle incrementing and decrementing the counter in the application state:

code/redux/redux-chat/redux-counter/src/app/counter.reducer.ts

```
import {
  INCREMENT,
  DECREMENT
} from './counter.actions';

const initialState: AppState = { counter: 0 };

// Create our reducer that will handle changes to the state
export const counterReducer: Reducer<AppState> =
  (state: AppState = initialState, action: Action): AppState => {
    switch (action.type) {
    case INCREMENT:
      return Object.assign({}, state, { counter: state.counter + 1 });
    case DECREMENT:
      return Object.assign({}, state, { counter: state.counter - 1 });
    default:
      return state;
    }
  };
```

We start by importing the constants `INCREMENT` and `DECREMENT`, which are exported by our action creators. They're just defined as the strings `'INCREMENT'` and `'DECREMENT'`, but it's nice to get the extra help from the compiler in case we make a typo. We'll look at those action creators in a minute.

The `initialState` is an `AppState` which sets the counter to `0`.

The `counterReducer` handles two actions: `INCREMENT`, which adds `1` to the current counter and `DECREMENT`, which subtracts `1`. Both actions use `Object.assign` to ensure that we don't *mutate* the old state, but instead create a new object that gets returned as the new state.

Since we're here, let's look at the action creators

Defining Action Creators

Our action creators are functions which return objects that define the action to be taken. `increment` and `decrement` below return an object that defines the appropriate `type`.

code/redux/redux-chat/redux-counter/src/app/counter.actions.ts

```
import {
  Action,
  ActionCreator
} from 'redux';

export const INCREMENT: string = 'INCREMENT';
export const increment: ActionCreator<Action> = () => ({
  type: INCREMENT
});

export const DECREMENT: string = 'DECREMENT';
export const decrement: ActionCreator<Action> = () => ({
  type: DECREMENT
});
```

Notice that our action creator functions return the type `ActionCreator<Action>`. `ActionCreator` is a generic class defined by Redux that we use to define functions that create actions. In this case we're using the concrete class `Action`, but we could use a more specific `Action` class, such as `AddMessageAction` that we defined in the last section.

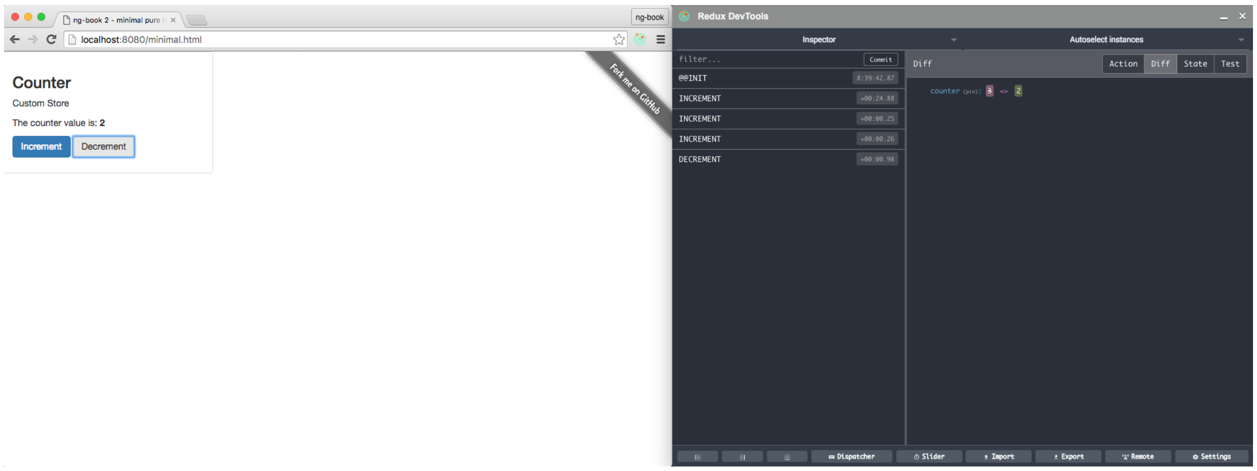
Creating the Store

Now that we have our reducer and state, we could create our store like so:

```
let store: Store<AppState> = createStore<AppState>(counterReducer);
```

However, one of the awesome things about Redux is that it has a robust set of developer tools. Specifically, there is a Chrome extension[110] that will let us monitor the state of our application and dispatch actions.

[110] https://chrome.google.com/webstore/detail/redux-devtools/lmhkpmbekcpmknklioeibfkpmmfibljd?hl=en



Counter App With Redux Devtools

What's really neat about the Redux Devtools is that it gives us clear insight to every action that flows through the system and it's affect on the state.

Go ahead and install the Redux Devtools Chrome extension[111] now!

In order to use the Devtools we have to do one thing: add it to our store.

code/redux/redux-chat/redux-counter/src/app/app.store.ts

```
const devtools: StoreEnhancer<AppState> =
  window['devToolsExtension'] ?
  window['devToolsExtension']() : f => f;
```

Not everyone who uses our app will necessarily have the Redux Devtools installed. The code above will check for `window.devToolsExtension`, which is defined by Redux Devtools, and if it exists, we will use it. If it doesn't exist, we're just returning an *identity function* (`f => f`) that will return whatever is passed to it.

Middleware is a term for a function that enhances the functionality of another library. The Redux Devtools is one of many possible middleware libraries for Redux. Redux supports lots of interesting middleware and it's easy to write our own.

You can read more about Redux middleware here[112]

In order to use this `devtools` we pass it as *middleware* to our Redux store:

[111]https://chrome.google.com/webstore/detail/redux-devtools/lmhkpmbekcpmknklioeibfkpmmfibljd?hl=en
[112]http://redux.js.org/docs/advanced/Middleware.html

code/redux/redux-chat/redux-counter/src/app/app.store.ts

```
export function createAppStore(): Store<AppState> {
  return createStore<AppState>(
    reducer,
    compose(devtools)
  );
}
```

Now whenever we dispatch an action and change our state, we can inspect it in our browser!

Providing the Store

Now that we have the Redux core setup, let's turn our attention to our Angular components. Let's create our top-level app component, `AppComponent`. This will be the component we use to `bootstrap` Angular:

We're going to use the `AppComponent` as the root component. Remember that since this is a Redux app, we need to make our store instance accessible everywhere in our app. How should we do this? We'll use dependency injection (DI).

If you recall from the dependency injection chapter, when we want to make something available via DI, then we use the `providers` configuration to add it to the list of `providers` in our `NgModule`.

When we provide something to the DI system, we specify two things:

1. the *token* to use to refer this injectable dependency
2. the *way* to inject the dependency

Oftentimes if we want to provide a singleton service we might use the `useClass` option as in:

```
{ provide: SpotifyService, useClass: SpotifyService }
```

In the case above, we're using the class `SpotifyService` as the *token* in the DI system. The `useClass` option tells Angular to *create an instance* of `SpotifyService` and reuse that instance whenever the `SpotifyService` injection is requested (e.g. maintain a Singleton).

One problem with us using this method is that we don't want Angular to create our store - we did it ourselves above with `createStore`. We just want to use the `store` we've already created.

To do this we'll use the `useValue` option of `provide`. We've done this before with configurable values like `API_URL`:

```
{ provide: API_URL, useValue: 'http://localhost/api' }
```

The one thing we have left to figure out is what token we want to use to inject. Our `store` is of type `Store<AppState>`:

code/redux/redux-chat/redux-counter/src/app/app.store.ts

```
export function createAppStore(): Store<AppState> {
  return createStore<AppState>(
    reducer,
    compose(devtools)
  );
}

export const appStoreProviders = [
   { provide: AppStore, useFactory: createAppStore }
];
```

`Store` is an *interface*, not a class and, unfortunately, we can't use interfaces as a dependency injection key.

> If you're interested in *why* we can't use an interface as a DI key, it's because TypeScript interfaces are removed after compilation and not available at runtime.
>
> If you'd like to read more, see here[113], here[114], and here[115].

This means we need to create our own token that we'll use for injecting the store. Thankfully, Angular makes this easy to do. Let's create this token in it's own file so that way we can `import` it from anywhere in our application;

code/redux/redux-chat/redux-counter/src/app/app.store.ts

```
export const AppStore = new InjectionToken('App.store');
```

Here we have created a `const AppStore` which uses the `InjectionToken` class from Angular. `InjectionToken` is a better choice than injecting a string directly because it helps us avoid collisions.

Now we can use this token `AppStore` with `provide`. Let's do that now.

Bootstrapping the App

Back in `app.module.ts`, let's create the `NgModule` we'll use to bootstrap our app:

[113] http://stackoverflow.com/questions/32254952/binding-a-class-to-an-interface
[114] https://github.com/angular/angular/issues/135
[115] http://victorsavkin.com/post/126514197956/dependency-injection-in-angular-1-and-angular-2

code/redux/redux-chat/redux-counter/src/app/app.module.ts

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { HttpModule } from '@angular/http';

import { appStoreProviders } from './app.store';

import { AppComponent } from './app.component';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    FormsModule,
    HttpModule
  ],
  providers: [ appStoreProviders ],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Now we are able to get a reference to our Redux store anywhere in our app by injecting `AppStore`. The place we need it most now is our `AppComponent`.

Notice that we exported the function `appStoreProviders` from `app.store.ts` and then used that function in `providers`. Why not use the `{ provide: ..., useFactory: ... }` syntax directly? The answer is related to AOT - if we want to ahead-of-time compile a provider that uses a function, we must first export is as a function from another module.

The `AppComponent`

With our setup out of the way, we can start creating our component that actually displays the counter to the user and provides buttons for the user to change the state.

`import`s

Let's start by looking at the imports:

code/redux/redux-chat/redux-counter/src/app/app.component.ts

```
import { Component, Inject } from '@angular/core';
import { Store } from 'redux';
import { AppStore } from './app.store';
import { AppState } from './app.state';
import * as CounterActions from './counter.actions';
```

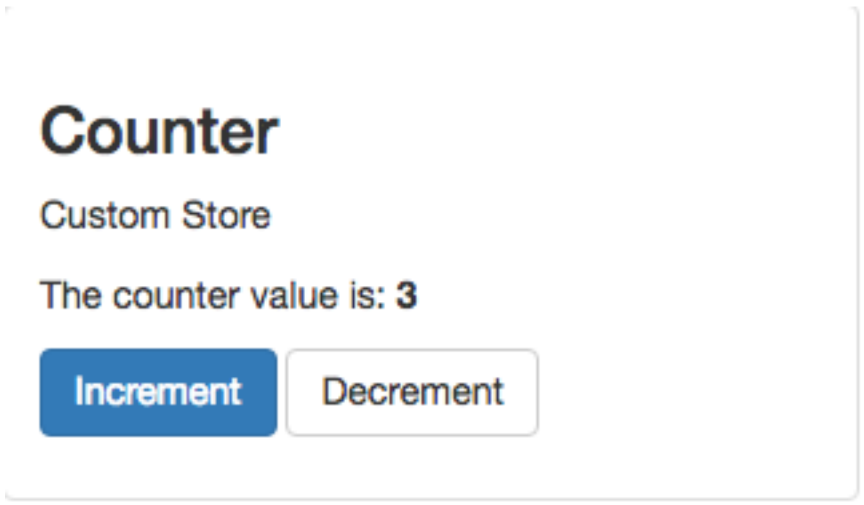
We import `Store` from Redux as well as our injector token `AppStore`, which will get us a reference to the singleton *instance* of our store. We also import the `AppState` type, which helps us know the structure of the central state.

Lastly, we import our action creators with `* as CounterActions`. This syntax will let us call `CounterActions.increment()` to create an `INCREMENT` action.

The template

Let's look at the template of our `AppComponent`.

In this chapter we are adding some style using the CSS framework Bootstrap[116]



Counter App Template

[116] http://getbootstrap.com

code/redux/redux-chat/redux-counter/src/app/app.component.html

```html
<div class="row">
  <div class="col-sm-6 col-md-4">
    <div class="thumbnail">
      <div class="caption">
        <h3>Counter</h3>
        <p>Custom Store</p>

        <p>
          The counter value is:
          <b>{{ counter }}</b>
        </p>

        <p>
          <button (click)="increment()"
                  class="btn btn-primary">
            Increment
          </button>
          <button (click)="decrement()"
                  class="btn btn-default">
             Decrement
          </button>
        </p>
      </div>
    </div>
  </div>
</div>
```

The three things to note here are that we're:

1. displaying the value of the counter in `{{ counter }}`
2. calling the `increment()` function in a button and
3. calling the `decrement()` function in a button.

The `constructor`

Remember that we need this component depends on the `Store`, so we need to inject it in the constructor. This is how we use our custom `AppStore` token to inject a dependency:

code/redux/redux-chat/redux-counter/src/app/app.component.ts

```
import { Component, Inject } from '@angular/core';
import { Store } from 'redux';
import { AppStore } from './app.store';
import { AppState } from './app.state';
import * as CounterActions from './counter.actions';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  counter: number;

  constructor(@Inject(AppStore) private store: Store<AppState>) {
    store.subscribe(() => this.readState());
    this.readState();
  }

  readState() {
    const state: AppState = this.store.getState() as AppState;
    this.counter = state.counter;
  }

  increment() {
    this.store.dispatch(CounterActions.increment());
  }

  decrement() {
    this.store.dispatch(CounterActions.decrement());
  }
}
```

We use the `@Inject` decorator to inject `AppStore` - notice that we define the type of the variable `store` to `Store<AppState>`. Having a different injection token than the type of the dependency injected is a little different than when we use the class as the injection token (and Angular infers what to inject).

We set the `store` to an instance variable (with `private store`). Now that we have the store we can listen for changes. Here we call `store.subscribe` and call `this.readState()`, which we define below.

The store will call `subscribe` only when a new action is dispatched, so in this case we need to make sure we manually call `readState` at least once to ensure that our component gets the initial data.

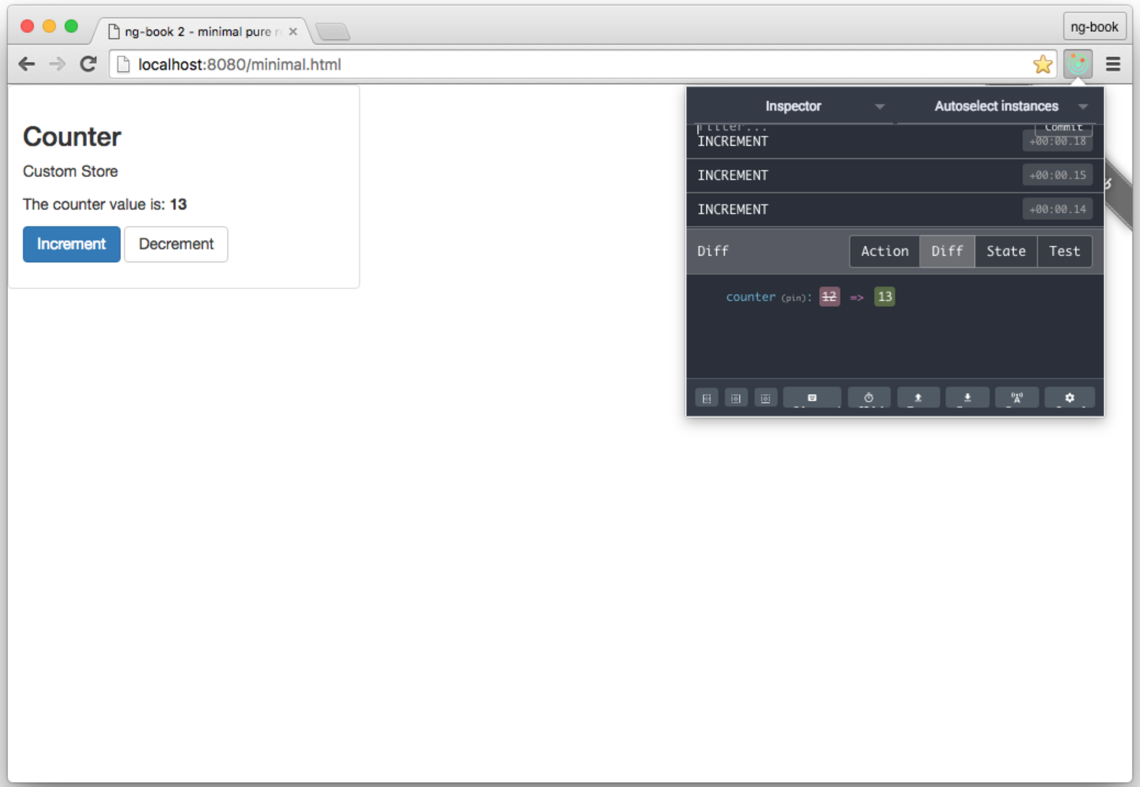
The method `readState` reads from our store and updates `this.counter` to the current value. Because `this.counter` is a property on this class and bound in the view, Angular will detect when it changes and re-render this component.

We define two helper methods: `increment` and `decrement`, each of which dispatch their respective actions to the store.

Putting It All Together

Try it out!

```
cd code/redux/redux-chat/redux-counter
npm install
npm start
open http://localhost:4200
```



Counter

Custom Store

The counter value is: **13**

Increment Decrement

```
INCREMENT  +00:00.18
INCREMENT  +00:00.15
INCREMENT  +00:00.14
counter (pin): 12 => 13
```

Working Counter App

Congratulations! You've created your first Angular and Redux app!

What's Next

Now that we've built a basic app using Redux and Angular, we should try building a more complicated app. When we build bigger apps we encounter new challenges like:

- How do we combine reducers?
- How do we extract data from different branches of the state?
- How should we organize our Redux code?

In the next chapter, we'll build a chat app which will tackle all of these questions!

References

If you want to learn more about Redux, here are some good resources:

- Official Redux Website[117]
- This Video Tutorial by Redux's Creator[118]
- Real World Redux[119] (presentation slides)
- The power of higher-order reducers[120]

To learn more about Redux and Angular checkout:

- angular2-redux[121]
- ng2-redux[122]
- ngrx/store[123]

Onward!

[117]http://redux.js.org/
[118]https://egghead.io/courses/getting-started-with-redux
[119]https://speakerdeck.com/chrisui/real-world-redux
[120]http://slides.com/omnidan/hor
[121]https://github.com/InfomediaLtd/angular2-redux
[122]https://github.com/angular-redux/ng2-redux
[123]https://github.com/ngrx/store