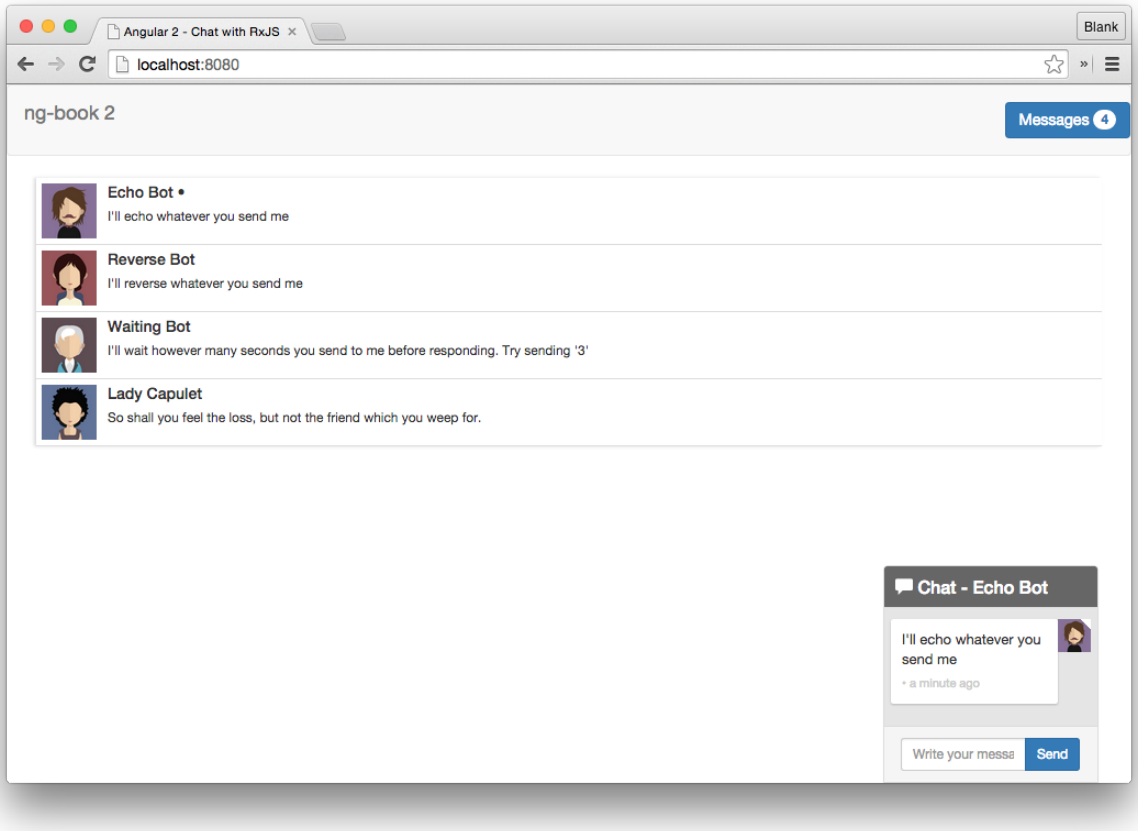
Intermediate Redux in Angular

In the last chapter we learned about Redux, the popular and elegant data architecture. In that chapter, we built an extremely basic app that tied our Angular components and the Redux store together.

In this chapter we're going to take on those ideas and build on them to create a more sophisticated chat app.

Here's a screenshot of the app we're going to build:



Completed Chat Application

Context For This Chapter

Earlier in this book we built a chat app using RxJS. We're going to be building that same app again only this time with Redux. The point is for you to be able to compare and contrast how the same app works with different data architecture strategies.

You are not required to have read the RxJS chapter in order to work through this one. This chapter stands on its own with regard to the RxJS chapters. If you have read that chapter, you'll be able to skim through some of the sections here where the code is largely the same (for instance, the data models themselves don't change much).

We *do* expect that you've read through the previous Redux chapter or at least have some familiarity with Redux.

Chat App Overview

In this application we've provided a few bots you can chat with. Open up the code and try it out:

```
cd code/redux/redux-chat
npm install
npm start
```

Now open your browser to `http://localhost:4200`.

Notice a few things about this application:

- You can click on the threads to chat with another person
- The bots will send you messages back, depending on their personality
- The unread message count in the top corner stays in sync with the number of unread messages

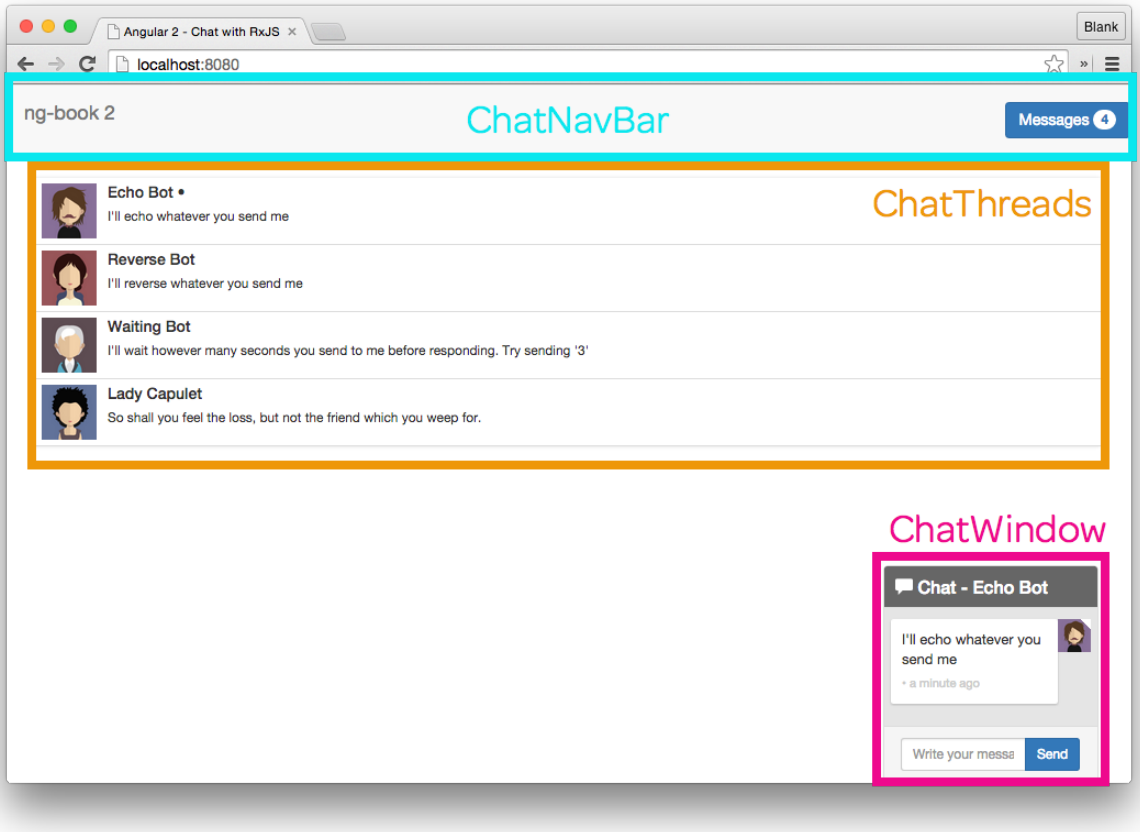
Let's look at an overview of how this app is constructed. We have

- 3 top-level Angular Components
- 3 models
- and 2 reducers, with their respective action creators

Let's look at them one at a time.

Components

The page is broken down into three top-level components:

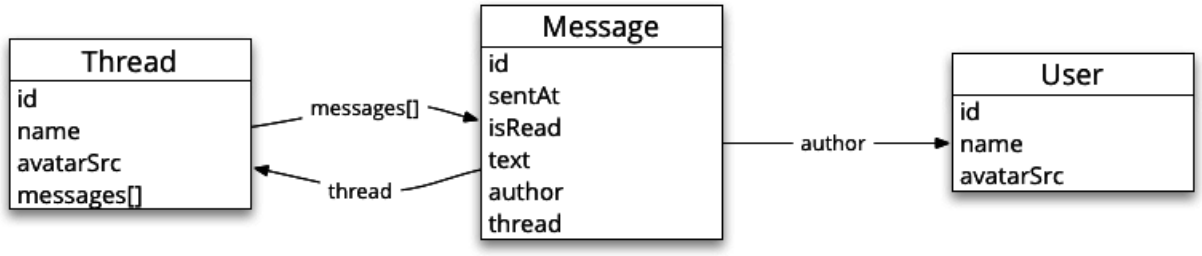


Redux Chat Top-Level Components

- `ChatNavBarComponent` - contains the unread messages count
- `ChatThreadsComponent` - shows a clickable list of threads, along with the most recent message and the conversation avatar
- `ChatWindowComponent` - shows the messages in the current thread with an input box to send new messages

Models

This application also has three models:



Redux Chat Models

- `User` - stores information about a chat participant
- `Message` - stores an individual message
- `Thread` - stores a collection of `Messages` as well as some data about the conversation

Reducers

In this app, we have two reducers:

- `UsersReducer` - handles information about the current user
- `ThreadsReducer` - handles threads and their messages

Summary

At a high level our data architecture looks like this:

- All information about the users and threads (which hold messages) are contained in our central store
- Components subscribe to changes in that store and display the appropriate data (unread count, list of threads, the messages themselves
- When the user sends a message, our components dispatch an action to the store

In the rest of this chapter, we're going to go in-depth on how we implement this using Angular and Redux. We'll start by implementing our models, then look at how we create our app state and reducers, and then finally we'll implement the Components.

Implementing the Models

Let's start with the easy stuff and take a look at the models.

We're going to be specifying each of our model definitions as `interfaces`. This isn't a requirement and you're free to use more elaborate objects if you wish. That said, objects with methods that mutate their internal state can break the functional model that we're striving for.

That is, all mutations to our app state should only be made by the reducers - the objects in the state should be immutable themselves.

So by defining an `interface` for our models,

1. we're able to ensure that the objects we're working with conform to an expected format at compile time and
2. we don't run the risk of someone accidentally adding a method to the model object that would work in an unexpected way.

`User`

Our `User` interface has an `id`, `name`, and `avatarSrc`.

code/redux/redux-chat/src/app/user/user.model.ts

```
/**
 * A User represents an agent that sends messages
 */
export interface User {
  id: string;
  name: string;
  avatarSrc: string;
  isClient?: boolean;
}
```

We also have a boolean `isClient` (the question mark indicates that this field is optional). We will set this value to `true` for the `User` that represents the client, the person using the app.

`Thread`

Similarly, `Thread` is also a TypeScript interface:

code/redux/redux-chat/src/app/thread/thread.model.ts

```
import { Message } from '../message/message.model';

/**
 * Thread represents a group of Users exchanging Messages
 */
export interface Thread {
  id: string;
  name: string;
  avatarSrc: string;
  messages: Message[];
}
```

We store the `id` of the `Thread`, the `name`, and the current `avatarSrc`. We also expect an array of `Messages` in the `messages` field.

Message

`Message` is our third and final model `interface`:

code/redux/redux-chat/src/app/message/message.model.ts

```
import { User } from '../user/user.model';
import { Thread } from '../thread/thread.model';

/**
 * Message represents one message being sent in a Thread
 */
export interface Message {
  id?: string;
  sentAt?: Date;
  isRead?: boolean;
  thread?: Thread;
  author: User;
  text: string;
}
```

Each message has:

- `id` - the id of the message

- `sentAt` - when the message was sent
- `isRead` - a boolean indicating that the message was read
- `author` - the `User` who wrote this message
- `text` - the text of the message
- `thread` - a reference to the containing `Thread`

App State

Now that we have our models, let's talk about the shape of our central state. In the previous chapter, our central state was a single object with the key `counter` which had the value of a `number`. This app, however, is more complicated.

Here's the first part of our app state:

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

```
18 export interface AppState {
19  users: UsersState;
20  threads: ThreadsState;
21 }
```

Our `AppState` is also an `interface` and it has two top level keys: `users` and `threads` - these are defined by two more interfaces `UsersState` and `ThreadsState`, which are defined in their respective reducers.

A Word on Code Layout

This is a common pattern we use in Redux apps: the top level state has a top-level key for each reducer. In our app we're going to keep this top-level reducer in `app.reducer.ts`.

Each reducer will have it's own file. In that file we'll store:

- The `interface` that describes that branch of the state tree
- The value of the initial state, for that branch of the state tree
- The reducer itself
- Any *selectors* that query that branch of the state tree - we haven't talked about *selectors* yet, but we will soon.

The reason we keep all of these different things together is because they all deal with the structure of this branch of the state tree. By putting these things in the same file it's very easy to refactor everything at the same time.

You're free to have multiple layers of nesting, if you so desire. It's a nice way to break up large modules in your app.

The Root Reducer

Since we're talking about how to split up reducers, let's look at our root reducer now:

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

```
18 export interface AppState {
19  users: UsersState;
20  threads: ThreadsState;
21 }
22
23 const rootReducer: Reducer<AppState> = combineReducers<AppState>({
24  users: UsersReducer,
25  threads: ThreadsReducer
26 });
27
28 export default rootReducer;
```

Notice the symmetry here - our `UsersReducer` will operate on the `users` key, which is of type `UsersState` and our `ThreadsReducer` will operate on the `threads` key, which is of type `ThreadsState`.

This is made possible by the `combineReducers` function which takes a map of keys and reducers and returns a new reducer that operates appropriately on those keys.

Of course we haven't finished looking at the structure of our `AppState` yet, so let's do that now.

The `UsersState`

Our `UsersState` holds a reference to the `currentUser`.

code/redux/redux-chat/src/app/user/users.reducer.ts

```
18 export interface UsersState {
19   currentUser: User;
20 };
21
22 const initialState: UsersState = {
23   currentUser: null
24 };
```

You could imagine that this branch of the state tree could hold information about all of the users, when they were last seen, their idle time, etc. But for now this will suffice.

We'll use `initialState` in our reducer when we define it below, but for now we're just going to set the current user to `null`.

The `ThreadsState`

Let's look at the `ThreadsState`:

code/redux/redux-chat/src/app/thread/threads.reducer.ts

```
25 export interface ThreadsEntities {
26   [id: string]: Thread;
27 }
28
29 export interface ThreadsState {
30   ids: string[];
31   entities: ThreadsEntities;
32   currentThreadId?: string;
33 };
34
35 const initialState: ThreadsState = {
36   ids: [],
37   currentThreadId: null,
38   entities: {}
39 };
```

We start by defining an interface called `ThreadsEntities` which is a map of thread `ids` to `Threads`. The idea is that we'll be able to look up any thread by id in this map.

In the `ThreadsState` we're also storing an array of the `ids`. This will store the list of possible ids that we might find in `entities`.

This strategy is used by the commonly-used library normalizr[124]. The idea is that when we standardize how we store entities in our Redux state, we're able to build helper libraries and it's clearer to work with. Instead of wondering what the format is for each tree of the state, when we use `normalizr` a lot of the choices have been made for us and we're able to work more quickly.

I've opted not to teach `normalizr` in this chapter because we're learning so many other things. That said, I would be very likely to use `normalizr` in my production applications.

That said, `normalizr` is totally optional - nothing major changes in our app by not using it.

If you'd like to learn how to use `normalizr`, checkout the official docs[125], this blog post[126], and the thread referenced by Redux creator Dan Abramov here[127]

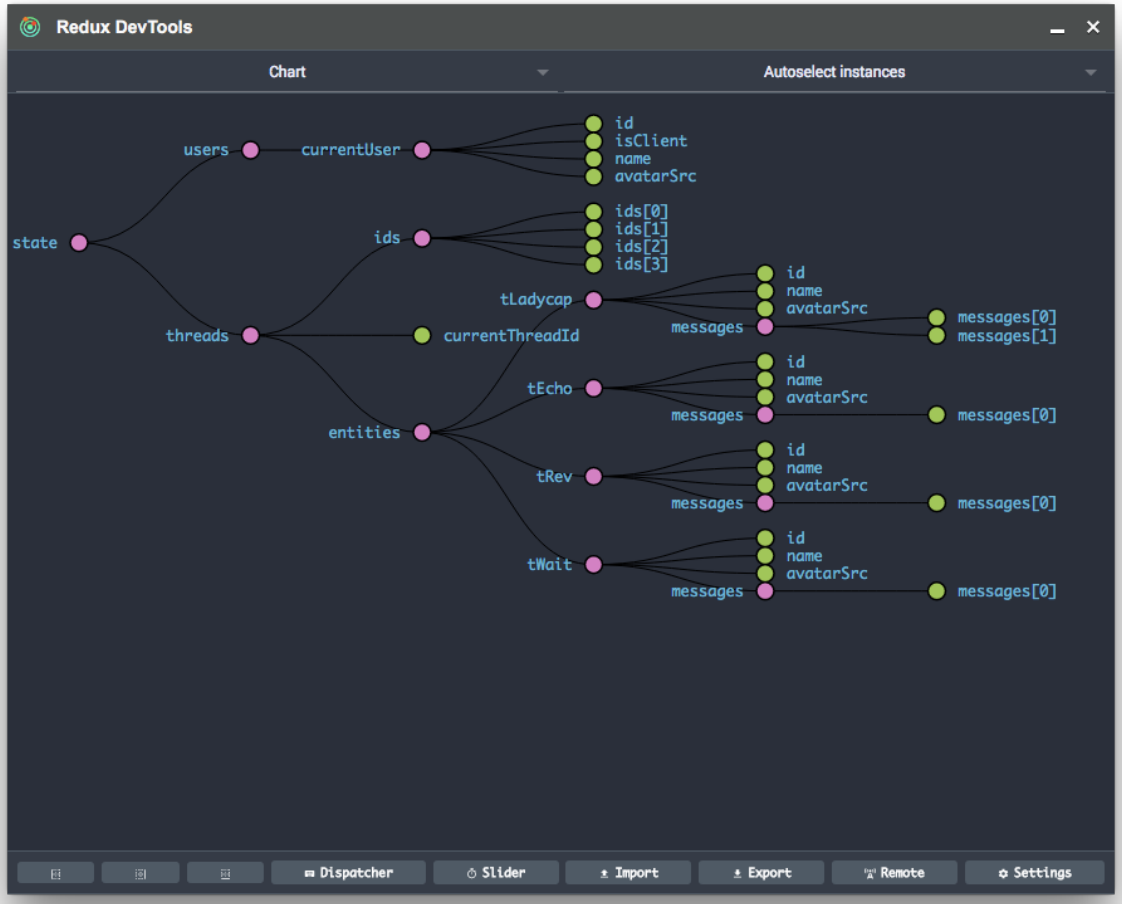
[124] https://github.com/paularmstrong/normalizr
[125] https://github.com/paularmstrong/normalizr
[126] https://medium.com/@mcowpercoles/using-normalizr-js-in-a-redux-store-96ab33991369#.l8ur7ipu6
[127] https://twitter.com/dan_abramov/status/663032263702106112

We store the currently viewed thread in `currentThreadId` - the idea here is that we want to know which thread the user is currently looking at.

We set our `initialState` to "empty" values.

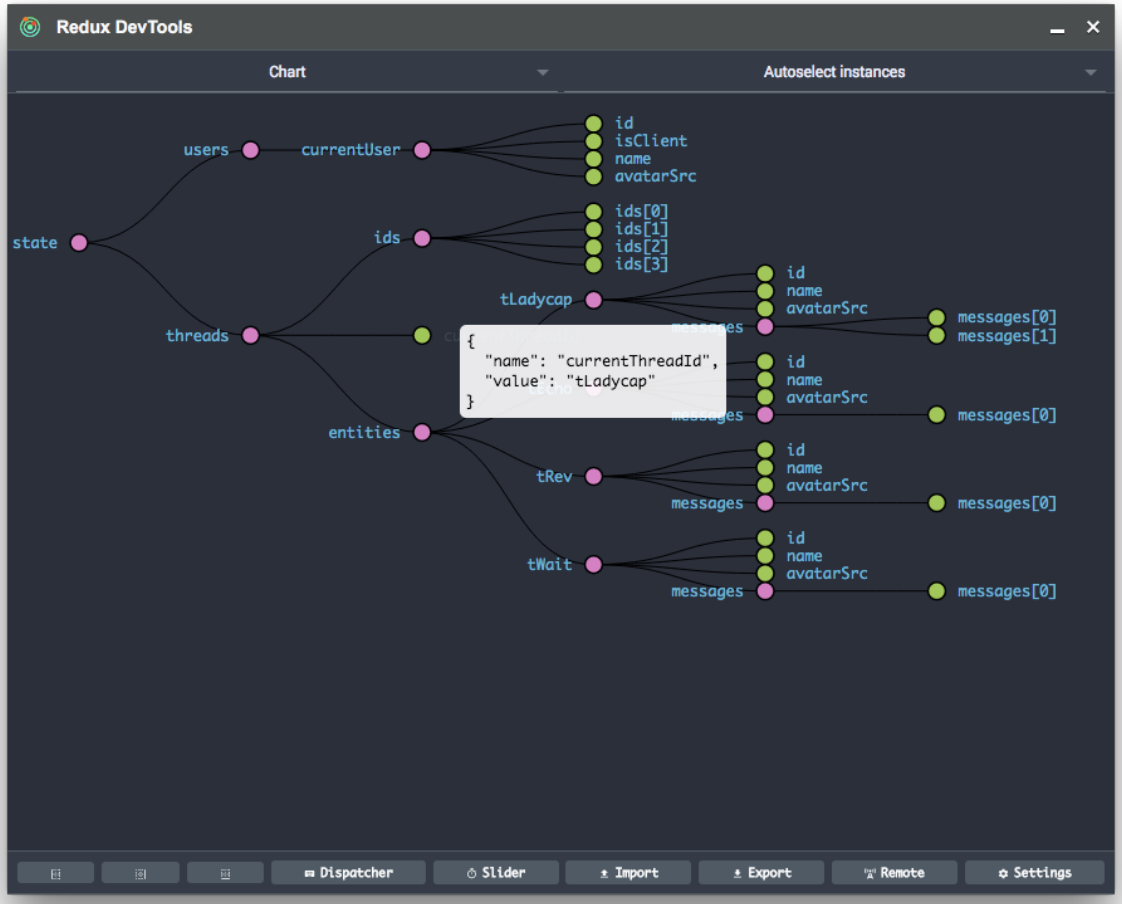
Visualizing Our `AppState`

Redux Devtools provides us with a "Chart" view that lets us inspect the state of our app. Here's what mine looks like after being booted with all of the demo data:



Redux Chat State Chart

What's neat is that we can hover over an individual node and see the attributes of that piece of data:



Inspecting the current thread

Building the Reducers (and Action Creators)

Now that we have our central state, we can start changing it using our reducers!

Since reducers handle actions, we need to know the format of our actions in our reducer. So let's build our action creators at the same time we build our reducers

Set Current User Action Creators

The `UsersState` stores the current user. This means we need an action to set the current user. We're going to keep our actions in the `actions` folder and name the actions to match their corresponding reducer, in this case `UserActions`.

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

```
20 export const SET_CURRENT_USER = '[User] Set Current';
21 export interface SetCurrentUserAction extends Action {
22   user: User;
23 }
24 export const setCurrentUser: ActionCreator<SetCurrentUserAction> =
25   (user) => ({
26     type: SET_CURRENT_USER,
27     user: user
28   });
```

Here we define the `const` `SET_CURRENT_USER`, which we'll use to `switch` on in our reducer.

We also define a new subinterface `SetCurrentUserAction` which extends `Action` to add a `user` property. We'll use the `user` property to indicate *which user* we want to make the current user.

The function `setCurrentUser` is our proper action creator function. It takes `user` as an argument, and returns a `SetCurrentUserAction` which we can give to our reducer.

`UsersReducer` - Set Current User

Now we turn our attention to our `UsersReducer`:

code/redux/redux-chat/src/app/user/users.reducer.ts

```
26 export const UsersReducer =
27   function(state: UsersState = initialState, action: Action): UsersState {
28   switch (action.type) {
29     case UserActions.SET_CURRENT_USER:
30     const user: User = (<UserActions.SetCurrentUserAction>action).user;
31       return {
32         currentUser: user
33       };
34     default:
35       return state;
36   }
37 };
```

Our `UsersReducer` takes a `UsersState` as the first argument. Notice that this isn't the `AppState`! Our "child reducer" only works with it's branch of the state tree.

Our `UsersReducer`, like all reducers, returns a new state, in this case it is of type `UsersState`.

Next we `switch` on the `action.type` and we handle the `UserActions.SET_CURRENT_USER`.

In order to set the current user, we need to get the `user` from the incoming action. To do this, we first cast the `action` to `UserActions.SetCurrentUserAction` and then we read the `.user` field.

It might seem a little weird that we originally created a `SetCurrentUserAction` but then now we switch on a `type` string instead of using the type directly.

Indeed, we are fighting TypeScript a little here. We lose interface metadata when the TypeScript is compiled to JavaScript. We could instead try some sort of reflection (through decorator metadata, or looking at a constructor etc.).

While down-casting our `SetCurrentUserAction` to an `Action` on `dispatch` and then re-casting is a bit ugly, it's a straightforward and portable way to handle this "polymorphic dispatch" for this app.

We need to return a new `UsersState`. Since `UsersState` only has one key, we return an object with the `currentUser` set to the incoming action's `user`.

Thread and Messages Overview

The core of our application is messages in threads. There are three actions we need to support:

1. Adding a new thread to the state
2. Adding messages to a thread
3. Selecting a thread

Let's start by creating a new thread

Adding a New Thread Action Creators

Here's the action creator for adding a new `Thread` to our state:

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

```
22 export const ADD_THREAD = '[Thread] Add';
23 export interface AddThreadAction extends Action {
24   thread: Thread;
25 }
26 export const addThread: ActionCreator<AddThreadAction> =
27   (thread) => ({
28     type: ADD_THREAD,
29     thread: thread
30   });
```

Notice that this is structurally very similar to our previous action creator. We define a `const ADD_THREAD` that we can switch on, a custom `Action`, and an action creator `addThread` which generates the `Action`.

Notice that we don't initialize the `Thread` itself here - the `Thread` is accepted as an argument.

Adding a New Thread Reducer

Now let's start our `ThreadsReducer` by handling `ADD_THREAD`:

code/redux/redux-chat/src/app/thread/threads.reducer.ts

```
45 export const ThreadsReducer =
46   function(state: ThreadsState = initialState, action: Action): ThreadsState {
47   switch (action.type) {
48
49     // Adds a new Thread to the list of entities
50     case ThreadActions.ADD_THREAD: {
51       const thread = (<ThreadActions.AddThreadAction>action).thread;
52
53       if (state.ids.includes(thread.id)) {
54         return state;
55       }
56
57       return {
58         ids: [ ...state.ids, thread.id ],
59         currentThreadId: state.currentThreadId,
60         entities: Object.assign({}, state.entities, {
61           [thread.id]: thread
62         })
63       };
64     }
65
66     // Adds a new Message to a particular Thread
```

Our `ThreadsReducer` handles the `ThreadsState`. When we handle the `ADD_THREAD` action, we cast the `action` object back into a `ThreadActions.AddThreadAction` and then pull the `Thread` out.

Next we check to see if this new `thread.id` already appears in the list of `state.ids`. If it does, then we don't make any changes, but instead return the current `state`.

However if this `thread` is new, then we need to add it to our current state.

Remember when we create a new `ThreadsState` we need to take care to now mutate our old state. This looks more complicated than any state we've done so far, but it's not very different in principle.

We start by adding our `thread.id` to the `ids` array. Here we're using the ES6 spread operator (. . .) to indicate that we want to put all of the existing `state.ids` into this new array and then append `thread.id` to the end.

`currentThreadId` does not change when we add a new thread, so we return the *old* `state.currentThreadId` for this field.

For `entities`, remember that it is an object where the key is the `string` id of each thread and the value is the thread itself. We're using `Object.assign` here to create a new object that merges the old `state.entities` with our newly added `thread` into a new object.

You might be kind of tired of meticulously copying these objects when we need to make changes. That's a common response! In fact, it's easy to make mutations here by accident.

This is why Immutable.js[128] was written. Immutable.js is often used with Redux for this purpose. When we use Immutable, these careful updates are handled for us.

I'd encourage you to take a look at Immutable.js and see if it is a good fit for your reducers.

Now we can add new threads to our central state!

Adding New Messages Action Creators

Now that we have threads we can start adding messages to them.

Let's define a new action for adding messages:

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

```
32 export const ADD_MESSAGE = '[Thread] Add Message';
33 export interface AddMessageAction extends Action {
34   thread: Thread;
35   message: Message;
36 }
```

The `AddMessageAction` adds a `Message` to a `Thread`.

Here's the action creator for adding a message:

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

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

```
37 export const addMessage: ActionCreator<AddMessageAction> =
38   (thread: Thread, messageArgs: Message): AddMessageAction => {
39     const defaults = {
40       id: uuid(),
41       sentAt: new Date(),
42       isRead: false,
43       thread: thread
44     };
45     const message: Message = Object.assign({}, defaults, messageArgs);
46
47     return {
48       type: ADD_MESSAGE,
49       thread: thread,
50       message: message
51     };
52   };
```

The `addMessage` action creator accepts a `thread` and an object we use for crafting the message. Notice here that we keep a list of `defaults`. The idea here is that we want to encapsulate creating an id, setting the timestamp, and setting the `isRead` status. Someone who wants to send a message shouldn't have to worry about how the UUIDs are formed, for instance.

That said, maybe the client using this library crafted the message beforehand and if they send a message with an existing id, we want to keep it. To enable this default behavior we merge the `messageArgs` into the `defaults` and copy those values to a new object.

Lastly we return the `ADD_MESSAGE` action with the this `thread` and new `message`.

Adding A New Message Reducer

Now we will add our `ADD_MESSAGE` handler to our `ThreadsReducer`. When a new message is added, we need to take the thread and add the message to it.

There is one tricky thing we need to handle that may not be obvious at this point: if the `thread` is the "current thread" we need to *mark this message as read.*

The user will always have one thread that is the "current thread" that they're looking at. We're going to say that if a new message is added to the current thread, then it's automatically marked as read.

code/redux/redux-chat/src/app/thread/threads.reducer.ts

```
67      case ThreadActions.ADD_MESSAGE: {
68        const thread = (<ThreadActions.AddMessageAction>action).thread;
69        const message = (<ThreadActions.AddMessageAction>action).message;
70
71        // special case: if the message being added is in the current thread, then
72        // mark it as read
73        const isRead = message.thread.id === state.currentThreadId ?
74                       true : message.isRead;
75        const newMessage = Object.assign({}, message, { isRead: isRead });
76
77        // grab the old thraed from entities
78        const oldThread = state.entities[thread.id];
79
80        // create a new thread which has our newMessage
81        const newThread = Object.assign({}, oldThread, {
82          messages: [...oldThread.messages, newMessage]
83        });
84
85        return {
86          ids: state.ids, // unchanged
87          currentThreadId: state.currentThreadId, // unchanged
88          entities: Object.assign({}, state.entities, {
89            [thread.id]: newThread
90          })
91        };
92      }
93
94      // Select a particular thread in the UI
```

The code is a bit long because we're being careful not to mutate the original thread, but it is not much different than what we've done so far in principle.

We start by extracting the `thread` and `message`.

Next we mark the message as read, if its part of the "current thread" (we'll look at how to set the current thread next).

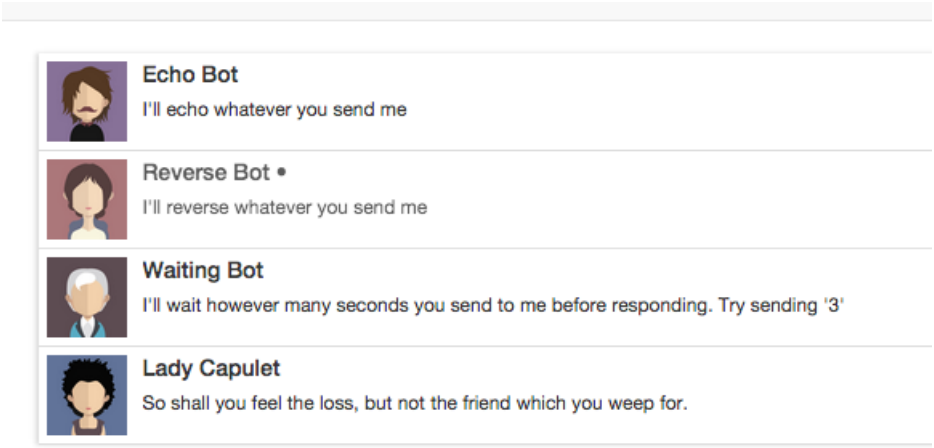
Then we grab the `oldThread` and create a `newThread` which has the `newMessage` appended on to the old `messages`.

Finally we return the new `ThreadsState`. The current list of thread `ids` and the `currentThreadId` are unchanged by adding a message, so we pass the old values here. The only thing we change is that we update `entities` with our `newThread`.

Now let's implement the last part of our data backbone: selecting a thread.

Selecting A Thread Action Creators

Our user can have multiple chat sessions in progress at the same time. However, we only have one chat window (where the user can read and send messages). When the user clicks on a thread, we want to show that thread's messages in the chat window.



Selecting A Thread

We need to keep track of which thread is the currently selected thread. To do that, we'll use the `currentThreadId` property in the `ThreadsState`.

Let's create the actions for this:

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

```
54 export const SELECT_THREAD = '[Thread] Select';
55 export interface SelectThreadAction extends Action {
56   thread: Thread;
57 }
58 export const selectThread: ActionCreator<SelectThreadAction> =
59   (thread) => ({
60     type: SELECT_THREAD,
61     thread: thread
62   });
```

There's nothing conceptually new in this action: we've got a new type of `SELECT_THREAD` and we pass the `Thread` that we're selecting as an argument.

Selecting A Thread Reducer

To select a `thread` we need to do two things:

1. set `currentThreadId` to the selected `thread`'s `id`
2. mark all messages in that `thread` as read

Here's the code for that reducer:

code/redux/redux-chat/src/app/thread/threads.reducer.ts

```
95      case ThreadActions.SELECT_THREAD: {
96        const thread = (<ThreadActions.SelectThreadAction>action).thread;
97        const oldThread = state.entities[thread.id];
98
99        // mark the messages as read
100       const newMessages = oldThread.messages.map(
101         (message) => Object.assign({}, message, { isRead: true }));
102
103       // give them to this new thread
104       const newThread = Object.assign({}, oldThread, {
105         messages: newMessages
106       });
107
108       return {
109         ids: state.ids,
110         currentThreadId: thread.id,
111         entities: Object.assign({}, state.entities, {
112           [thread.id]: newThread
113         })
114       };
115     }
116
117     default:
118       return state;
119   }
120 };
```

We start by getting the `thread`-to-select and then using that `thread.id` to get the current `Thread` that exists in `state` to get the values.

This maneuver is a bit defensive. Why not just use the `thread` that is passed in? That might be the right design decision for some apps. In this case we protect against some external mutation of `thread` by reading the last known values of that thread in `state.entities`.

Next we create a copy of all of the old messages and set them as `isRead: true`. Then we assign those new read messages to `newThread`.

Finally we return our new `ThreadsState`.

Reducers Summary

We did it! Above is everything we need for the backbone of our data architecture.

To recap, we have a `UsersReducer` which maintains the current user. We have a `ThreadsReducer` which manages:

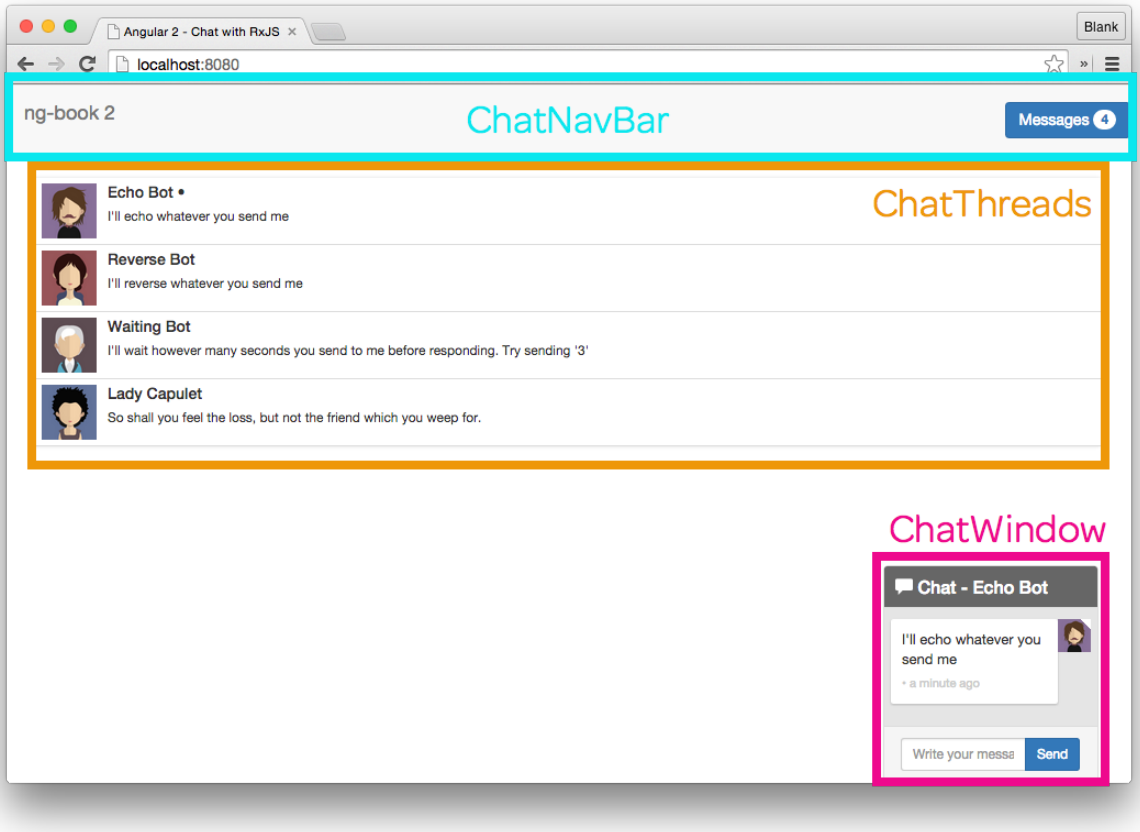
- The list of threads
- The messages in those threads
- The currently selected thread

We can derive everything else that we need (e.g. the unread count) from these pieces of data.

Now we need to hook them up to our components!

Building the Angular Chat App

As we mentioned earlier in the chapter, the page is broken down into three top-level components:



Redux Chat Top-Level Components

- `ChatNavBarComponent` - contains the unread messages count
- `ChatThreadsComponent` - shows a clickable list of threads, along with the most recent message and the conversation avatar
- `ChatWindowComponent` - shows the messages in the current thread with an input box to send new messages

We're going to bootstrap our app much like we did in the last chapter. We're going to initialize our Redux store at the top of the app and provide it via Angular's dependency injection system (take a look at the previous chapter if this looks unfamiliar):

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

```
import { InjectionToken } from '@angular/core';
import {
  createStore,
  Store,
  compose,
  StoreEnhancer
} from 'redux';

import {
  AppState,
  default as reducer
} from './app.reducer';

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

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

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

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

The top-level `AppComponent`

Our `AppComponent` component is the top-level component. It doesn't do much other than render the `ChatPage`.

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

```
import { Component, Inject } from '@angular/core';
import { Store } from 'redux';

import { AppStore } from './app.store';
import { AppState } from './app.reducer';
import { ChatExampleData } from './data/chat-example-data';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  constructor(@Inject(AppStore) private store: Store<AppState>) {
    ChatExampleData(store);
  }
}
```

and the template:

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

```
<div>
  <chat-page></chat-page>
</div>
```

For this app the bots operate on data on the client and are not connected to a server. The function `ChatExampleData()` sets up the initial data for the app. We won't be covering this code in detail in the book, so feel free to look at the code on disk if you want to learn more about how it works.

We're not using a router in this app, but if we were, we would put it here at the top level of the app. For now, we're going to create a `ChatPage` which will render the bulk of our app.

We don't have any other pages in this app, but it's a good idea to give each page it's own component in case we add some in the future.

The `ChatPage`

Our chat page renders our three main components:

- `ChatNavBarComponent`
- `ChatThreadsComponent` and
- `ChatWindowComponent`

Here it is in code:

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

```
import { Component, OnInit } from '@angular/core';

@Component({
  selector: 'chat-page',
  templateUrl: './chat-page.component.html',
  styleUrls: ['./chat-page.component.css']
})
export class ChatPageComponent implements OnInit {
  constructor() { }
  ngOnInit() { }
}
```

and the template:

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

```
<div>
  <chat-nav-bar></chat-nav-bar>
  <div class="container">
    <chat-threads></chat-threads>
    <chat-window></chat-window>
  </div>
</div>
```

For this app we are using a design pattern called *container components* and these three components are all container components. Let's talk about what that means.

Container vs. Presentational Components

It is hard to reason about our apps if there is data spread throughout all of our components. However, our apps are dynamic - they need to be populated with runtime data and they need to be responsive to user interaction.

One of the patterns that has emerged in managing this tension is the idea of presentational vs. container components. The idea is this:

1. You want to minimize the number of components which interact with outside data sources. (e.g. APIs, the Redux Store, Cookies etc.)
2. Therefore deliberately put data access into "container" components and
3. Require purely 'functional' presentation components to have all of their properties (inputs and outputs) managed by container components.

The great thing about this design is that presentational components are predictable. They're reusable because they don't make assumptions about your overall data-architecture, they only give requirements for their own use.

But even beyond reuse, they're predictable Given the same inputs, they always return the same outputs (e.g. render the same way).

If you squint, you can see that the philosophy that requires reducers to be pure functions is the same that requires presentational components be 'pure components'

It would be great if our entire app could be all presentational components, but of course, the real world has messy, changing data. So we try to put this complexity of adapting our real-world data into our container components.

If you're an advanced programmer you may see that there is a loose analogy between MVC and container/presentation components. That is, the presentational component is sort of a "view" of data that is passed in. A container component is sort of a "controller" in that it takes the "model" (the data from the rest of the app) and adapts it for the presentational components.

That said, if you haven't been programming very long, take this analogy with a grain of salt as Angular components are already a view and a controller themselves.

In our app the container components are going to be the components which interact with the store. This means our container components will be anything that:

1. Reads data from the store

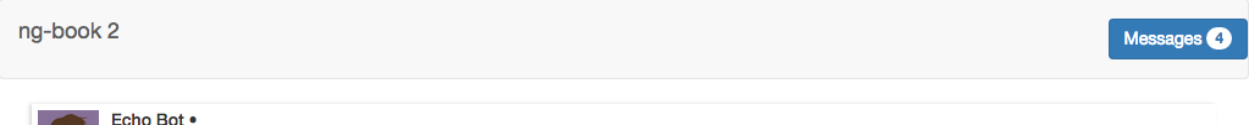
2. Subscribes to the store for changes
3. Dispatches actions to the store

Our three main components are container components and anything below them will be presentational (i.e. functional / pure / not interact with the store).

Let's build our first container component, the nav bar.

Building the `ChatNavBarComponent`

In the nav bar we'll show an unread messages count to the user.



The Unread Count in the `ChatNavBarComponent`

The best way to try out the unread messages count is to use the "Waiting Bot". If you haven't already, try sending the message '3' to the Waiting Bot and then switch to another window. The Waiting Bot will then wait 3 seconds before sending you a message and you will see the unread messages counter increment.

Let's look at the component code first:

code/redux/redux-chat/src/app/chat-nav-bar/chat-nav-bar.component.ts

```
import { Component, Inject } from '@angular/core';
import { AppStore } from '../app.store';
import { Store } from 'redux';
import {
  AppState,
  getUnreadMessagesCount
} from '../app.reducer';

@Component({
  selector: 'chat-nav-bar',
  templateUrl: './chat-nav-bar.component.html',
  styleUrls: ['./chat-nav-bar.component.css']
})
export class ChatNavBarComponent {
  unreadMessagesCount: number;

```

```
17    constructor(@Inject(AppStore) private store: Store<AppState>) {
18      store.subscribe(() => this.updateState());
19      this.updateState();
20    }
21
22    updateState() {
23      this.unreadMessagesCount = getUnreadMessagesCount(this.store.getState());
24    }
25  }
```

and the template:

code/redux/redux-chat/src/app/chat-nav-bar/chat-nav-bar.component.html

```
1  <nav class="navbar navbar-default">
2    <div class="container-fluid">
3      <div class="navbar-header">
4        <a class="navbar-brand" href="https://ng-book.com/2">
5          <img src="assets/images/logos/ng-book-2-minibook.png"/>
6           ng-book 2
7        </a>
8      </div>
9      <p class="navbar-text navbar-right">
10       <button class="btn btn-primary" type="button">
11         Messages <span class="badge">{{ unreadMessagesCount }}</span>
12       </button>
13     </p>
14   </div>
15 </nav>
```

Our template gives us the DOM structure and CSS necessary for rending a nav bar (these CSS-classes come from the CSS framework Bootstrap).

The only variable we're showing in this template us `unreadMessagesCount`.

Our `ChatNavBarComponent` has `unreadMessagesCount` as an instance variable. This number will be set to the sum of unread messages in all threads.

Notice in our `constructor` we do three things:

1. `Inject` our store
2. Subscribe to any changes in the store
3. Call `this.updateState()`

We call `this.updateState()` after `subscribe` because we want to make sure this component is initialized with the most recent data. `subscribe` will only be called if something changes *after* this component is initialized.

`updateState()` is the most interesting function - we set `unreadMessagesCount` to the value of the function `getUnreadMessagesCount`. What is `getUnreadMessagesCount` and where did it come from?

`getUnreadMessagesCount` is a new concept called *selectors*.

Redux Selectors

Thinking about our `AppState`, how might we go about getting the unread messages count? How about something like this:

```
// get the state
let state = this.store.getState();

// get the threads state
let threadsState = state.threads;

// get the entities from the threads
let threadsEntities = threadsState.entities;

// get all of the threads from state
let allThreads = Object.keys(threadsEntities)
                .map((threadId) => entities[threadId]);

// iterate over all threads and ...
let unreadCount = allThreads.reduce(
      (unreadCount: number, thread: Thread) => {
        // foreach message in that thread
        thread.messages.forEach((message: Message) => {
          if (!message.isRead) {
            // if it's unread, increment unread count
            ++unreadCount;
          }
        });
        return unreadCount;
      },
      0);
```

Should we put this logic in the `ChatNavBarComponent`? There's two problems with that approach:

1. This chunk of code reaches deep into our `AppState`. A better approach would be to co-locate this logic next to where the state itself is written.
2. What if we need the unread count somewhere else in the app? How could we share this logic?

Solving these problems is the idea behind *selectors*.

Selectors are functions that take a part of the state and return a value.

Let's take a look at how to make a few selectors.

Threads Selectors

Let's start with an easy one. Say we have our `AppState` and we want to get the `ThreadsState`:

code/redux/redux-chat/src/app/thread/threads.reducer.ts

```
122 export const getThreadsState = (state): ThreadsState => state.threads;
```

Pretty easy, right? Here we're saying, given the top-level `AppState`, we can find the `ThreadsState` at `state.threads`.

Let's say that we want to get the current thread. We could do it like this:

```
1 const getCurrentThread = (state: AppState): Thread => {
2   let currentThreadId = state.threads.currentThreadId;
3   return state.threads.entities[currentThreadId];
4 }
```

For this small example, this selector works fine. But it's worth thinking about how we can make our selectors maintainable as the app grows. It would be nice if we could use selectors to query other selectors. It also would be nice to be able to specify a selector that has multiple selectors as a dependency.

This is what the reselect[129] library provides. With `reselect` we can create small, focused selectors and then combine them together into bigger functionality.

Let's look at how we will get the current thread using `createSelector` from `reselect`.

[129] https://github.com/reactjs/reselect#createselectorinputselectors--inputselectors-resultfunc

code/redux/redux-chat/src/app/thread/threads.reducer.ts

```
124 export const getThreadsEntities = createSelector(
125   getThreadsState,
126   ( state: ThreadsState ) => state.entities );
```

We start by writing `getThreadsEntities`. `getThreadsEntities` uses `createSelector` and passes two arguments:

1. `getThreadsState`, the selector we defined above and
2. A callback function which will receive *the value of the selector in #1* and return the value we want to select.

This might seem like a lot of overhead to call `state.entities`, but it sets us up for a much more maintainable selectors down the line. Let's look at `getCurrentThread` using `createSelector`:

code/redux/redux-chat/src/app/thread/threads.reducer.ts

```
147 export const getCurrentThread = createSelector(
148   getThreadsEntities,
149   getThreadsState,
150   ( entities: ThreadsEntities, state: ThreadsState ) =>
151     entities[state.currentThreadId] );
```

Notice here that we're citing **two** selectors as dependencies: `getThreadsEntities` and `getThreadsState` - when these selectors resolve they become the arguments to the callback function. We can then combine them together to return the selected thread.

Unread Messages Count Selector

Now that we understand how selectors work, let's create a selector that will get the number of unread messages. If you look at our first attempt at unread messages above, we can see that each variable could instead become it's own selector (`getThreadsState`, `getThreadsEntities`, etc.)

Here's a selector that will get all `Threads`:

code/redux/redux-chat/src/app/thread/threads.reducer.ts

```
export const getAllThreads = createSelector(
  getThreadsEntities,
  ( entities: ThreadsEntities ) => Object.keys(entities)
                        .map((threadId) => entities[threadId]));
```

And then given all of the threads, we can get the sum of the unread messages over all threads:

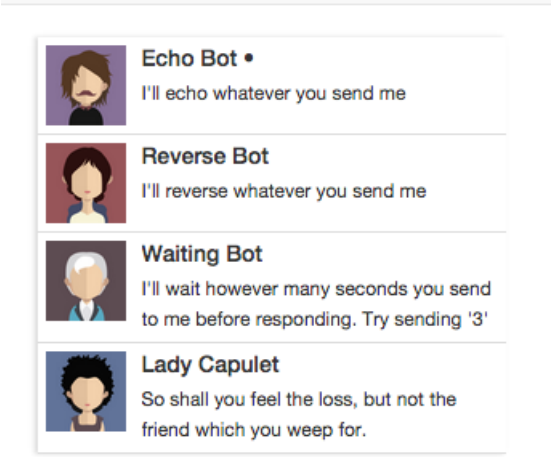
code/redux/redux-chat/src/app/thread/threads.reducer.ts

```
export const getUnreadMessagesCount = createSelector(
  getAllThreads,
  ( threads: Thread[] ) => threads.reduce(
      (unreadCount: number, thread: Thread) => {
        thread.messages.forEach((message: Message) => {
          if (!message.isRead) {
            ++unreadCount;
          }
        });
        return unreadCount;
      },
      0));
```

Now that we have this selector, we can use it to get the number of unread messages in our `ChatNavBarComponent` (and anywhere else in our app where we might need it).

Building the `ChatThreadsComponent`

Next let's build our thread list in the `ChatThreadsComponent`.



Time Ordered List of Threads

`ChatThreadsComponent` Controller

Let's take a look at our component controller `ChatThreadsComponent` before we look at the template:

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

```
import {
  Component,
  OnInit,
  Inject
} from '@angular/core';
import { AppStore } from '../app.store';
import { Store } from 'redux';
import {
  Thread
} from '../thread/thread.model';
import * as ThreadActions from '../thread/thread.actions';
import {
  AppState,
  getCurrentThread,
  getAllThreads
} from '../app.reducer';

@Component({
  selector: 'chat-threads',
  templateUrl: './chat-threads.component.html',
  styleUrls: ['./chat-threads.component.css']
```

```
})
export class ChatThreadsComponent {
  threads: Thread[];
  currentThreadId: string;

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

  updateState() {
    const state = this.store.getState();

    // Store the threads list
    this.threads = getAllThreads(state);

    // We want to mark the current thread as selected,
    // so we store the currentThreadId as a value
    this.currentThreadId = getCurrentThread(state).id;
  }

  handleThreadClicked(thread: Thread) {
    this.store.dispatch(ThreadActions.selectThread(thread));
  }
}
```

We're storing two instance variables on this component:

- `threads` - the list of `Threads`
- `currentThreadId` - the current thread (conversation) that the user is participating in

In our `constructor` we keep a reference to the Redux store and subscribe to updates. When the store changes, we call `updateState()`.

`updateState()` keeps our instance variables in sync with the Redux store. Notice that we're using two selectors:

- `getAllThreads` and
- `getCurrentThread`

which keep their respective instance variables up to date.

The one new idea we've added is an event handler: `handleThreadClicked`. `handleThreadClicked` will dispatch the `selectThread` action. The idea here is that when a thread is clicked on, we'll tell our store to set this new thread as the selected thread and the rest of the application should update in turn.

`ChatThreadsComponent` template

Let's look at the `ChatThreadsComponent` template and its configuration:

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

```
<!-- conversations -->
<div class="row">
  <div class="conversation-wrap">
    <chat-thread
         *ngFor="let thread of threads"
         [thread]="thread"
         [selected]="thread.id === currentThreadId"
         (onThreadSelected)="handleThreadClicked($event)">
    </chat-thread>
  </div>
</div>
```

In our template we're using `ngFor` to iterate over our `threads`. We're using a new directive to render the individual threads called `ChatThreadComponent`.

`ChatThreadComponent` is a *presentational* component. We **won't** be able to access the store in `ChatThreadComponent`, neither for fetching data nor dispatching actions. Instead, we're going to pass everything we need to this component through `inputs` and handle any interaction through `outputs`.

We'll look at the implementation of `ChatThreadComponent` next, but look at the inputs and outputs we have in this template first.

- We're sending the input `[thread]` with the individual `thread`
- On the input `[selected]` we're passing a *boolean* which indicates if this thread (`thread.id`) is the "current" thread (`currentThreadId`)
- If the thread is clicked, we will emit the output event `(onThreadSelected)` - when this happens we'll call `handleThreadClicked()` (which dispatches a thread selected event to the store).

Let's dig in to the `ChatThreadComponent`.

The Single `ChatThreadComponent`

The `ChatThreadComponent` will be used to display a **single thread** in the list of threads. Remember that `ChatThreadComponent` is a *presentational component* - it doesn't manipulate any data that isn't given to it directly.

Here's the component controller code:

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

```
import {
  Component,
  OnInit,
  Input,
  Output,
  EventEmitter
} from '@angular/core';
import { Thread } from '../thread/thread.model';

@Component({
  selector: 'chat-thread',
  templateUrl: './chat-thread.component.html',
  styleUrls: ['./chat-thread.component.css']
})
export class ChatThreadComponent implements OnInit {
  @Input() thread: Thread;
  @Input() selected: boolean;
  @Output() onThreadSelected: EventEmitter<Thread>;

  constructor() {
    this.onThreadSelected = new EventEmitter<Thread>();
  }

  ngOnInit() { }

  clicked(event: any): void {
    this.onThreadSelected.emit(this.thread);
    event.preventDefault();
  }
}
```

The main thing to look at here is the `onThreadSelected EventEmitter`. If you haven't used `EventEmitters` much, the idea is that it's an implementation of the observer pattern. We use it as the

"output channel" for this component - when we want to send data we call `onThreadSelected.emit` and pass whatever data we want along with it.

In this case, we want to emit the current thread as the argument to the `EventEmitter`. When this element is clicked, we will call `onThreadSelected.emit(this.thread)` which will trigger the callback in our parent (`ChatThreadsComponent`) component.

Here is where we specify our `@Input()`s of `thread` and `selected`, as well as the `@Output()` of `onThreadSelected`.

`ChatThreadComponent` template

Here's the code for our `@Component` decorator and `template`:

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

```
<div class="media conversation">
  <div class="pull-left">
    <img class="media-object avatar"
         src="{{thread.avatarSrc}}">
  </div>
  <div class="media-body">
    <h5 class="media-heading contact-name">{{thread.name}}
      <span *ngIf="selected">&bull;</span>
    </h5>
    <small class="message-preview">
      {{thread.messages[thread.messages.length - 1].text}}
    </small>
  </div>
  <a (click)="clicked($event)" class="div-link">Select</a>
</div>
```

Notice that in our view we've got some straight-forward bindings like `{{thread.avatarSrc}}`, `{{thread.name}}`. In the `message-preview` tag we've got the following:

```
  {{ thread.messages[thread.messages.length - 1].text }}
```

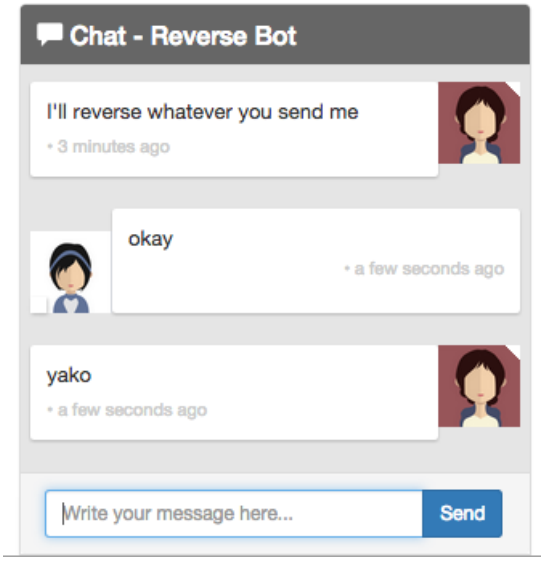
This gets the last message in the thread and displays the text of that message. The idea is we are showing a preview of the most recent message in that thread.

We've got an `*ngIf` which will show the `&bull;` symbol only if this is the selected thread.

Lastly, we're binding to the `(click)` event to call our `clicked()` handler. Notice that when we call `clicked` we're passing the argument `$event`. This is a special variable provided by Angular that describes the event. We use that in our `clicked` handler by calling `event.preventDefault();`. This makes sure that we don't navigate to a different page.

Building the `ChatWindowComponent`

The `ChatWindowComponent` is the most complicated component in our app. Let's take it one section at a time:



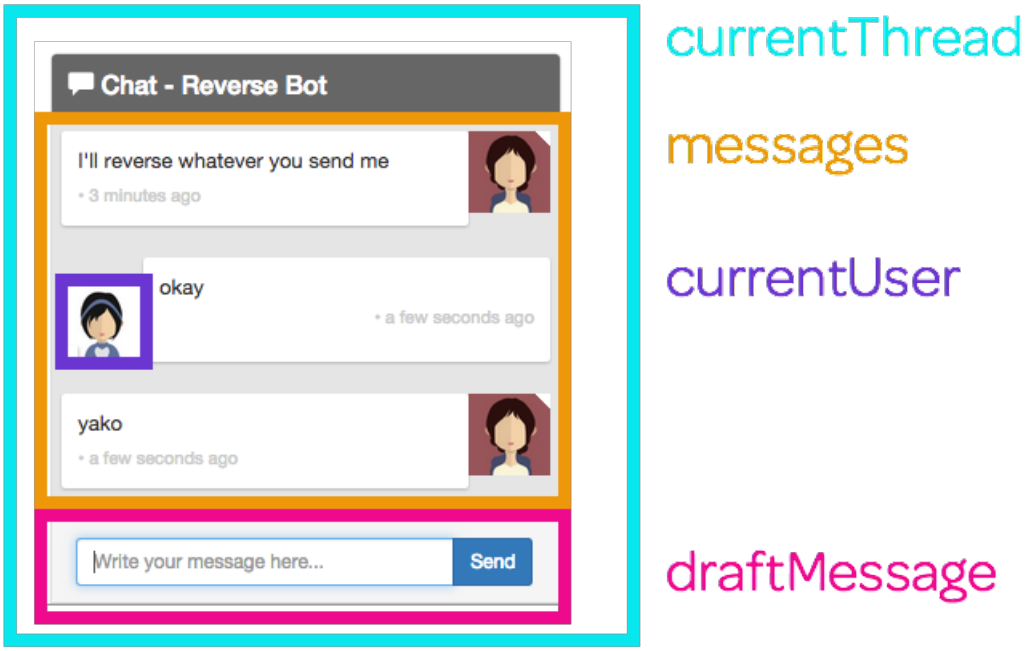
The Chat Window

Our `ChatWindowComponent` class has three properties: `currentThread` (which holds a `Thread` (that contains `Message[]` as a property), `draftMessage`, and `currentUser`:

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

```
23 export class ChatWindowComponent {
24   currentThread: Thread;
25   draftMessage: { text: string };
26   currentUser: User;
```

Here's a diagram of where each one is used:



Chat Window Properties

In our `constructor` we're going to inject two things:

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

```
28  constructor(@Inject(AppStore) private store: Store<AppState>,
29              private el: ElementRef) {
30    store.subscribe(() => this.updateState() );
31    this.updateState();
32    this.draftMessage = { text: '' };
33  }
```

The first is our Redux Store. The second, `el` is an `ElementRef` which we can use to get access to the host DOM element. We'll use that when we scroll to the bottom of the chat window when we create and receive new messages.

In our constructor we subscribe to our store, as we have in our other container components.

The next thing we do is to set a default `draftMessage` with an empty string for the `text`. We'll use the `draftMessage` to keep track of the input box as the user is typing their message.

ChatWindowComponent updateState()

When the store changes we will update the instance variables for this component:

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

```
35  updateState() {
36    const state = this.store.getState();
37    this.currentThread = getCurrentThread(state);
38    this.currentUser = getCurrentUser(state);
39    this.scrollToBottom();
40  }
```

Here we store the current thread and the current user. If a new message comes in, we also want to scroll to the bottom of the window. It's a bit coarse to call `scrollToBottom` here, but it's a simple way to make sure that the user doesn't have to scroll manually each time there is a new message (or they switch to a new thread).

`ChatWindowComponent scrollToBottom()`

To scroll to the bottom of the chat window, we're going to use the `ElementRef el` that we saved in the constructor. To make this element scroll, we're going to set the `scrollTop` property of our host element:

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

```
42  scrollToBottom(): void {
43    const scrollPane: any = this.el
44      .nativeElement.querySelector('.msg-container-base');
45    if (scrollPane) {
46      setTimeout(() => scrollPane.scrollTop = scrollPane.scrollHeight);
47    }
48  }
```

Why do we have the `setTimeout`?

If we call `scrollToBottom` immediately when we get a new message then what happens is we scroll to the bottom before the new message is rendered. By using a `setTimeout` we're telling JavaScript that we want to run this function when it is finished with the current execution queue. This happens **after** the component is rendered, so it does what we want.

`ChatWindowComponent sendMessage`

When we want to send a new message, we'll do it by taking:

- The current thread
- The current user
- The draft message text

And then dispatching a new `addMessage` action on the store. Here's what it looks like in code:

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

```
50  sendMessage(): void {
51    this.store.dispatch(ThreadActions.addMessage(
52      this.currentThread,
53      {
54        author: this.currentUser,
55        isRead: true,
56        text: this.draftMessage.text
57      }
58    ));
59    this.draftMessage = { text: '' };
60  }
```

The `sendMessage` function above takes the `draftMessage`, sets the `author` and `thread` using our component properties. Every message we send has "been read" already (we wrote it) so we mark it as read.

After we dispatch the message, we create a new `Message`** and set that new `Message` to `this.draftMessage`. This will clear the search box, and by creating a new object we ensure we don't mutate the message that was sent to the store.

`ChatWindowComponent onEnter`

In our view, we want to send the message in two scenarios

1. the user hits the "Send" button or
2. the user hits the Enter (or Return) key.

Let's define a function that will handle both events:

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

```
62   onEnter(event: any): void {
63     this.sendMessage();
64     event.preventDefault();
65   }
```

We create this `onEnter` event handler as a separate function from `sendMessage` because `onEnter` will accept an `event` as an argument and then call `event.preventDefault()`. This way we *could* call `sendMessage` in scenarios other than in response to a browser event. In this case, we're not really calling `sendMessage` in any other situation, but I find that it's nice to separate the event handler from the function that 'does the work'.

That is, a `sendMessage` function that also 1. requires an event to be passed to it and 2. handles that event is feels like a function that may be handling too many concerns.

Now that we've handled the controller code, let's look at the template

`ChatWindowComponent` template

We start our template by opening the panel tags: and showing the chat name in the header:

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

```
1  <div class="chat-window-container">
2    <div class="chat-window">
3      <div class="panel-container">
4        <div class="panel panel-default">
5
6          <div class="panel-heading top-bar">
7            <div class="panel-title-container">
8              <h3 class="panel-title">
9                <span class="glyphicon glyphicon-comment"></span>
10               Chat - {{currentThread.name}}
11             </h3>
12           </div>
13           <div class="panel-buttons-container"  >
14             <!-- you could put minimize or close buttons here -->
15           </div>
16         </div>
17
18         <div class="panel-body msg-container-base">
19           <chat-message
```

```
20                *ngFor="let message of currentThread.messages"
21                [message]="message">
22            </chat-message>
23          </div>
24
25          <div class="panel-footer">
26            <div class="input-group">
27              <input type="text"
28                     class="chat-input"
29                     placeholder="Write your message here..."
30                     (keydown.enter)="onEnter($event)"
31                     [(ngModel)]="draftMessage.text" />
32              <span class="input-group-btn">
33                <button class="btn-chat"
34                   (click)="onEnter($event)"
35                   >Send</button>
36              </span>
37            </div>
38          </div>
39
40        </div>
41      </div>
42    </div>
43  </div>
```

Next we show the list of messages. Here we use `ngFor` to iterate over our list of messages. We'll describe the individual `chat-message` component in a minute.

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

```
18          <div class="panel-body msg-container-base">
19            <chat-message
20                 *ngFor="let message of currentThread.messages"
21                 [message]="message">
22            </chat-message>
23          </div>
```

Lastly we have the message input box and closing tags:

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

```
25          <div class="panel-footer">
26            <div class="input-group">
27              <input type="text"
28                     class="chat-input"
29                     placeholder="Write your message here..."
30                     (keydown.enter)="onEnter($event)"
31                     [(ngModel)]="draftMessage.text" />
32              <span class="input-group-btn">
33                <button class="btn-chat"
34                   (click)="onEnter($event)"
35                   >Send</button>
36              </span>
37            </div>
38          </div>
39
40        </div>
41      </div>
42    </div>
```

The message `input` box is the most interesting part of this view, so let's talk about two interesting properties: 1. `(keydown.enter)` and 2. `[(ngModel)]`.

Handling keystrokes

Angular provides a straightforward way to handle keyboard actions: we bind to the event on an element. In this case, we're binding to `keydown.enter` which says if "Enter" is pressed, call the function in the expression, which in this case is `onEnter($event)`.

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

```
27              <input type="text"
28                     class="chat-input"
29                     placeholder="Write your message here..."
30                     (keydown.enter)="onEnter($event)"
31                     [(ngModel)]="draftMessage.text" />
```

Using `ngModel`

As we've talked about before, we don't generally use two-way data binding as the crux of our data architecture (like we might have in Angular 1). This is particularly true when we're using Redux which is strictly a one-way data flow.

However it can be very useful to have a two-way binding between a component and its view. As long as the side-effects are kept local to the component, it can be a very convenient way to keep a component property in sync with the view.

In this case, we're establishing a two-way bind **between the value of the input tag and `draftMessage.text`**. That is, if we type into the `input` tag, `draftMessage.text` will automatically be set to the value of that `input`. Likewise, if we were to update `draftMessage.text` in our code, the value in the `input` tag would change in the view.

Clicking "Send"

On our "Send" button we bind the `(click)` property to the `onEnter` function of our component:

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

```
33              <button class="btn-chat"
34                 (click)="onEnter($event)"
35                 >Send</button>
```

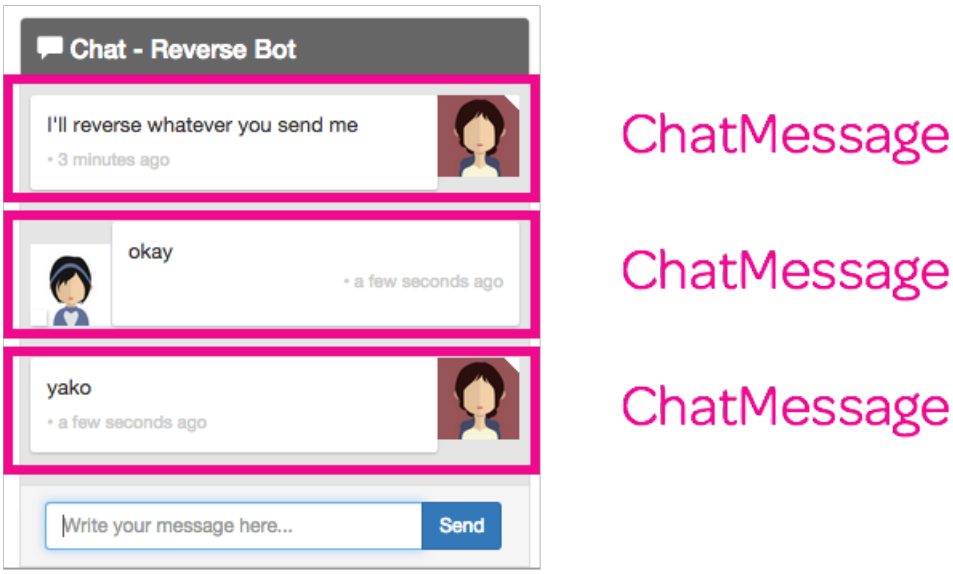
We're using the same `onEnter` function to handle the events which should send the draft message for both the button and hitting the enter button.

The `ChatMessageComponent`

Instead of putting the rendering code for each individual message in this component, instead we're going to create another *presentational component* `ChatMessageComponent`.

Tip: If you're using `ngFor` that's a good indication you should create a new component.

Each `Message` is rendered by the `ChatMessageComponent`.



The `ChatMessageComponent`

This component is relatively straightforward. The main logic here is rendering a slightly different view depending on if the message was authored by the current user. If the `Message` was **not** written by the current user, then we consider the message `incoming`.

Setting `incoming`

Remember that each `ChatMessageComponent` belongs to one `Message`. So in `ngOnInit` we will set `incoming` depending on if this `Message` was written by the current user:

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

```
import {
  Component,
  OnInit,
  Input
} from '@angular/core';
import { Message } from '../message/message.model';

@Component({
  selector: 'chat-message',
  templateUrl: './chat-message.component.html',
  styleUrls: ['./chat-message.component.css']
})
```

```
export class ChatMessageComponent implements OnInit {
  @Input() message: Message;
  incoming: boolean;

  ngOnInit(): void {
    this.incoming = !this.message.author.isClient;
  }
}
```

The `ChatMessageComponent` template

In our template we have two interesting ideas:

1. the `FromNowPipe`
2. `[ngClass]`

First, here's the code:

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

```
<div class="msg-container"
     [ngClass]="{'base-sent': !incoming, 'base-receive': incoming}">

  <div class="avatar"
       *ngIf="!incoming">
    <img src="{{message.author.avatarSrc}}">
  </div>

  <div class="messages"
    [ngClass]="{'msg-sent': !incoming, 'msg-receive': incoming}">
    <p>{{message.text}}</p>
    <p class="time">{{message.sender}} • {{message.sentAt | fromNow}}</p>
  </div>

  <div class="avatar"
       *ngIf="incoming">
    <img src="{{message.author.avatarSrc}}">
  </div>
</div>
```

The `FromNowPipe` is a pipe that casts our `Messages` sent-at time to a human-readable "x seconds ago" message. You can see that we use it by: `{{message.sentAt | fromNow}}`

`FromNowPipe` uses the excellent `moment.js`[130] library. You can read the source of the `FromNowPipe` in `code/redux/redux-chat/src/app/pipes/from-now.pipe.ts`

We also make extensive use of `ngClass` in this view. The idea is, when we say:

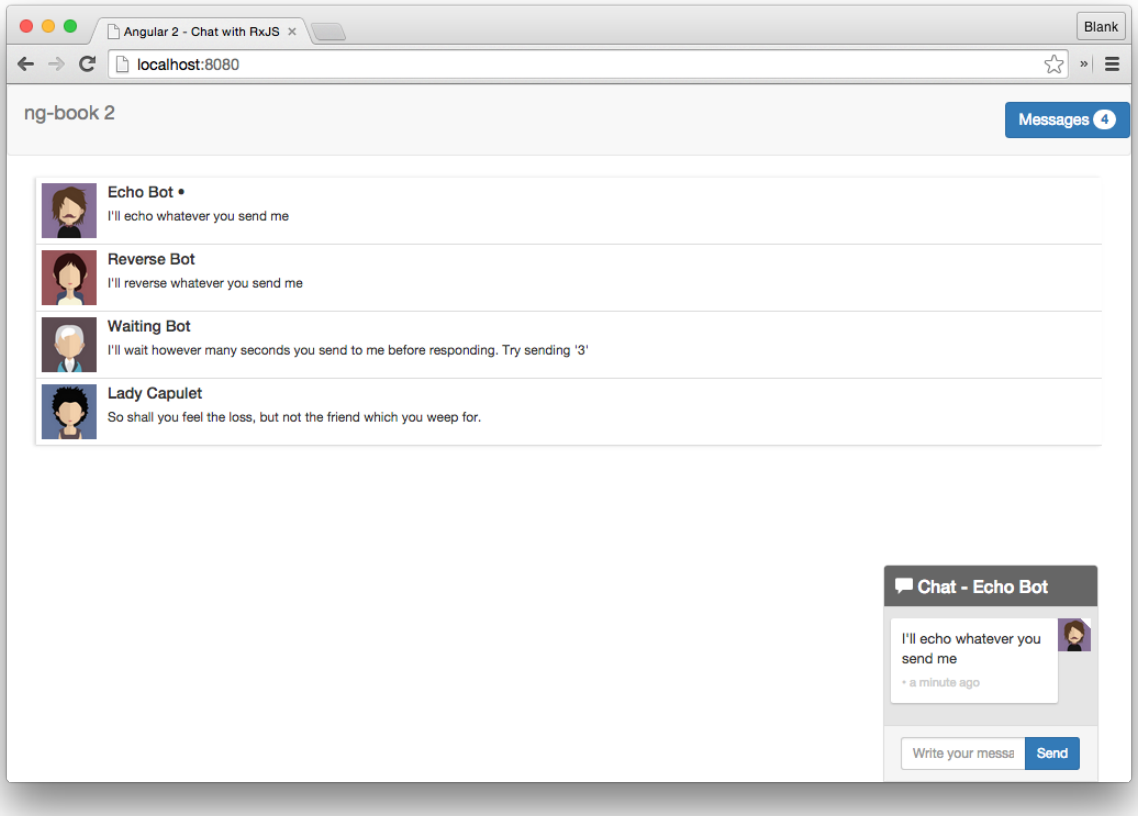
```
1     [ngClass]="{'msg-sent': !incoming, 'msg-receive': incoming}"
```

We're asking Angular to apply the `msg-receive` class if `incoming` is truthy (and apply `msg-sent` if `incoming` is falsey).

By using the `incoming` property, we're able to display incoming and outgoing messages differently.

Summary

There we go, if we put them all together we've got a fully functional chat app!



Completed Chat Application

[130] http://momentjs.com/

If you checkout `code/redux/redux-chat/src/app/data/chat-example-data.ts` you'll see we've written a handful of bots for you that you can chat with. Checkout the code and try writing a few bots of your own!