Data Architecture with Observables - Part 1: Services

Observables and RxJS

In Angular, we can structure our application to use Observables as the backbone of our data architecture. Using Observables to structure our data is called *Reactive Programming*.

But what are Observables, and Reactive Programming anyway? Reactive Programming is a way to work with asynchronous streams of data. Observables are the main data structure we use to implement Reactive Programming. But I'll admit, those terms may not be that clarifying. So we'll look at concrete examples through the rest of this chapter that should be more enlightening.

Note: Some RxJS Knowledge Required

I want to point out **this book is not primarily about Reactive Programming**. There are several other good resources that can teach you the basics of Reactive Programming and you should read them. We've listed a few below.

Consider this chapter a tutorial on how to work with RxJS and Angular rather than an exhaustive introduction to RxJS and Reactive Programming.

In this chapter, I'll **explain in detail the RxJS concepts and APIs that we encounter**. But know that you may need to supplement the content here with other resources if RxJS is still new to you.

Use of Underscore.js in this chapter

Underscore.js[76] is a popular library that provides functional operators on JavaScript data structures such as `Array` and `Object`. We use it a bunch in this chapter alongside RxJS. If you see the _ in code, such as `_.map` or `_.sortBy` know that we're using the Underscore.js library. You can find the docs for Underscore.js here[77].

Learning Reactive Programming and RxJS

If you're just learning RxJS I recommend that you read this article first:

[76] http://underscorejs.org/

[77] http://underscorejs.org/

- The introduction to Reactive Programming you've been missing[78] by Andre Staltz

After you've become a bit more familiar with the concepts behind RxJS, here are a few more links that can help you along the way:

- Which static operators to use to create streams?[79]
- Which instance operators to use on streams?[80]
- RxMarbles[81] - Interactive diagrams of the various operations on streams

Throughout this chapter I'll provide links to the API documentation of RxJS. The RxJS docs have tons of great example code that shed light on how the different streams and operators work.

Do I have to use RxJS to use Angular 4? - No, you definitely don't. Observables are just one pattern out of many that you can use with Angular 4. We talk more about other data patterns you can use here.

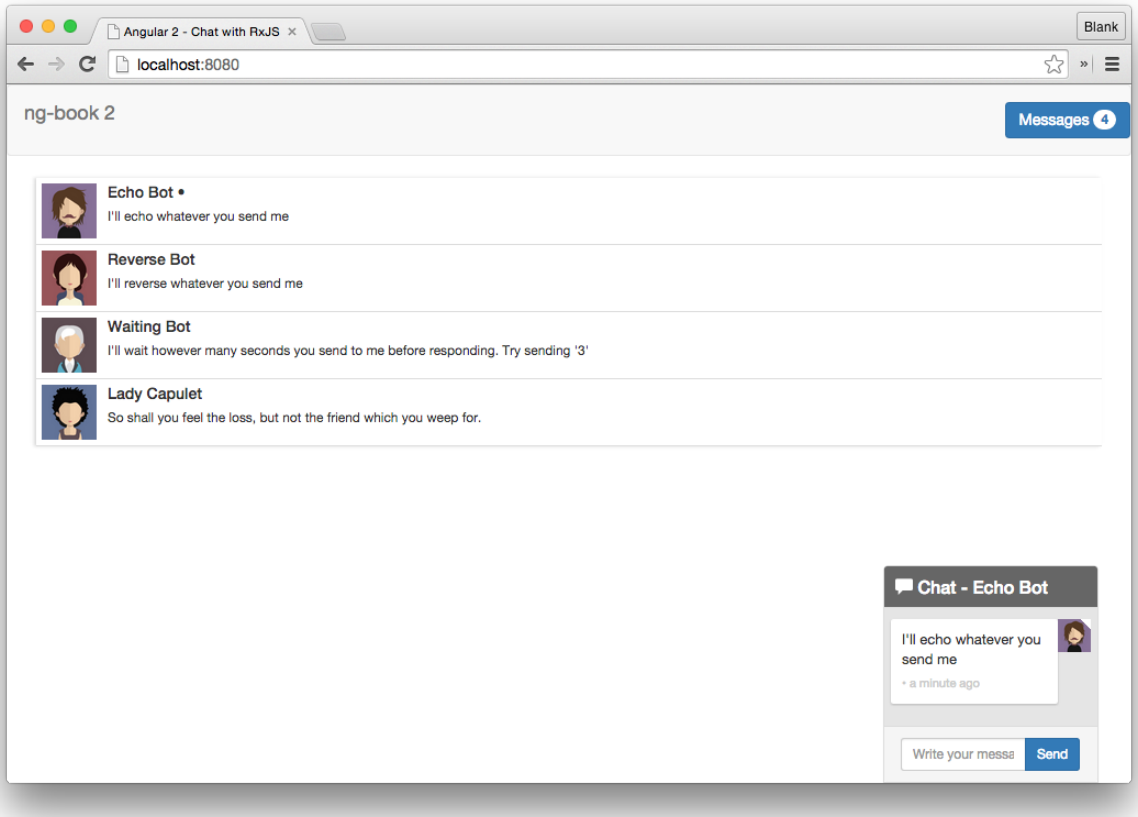
I want to give you fair warning: learning RxJS can be a bit mind-bending at first. But trust me, you'll get the hang of it and it's worth it. Here's a few big ideas about streams that you might find helpful:

1. **Promises emit a single value whereas streams emit many values**. - Streams fulfill the same role in your application as promises. If you've made the jump from callbacks to promises, you know that promises are a big improvement in readability and data maintenance vs. callbacks. In the same way, streams improve upon the promise pattern in that we can continuously respond to data changes on a stream (vs. a one-time resolve from a promise)
2. **Imperative code "pulls" data whereas reactive streams "push" data** - In Reactive Programming our code subscribes to be notified of changes and the streams "push" data to these subscribers
3. **RxJS is *functional*** - If you're a fan of functional operators like `map`, `reduce`, and `filter` then you'll feel right at home with RxJS because streams are, in some sense, lists and so the powerful functional operators all apply
4. **Streams are composable** - Think of streams like a pipeline of operations over your data. You can subscribe to any part of your stream and even combine them to create new streams

[78] https://gist.github.com/staltz/868e7e9bc2a7b8c1f754
[79] https://github.com/Reactive-Extensions/RxJS/blob/master/doc/gettingstarted/which-static.md
[80] https://github.com/Reactive-Extensions/RxJS/blob/master/doc/gettingstarted/which-instance.md
[81] http://staltz.com/rxmarbles

Chat App Overview

In this chapter, we're going to use RxJS to build a chat app. Here's a screenshot:



Completed Chat Application

Usually we try to show every line of code here in the book text. However, this chat application has a lot of moving parts, so in this chapter we're not going to have every single line of code in the text. You can find the sample code for this chapter in the folder `code/rxjs/rxjs-chat`. We'll call out each filter where you can view the context, where appropriate.

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

```
cd code/rxjs/rxjs-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

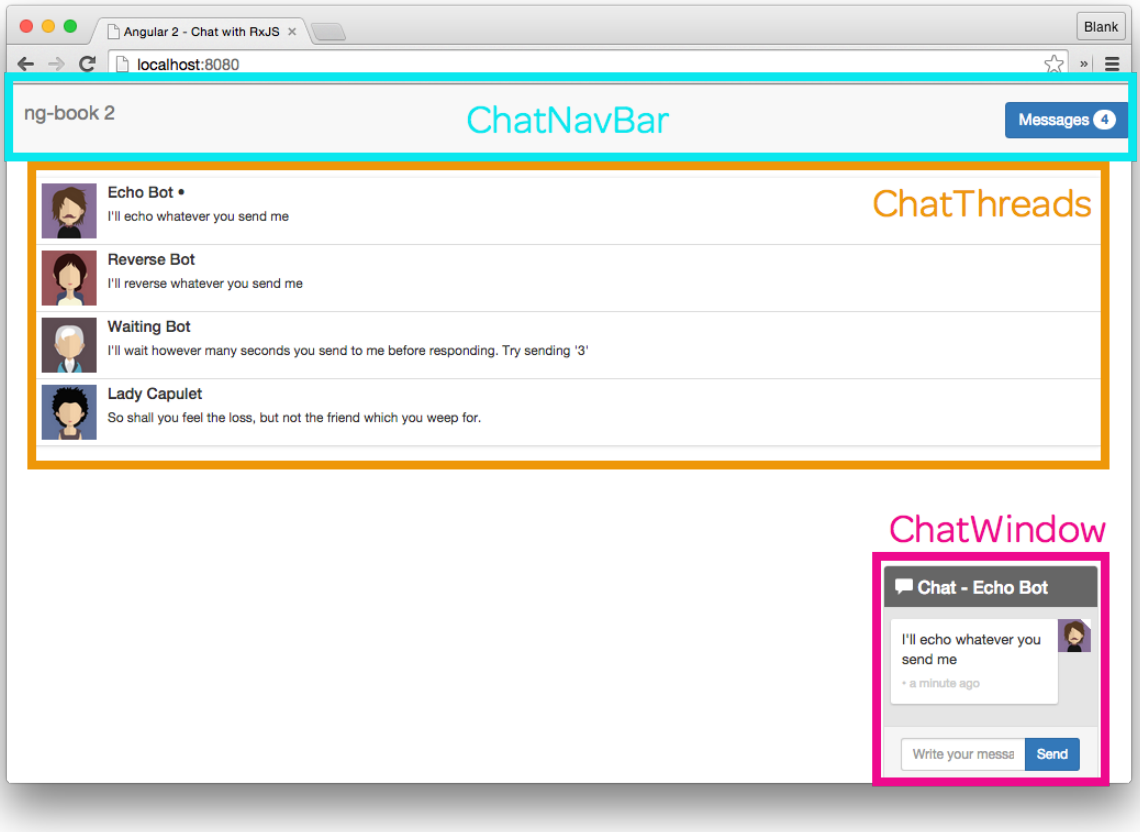
Let's look at an overview of how this app is constructed. We have

- 3 top-level Angular Components
- 3 models
- and 3 services

Let's look at them one at a time.

Components

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

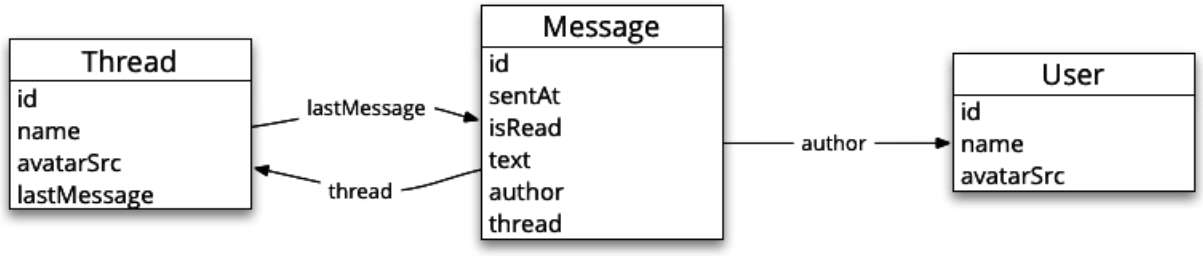


Chat Top-Level Components

- `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:



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

Services

In this app, each of our models has a corresponding *service*. The services are singleton objects that play two roles:

1. **Provide streams** of data that our application can subscribe to
2. **Provide operations** to add or modify data

For instance, the `UsersService`:

- publishes a stream that emits the current user and
- offers a `setCurrentUser` function which will set the current user (that is, emit the current user from the `currentUser` stream)

Summary

At a high level, the application data architecture is straightforward:

- The **services** maintain streams which emit models (e.g. `Messages`)
- The **components** subscribe to those streams and render according to the most recent values

For instance, the `ChatThreads` component listens for the most recent list of threads from the `ThreadService` and the `ChatWindow` subscribes for the most recent list of messages.

In the rest of this chapter, we're going to go in-depth on how we implement this using Angular `4` and RxJS. We'll start by implementing our models, then look at how we create Services to manage our streams, and then finally implement the Components.

Implementing the Models

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

User

Our `User` class is straightforward. We have an `id`, `name`, and `avatarSrc`.

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

```
1  import { uuid } from '../util/uuid';
2
3  /**
4   * A User represents an agent that sends messages
5   */
6  export class User {
7    id: string;
8
9    constructor(public name: string,
10                public avatarSrc: string) {
11     this.id = uuid();
12   }
13 }
```

Notice above that we're using a TypeScript shorthand in the constructor. When we say `public name: string` we're telling TypeScript that 1. we want `name` to be a public property on this class and 2. assign the argument value to that property when a new instance is created.

Thread

Similarly, `Thread` is also a straightforward TypeScript class:

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

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

/**
 * Thread represents a group of Users exchanging Messages
 */
 export class Thread {
   id: string;
   lastMessage: Message;
   name: string;
   avatarSrc: string;

   constructor(id?: string,
               name?: string,
               avatarSrc?: string) {
     this.id = id || uuid();
     this.name = name;
     this.avatarSrc = avatarSrc;
   }
 }
```

Note that we store a reference to the `lastMessage` in our `Thread`. This lets us show a preview of the most recent message in the threads list.

Message

`Message` is also a simple TypeScript class, however in this case we use a slightly different form of constructor:

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

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

/**
 * Message represents one message being sent in a Thread
 */
 export class Message {
   id: string;
   sentAt: Date;
```

```
11    isRead: boolean;
12    author: User;
13    text: string;
14    thread: Thread;
15
16    constructor(obj?: any) {
17      this.id              = obj && obj.id              || uuid();
18      this.isRead          = obj && obj.isRead          || false;
19      this.sentAt          = obj && obj.sentAt          || new Date();
20      this.author          = obj && obj.author          || null;
21      this.text            = obj && obj.text            || null;
22      this.thread          = obj && obj.thread          || null;
23    }
24  }
```

The pattern you see here in the constructor allows us to simulate using keyword arguments in the constructor. Using this pattern, we can create a new `Message` using whatever data we have available and we don't have to worry about the order of the arguments. For instance we could do this:

```
1   let msg1 = new Message();
2
3   # or this
4
5   let msg2 = new Message({
6     text: "Hello Nate Murray!"
7   })
```

Now that we've looked at our models, let's take a look at our first service: the `UsersService`.

Implementing `UsersService`

The point of the `UsersService` is to provide a place where our application can learn about the current user and also notify the rest of the application if the current user changes.

The first thing we need to do is create a TypeScript class and add the `@Injectable` decorator.

code/rxjs/rxjs-chat/src/app/user/users.service.ts

```
10 export class UsersService {
11   // `currentUser` contains the current user
12   currentUser: Subject<User> = new BehaviorSubject<User>(null);
13
14   public setCurrentUser(newUser: User): void {
15     this.currentUser.next(newUser);
16   }
17 }
```

We make a class that we will be able to use as a dependency to other components in our application. Briefly, two benefits of dependency-injection are:

1. we let Angular handle the lifecycle of the object and
2. it's easier to test injected components.

We talk more about `@Injectable` in the chapter on dependency injection, but the result is that we can now injector other dependencies into our constructor like so:

```
1  class UsersService {
2    constructor(public someOtherService: SomeOtherService) {
3      // do something with `someOtherService` here
4    }
5  }
```

`currentUser` stream

Next we setup a stream which we will use to manage our current user:

code/rxjs/rxjs-chat/src/app/user/users.service.ts

```
12   currentUser: Subject<User> = new BehaviorSubject<User>(null);
```

There's a lot going on here, so let's break it down:

- We're defining an instance variable `currentUser` which is a `Subject` stream.
- Concretely, `currentUser` is a `BehaviorSubject` which will contain `User`.

- However, the first value of this stream is `null` (the constructor argument).

If you haven't worked with RxJS much, then you may not know what `Subject` or `BehaviorSubject` are. You can think of a `Subject` as a "read/write" stream.

Technically a `Subject`[82] inherits from both `Observable`[83] and `Observer`[84]

One consequence of streams is that, because messages are published immediately, a new subscriber risks missing the latest value of the stream. BehaviourSubject compensates for this.

`BehaviourSubject`[85] has a special property in that it **stores the last value**. Meaning that any subscriber to the stream will receive the latest value. This is great for us because it means that any part of our application can subscribe to the `UsersService.currentUser` stream and immediately know who the current user is.

Setting a new user

We need a way to publish a new user to the stream whenever the current user changes (e.g. logging in).

There's two ways we can expose an API for doing this:

1. Add new users to the stream directly:

The most straightforward way to update the current user is to have clients of the `UsersService` simply publish a new `User` directly to the stream like this:

```
1  UsersService.currentUser.subscribe((newUser) => {
2    console.log('New User is: ', newUser.name);
3  })
4
5  // => New User is: originalUserName
6
7  let u = new User('Nate', 'anImgSrc');
8  UsersService.currentUser.next(u);
9
10 // => New User is: Nate
```

[82] https://github.com/Reactive-Extensions/RxJS/blob/master/doc/api/subjects/subject.md
[83] https://github.com/Reactive-Extensions/RxJS/blob/master/doc/api/core/observable.md
[84] https://github.com/Reactive-Extensions/RxJS/blob/master/doc/api/core/observer.md
[85] https://github.com/Reactive-Extensions/RxJS/blob/master/doc/api/subjects/behaviorsubject.md

Note here that we use the `next` method on a `Subject` to push a new value to the stream

The pro here is that we're able to reuse the existing API from the stream, so we're not introducing any new code or APIs

2. Create a `setCurrentUser(newUser: User)` method

The other way we could update the current user is to create a helper method on the `UsersService` like this:

code/rxjs/rxjs-chat/src/app/user/users.service.ts

```
14   public setCurrentUser(newUser: User): void {
15     this.currentUser.next(newUser);
16   }
```

You'll notice that we're still using the `next` method on the `currentUser` stream, so why bother doing this?

Because there is value in decoupling the implementation of the currentUser from the implementation of the stream. By wrapping the `next` in the `setCurrentUser` call we give ourselves room to change the implementation of the `UsersService` without breaking our clients.

In this case, I wouldn't recommend one method very strongly over the other, but it can make a big difference on the maintainability of larger projects.

A third option could be to have the updates expose streams of their own (that is, a stream where we place the action of changing the current user). We explore this pattern in the `MessagesService` below.

`UsersService.ts`

Putting it together, our `UsersService` looks like this:

code/rxjs/rxjs-chat/src/app/user/users.service.ts

```
import { Injectable } from '@angular/core';
import { Subject, BehaviorSubject } from 'rxjs';
import { User } from './user.model';


/**
 * UserService manages our current user
 */
@Injectable()
export class UsersService {
  // `currentUser` contains the current user
  currentUser: Subject<User> = new BehaviorSubject<User>(null);

  public setCurrentUser(newUser: User): void {
    this.currentUser.next(newUser);
  }
}

export const userServiceInjectables: Array<any> = [
  UsersService
];
```

The `MessagesService`

The `MessagesService` is the backbone of this application. In our app, all messages flow through the `MessagesService`.

Our `MessagesService` has much more sophisticated streams compared to our `UsersService`. There are five streams that make up our `MessagesService`: 3 "data management" streams and 2 "action" streams.

The three data management streams are:

- `newMessages` - emits each new `Message` only once
- `messages` - emits **an array** of the current `Messages`
- `updates` - performs operations on `messages`

the `newMessages` stream

`newMessages` is a `Subject` that will publish each new `Message` only once.

code/rxjs/rxjs-chat/src/app/message/messages.service.ts

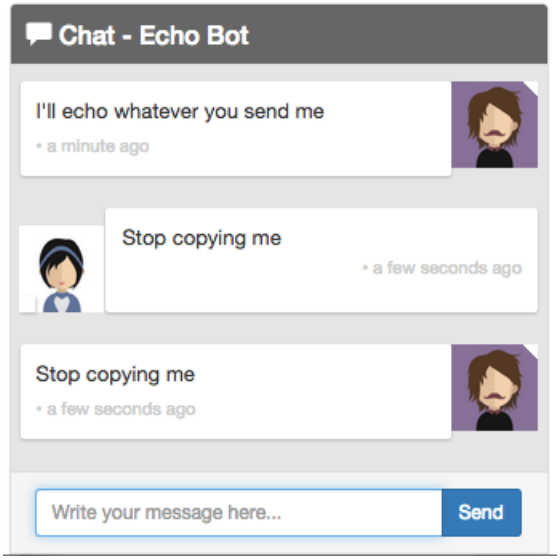
```
14 export class MessagesService {
15   // a stream that publishes new messages only once
16   newMessages: Subject<Message> = new Subject<Message>();
```

If we want, we can define a helper method to add `Messages` to this stream:

code/rxjs/rxjs-chat/src/app/message/messages.service.ts

```
90   addMessage(message: Message): void {
91     this.newMessages.next(message);
92   }
```

It would also be helpful to have a stream that will get all of the messages from a thread that are not from a particular user. For instance, consider the Echo Bot:



Real mature, Echo Bot

When we are implementing the Echo Bot, we don't want to enter an infinite loop and repeat back the bot's messages to itself.

To implement this we can subscribe to the `newMessages` stream and filter out all messages that are

1. part of this thread and
2. not written by the bot.

You can think of this as saying, for a given `Thread` I want a stream of the messages that are "for" this `User`.

code/rxjs/rxjs-chat/src/app/message/messages.service.ts

```
94   messagesForThreadUser(thread: Thread, user: User): Observable<Message> {
95     return this.newMessages
96       .filter((message: Message) => {
97               // belongs to this thread
98         return (message.thread.id === thread.id) &&
99               // and isn't authored by this user
100              (message.author.id !== user.id);
101      });
102  }
```

`messagesForThreadUser` takes a `Thread` and a `User` and returns a new stream of `Messages` that are filtered on that `Thread` and not authored by the `User`. That is, it is a stream of "everyone else's" messages in this `Thread`.

the `messages` stream

Whereas `newMessages` emits individual `Messages`, the `messages` stream emits **an Array of the most recent `Messages`**.

code/rxjs/rxjs-chat/src/app/message/messages.service.ts

```
19   messages: Observable<Message[]>;
```

The type `Message[]` is the same as `Array<Message>`. Another way of writing the same thing would be: `Observable<Array<Message>>`. When we define the type of `messages` to be `Observable<Message[]>` we mean that this stream emits an **`Array`** (of `Messages`), not individual `Messages`.

So how does `messages` get populated? For that we need to talk about the `updates` stream and a new pattern: the Operation stream.

The Operation Stream Pattern

Here's the idea:

- We'll maintain state in `messages` which will hold an `Array` of the most current `Messages`
- We use an `updates` stream which is a **stream of functions** to apply to `messages`

You can think of it this way: any function that is put on the `updates` stream will change the list of the current messages. A function that is put on the `updates` stream should **accept a list of `Messages`** and then **return a list of `Messages`**. Let's formalize this idea by creating an interface in code:

code/rxjs/rxjs-chat/src/app/message/messages.service.ts

```
9  interface IMessagesOperation extends Function {
10   (messages: Message[]): Message[];
11 }
```

Let's define our `updates` stream:

code/rxjs/rxjs-chat/src/app/message/messages.service.ts

```
21   // `updates` receives _operations_ to be applied to our `messages`
22   // it's a way we can perform changes on *all* messages (that are currently
23   // stored in `messages`)
24   updates: Subject<any> = new Subject<any>();
```

Remember, `updates` receives *operations* that will be applied to our list of messages. But how do we make that connection? We do (in the `constructor` of our `MessagesService`) like this:

code/rxjs/rxjs-chat/src/app/message/messages.service.ts

```
30   constructor() {
31     this.messages = this.updates
32       // watch the updates and accumulate operations on the messages
33       .scan((messages: Message[],
34              operation: IMessagesOperation) => {
35                return operation(messages);
36              },
37             initialMessages)
38       // make sure we can share the most recent list of messages across anyone
```

This code introduces a new stream function: `scan`[86]. If you're familiar with functional programming, `scan` is a lot like `reduce`: it runs the function for each element in the incoming stream and **accumulates a value**. What's special about `scan` is that it will **emit a value for each intermediate result**. That is, it doesn't wait for the stream to complete before emitting a result, which is exactly what we want.

When we call `this.updates.scan`, we are creating a new stream that is subscribed to the `updates` stream. On each pass, we're given:

1. the `messages` we're accumulating and
2. the new `operation` to apply.

and then we return the new `Message[]`.

[86] https://github.com/Reactive-Extensions/RxJS/blob/master/doc/api/core/operators/scan.md

Sharing the Stream

One thing to know about streams is that they aren't shareable by default. That is, if one subscriber reads a value from a stream, it can be gone forever. In the case of our messages, we want to 1. share the same stream among many subscribers and 2. replay the last value for any subscribers who come "late".

To do that, we use two operators: `publishReplay` and `refCount`.

- `publishReplay` let's us share a subscription between multiple subscribers and replay *n* number of values to future subscribers. (see `publish`[87] and `replay`[88])
- `refCount`[89] - makes it easier to use the return value of publish, by managing when the observable will emit values

Wait, so what does `refCount` do?

`refCount` can be a little tricky to understand because it relates to how one manages "hot" and "cold" observables. We're not going to dive deep into explaining how this works and we direct the reader to:

- RxJS docs on `refCount`[90]
- Introduction to Rx: Hot and Cold observables[91]
- RefCount Marble Diagram[92]

code/rxjs/rxjs-chat/src/app/message/messages.service.ts

```
32      // watch the updates and accumulate operations on the messages
33      .scan((messages: Message[],
34             operation: IMessagesOperation) => {
35               return operation(messages);
36             },
37            initialMessages)
38      // make sure we can share the most recent list of messages across anyone
39      // who's interested in subscribing and cache the last known list of
40      // messages
41      .publishReplay(1)
42      .refCount();
```

[87] https://github.com/Reactive-Extensions/RxJS/blob/master/doc/api/core/operators/publish.md
[88] https://github.com/Reactive-Extensions/RxJS/blob/master/doc/api/core/operators/replay.md
[89] https://github.com/Reactive-Extensions/RxJS/blob/master/doc/api/core/operators/refcount.md
[90] https://github.com/Reactive-Extensions/RxJS/blob/master/doc/api/core/operators/refcount.md
[91] http://www.introtorx.com/Content/v1.0.10621.0/14_HotAndColdObservables.html#RefCount
[92] http://reactivex.io/documentation/operators/refcount.html

Adding `Message`s to the `messages` Stream

Now we could add a `Message` to the `messages` stream like so:

```
var myMessage = new Message(/* params here... */);

updates.next( (messages: Message[]): Message[] => {
  return messages.concat(myMessage);
})
```

Above, we're adding an operation to the `updates` stream. The effect is that `messages` is "subscribed" to that stream and so it will apply that operation which will `concat` our `newMessage` on to the accumulated list of `messages`.

It's okay if this takes a few minutes to mull over. It can feel a little foreign if you're not used to this style of programming.

One problem with the above approach is that it's a bit verbose to use. It would be nice to not have to write that inner function every time. We could do something like this:

```
addMessage(newMessage: Message) {
  updates.next( (messages: Message[]): Message[] => {
    return messages.concat(newMessage);
  })
}

// somewhere else

var myMessage = new Message(/* params here... */);
MessagesService.addMessage(myMessage);
```

This is a little bit better, but it's not "the reactive way". In part, because this action of creating a message isn't composable with other streams. (Also this method is circumventing our `newMessages` stream. More on that later.)

A reactive way of creating a new message would be **to have a stream that accepts `Messages` to add to the list**. Again, this can be a bit new if you're not used to thinking this way. Here's how you'd implement it:

First we make an "action stream" called `create`. (The term "action stream" is only meant to describe its role in our service. The stream itself is still a regular `Subject`):

code/rxjs/rxjs-chat/src/app/message/messages.service.ts

```
26    // action streams
27    create: Subject<Message> = new Subject<Message>();
```

Next, in our `constructor` we configure the `create` stream:

code/rxjs/rxjs-chat/src/app/message/messages.service.ts

```
58      this.create
59        .map( function(message: Message): IMessagesOperation {
60          return (messages: Message[]) => {
61            return messages.concat(message);
62          };
63        })
```

The `map`[93] operator is a lot like the built-in `Array.map` function in JavaScript except that it works on streams. That is, it runs the function once for each item in the stream and emits the return value of the function.

In this case, we're saying "for each `Message` we receive as input, return an `IMessagesOperation` that adds this message to the list". Put another way, this stream will emit a **function** which accepts the list of `Messages` and adds this `Message` to our list of messages.

Now that we have the `create` stream, we still have one thing left to do: we need to actually hook it up to the `updates` stream. We do that by using `subscribe`[94].

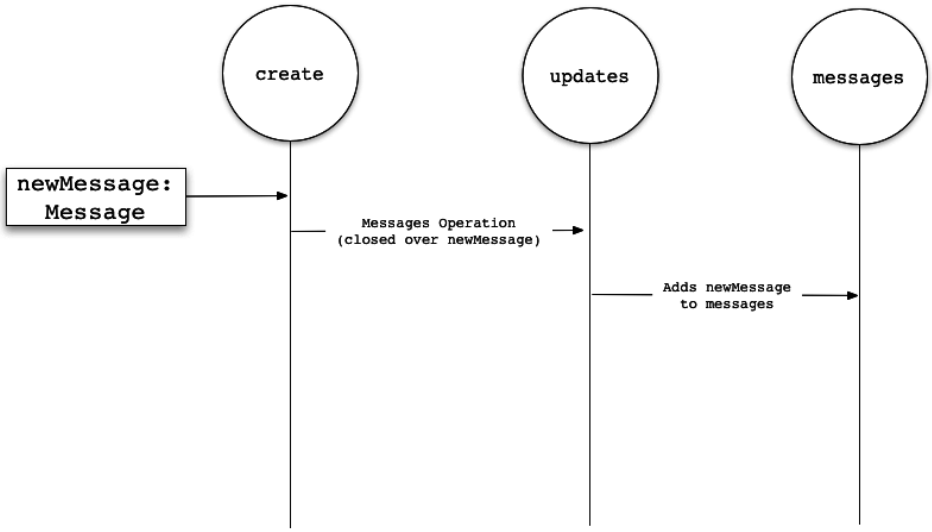
code/rxjs/rxjs-chat/src/app/message/messages.service.ts

```
58      this.create
59        .map( function(message: Message): IMessagesOperation {
60          return (messages: Message[]) => {
61            return messages.concat(message);
62          };
63        })
64        .subscribe(this.updates);
```

What we're doing here is *subscribing* the `updates` stream to listen to the `create` stream. This means that if `create` receives a `Message` it will emit an `IMessagesOperation` that will be received by `updates` and then the `Message` will be added to `messages`.

Here's a diagram that shows our current situation:

[93] https://github.com/Reactive-Extensions/RxJS/blob/master/doc/api/core/operators/select.md
[94] https://github.com/Reactive-Extensions/RxJS/blob/master/doc/api/core/operators/subscribe.md



Creating a new message, starting with the `create` stream

This is great because it means we get a few things:

1. The current list of messages from `messages`
2. A way to process operations on the current list of messages (via `updates`)
3. An easy-to-use stream to put create operations on our `updates` stream (via `create`)

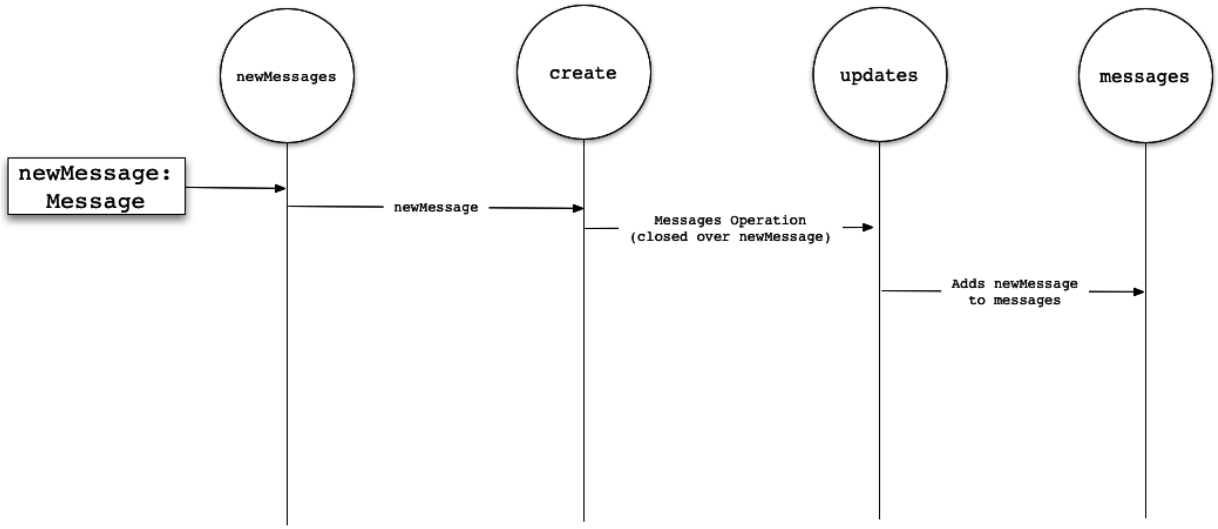
Anywhere in our code, if we want to get the most current list of messages, we just have to go to the `messages` stream. But we have a problem, **we still haven't connected this flow to the `newMessages` stream**.

It would be great if we had a way to easily connect this stream with any `Message` that comes from `newMessages`. It turns out, it's really easy:

code/rxjs/rxjs-chat/src/app/message/messages.service.ts

```
66    this.newMessages
67      .subscribe(this.create);
```

Now our diagram looks like this:



Creating a new message, starting with the `newMessages` stream

Now our flow is complete! It's the best of both worlds: we're able to subscribe to the stream of individual messages through `newMessages`, but if we just want the most up-to-date list, we can subscribe to `messages`.

It's worth pointing out some implications of this design: if you subscribe to `newMessages` directly, you have to be careful about changes that may happen downstream. Here are three things to consider:

First, you obviously won't get any downstream updates that are applied to the `Messages`.

Second, in this case, we have **mutable** `Message` objects. So if you subscribe to `newMessages` and store a reference to a `Message`, that `Message`'s attributes may change.

Third, in the case where you want to take advantage of the mutability of our `Messages` you may not be able to. Consider the case where we could put an operation on the `updates` queue that makes a copy of each `Message` and then mutates the copy. (This is probably a better design than what we're doing here.) In this case, you couldn't rely on any `Message` emitted directly from `newMessages` being in its "final" state.

That said, as long as you keep these considerations in mind, you shouldn't have too much trouble.

Our completed `MessagesService`

Here's what the completed `MessagesService` looks like:

code/rxjs/rxjs-chat/src/app/message/messages.service.ts

```
import { Injectable } from '@angular/core';
import { Subject, Observable } from 'rxjs';
import { User } from '../user/user.model';
import { Thread } from '../thread/thread.model';
import { Message } from '../message/message.model';

const initialMessages: Message[] = [];

interface IMessagesOperation extends Function {
  (messages: Message[]): Message[];
}

@Injectable()
export class MessagesService {
  // a stream that publishes new messages only once
  newMessages: Subject<Message> = new Subject<Message>();

  // `messages` is a stream that emits an array of the most up to date messages
  messages: Observable<Message[]>;

  // `updates` receives _operations_ to be applied to our `messages`
  // it's a way we can perform changes on *all* messages (that are currently
  // stored in `messages`)
  updates: Subject<any> = new Subject<any>();

  // action streams
  create: Subject<Message> = new Subject<Message>();
  markThreadAsRead: Subject<any> = new Subject<any>();

  constructor() {
    this.messages = this.updates
      // watch the updates and accumulate operations on the messages
      .scan((messages: Message[],
             operation: IMessagesOperation) => {
               return operation(messages);
             },
            initialMessages)
      // make sure we can share the most recent list of messages across anyone
      // who's interested in subscribing and cache the last known list of
      // messages
      .publishReplay(1)
```

```
42        .refCount();
43
44      // `create` takes a Message and then puts an operation (the inner function)
45      // on the `updates` stream to add the Message to the list of messages.
46      //
47      // That is, for each item that gets added to `create` (by using `next`)
48      // this stream emits a concat operation function.
49      //
50      // Next we subscribe `this.updates` to listen to this stream, which means
51      // that it will receive each operation that is created
52      //
53      // Note that it would be perfectly acceptable to simply modify the
54      // "addMessage" function below to simply add the inner operation function to
55      // the update stream directly and get rid of this extra action stream
56      // entirely. The pros are that it is potentially clearer. The cons are that
57      // the stream is no longer composable.
58      this.create
59        .map( function(message: Message): IMessagesOperation {
60          return (messages: Message[]) => {
61            return messages.concat(message);
62          };
63        })
64        .subscribe(this.updates);
65
66      this.newMessages
67        .subscribe(this.create);
68
69      // similarly, `markThreadAsRead` takes a Thread and then puts an operation
70      // on the `updates` stream to mark the Messages as read
71      this.markThreadAsRead
72        .map( (thread: Thread) => {
73          return (messages: Message[]) => {
74            return messages.map( (message: Message) => {
75              // note that we're manipulating `message` directly here. Mutability
76              // can be confusing and there are lots of reasons why you might want
77              // to, say, copy the Message object or some other 'immutable' here
78              if (message.thread.id === thread.id) {
79                message.isRead = true;
80              }
81              return message;
82            });
83          };
```

```
84        })
85        .subscribe(this.updates);
86
87    }
88
89    // an imperative function call to this action stream
90    addMessage(message: Message): void {
91      this.newMessages.next(message);
92    }
93
94    messagesForThreadUser(thread: Thread, user: User): Observable<Message> {
95      return this.newMessages
96        .filter((message: Message) => {
97                // belongs to this thread
98          return (message.thread.id === thread.id) &&
99                // and isn't authored by this user
100               (message.author.id !== user.id);
101       });
102   }
103 }
104
105 export const messagesServiceInjectables: Array<any> = [
106   MessagesService
107 ];
```

Trying out `MessagesService`

If you haven't already, this would be a good time to open up the code and play around with the `MessagesService` to get a feel for how it works. We've got an example you can start with in `code/rxjs/rxjs-chat/src/app/message/messages.service.spec.ts`.

To run the tests in this project, open up your terminal then:

```
1 cd /path/to/code/rxjs/rxjs-chat // <-- your path will vary
2 npm install
3 npm run test
```

Let's start by creating a few instances of our models to use:

code/rxjs/rxjs-chat/src/app/message/messages.service.spec.ts

```
import { MessagesService } from './messages.service';

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

describe('MessagesService', () => {
  it('should test', () => {

    const user: User = new User('Nate', '');
    const thread: Thread = new Thread('t1', 'Nate', '');
    const m1: Message = new Message({
      author: user,
      text: 'Hi!',
      thread: thread
    });

    const m2: Message = new Message({
      author: user,
      text: 'Bye!',
      thread: thread
    });
```

Next let's subscribe to a couple of our streams:

code/rxjs/rxjs-chat/src/app/message/messages.service.spec.ts

```
    const messagesService: MessagesService = new MessagesService();

    // listen to each message indivdually as it comes in
    messagesService.newMessages
      .subscribe( (message: Message) => {
        console.log('=> newMessages: ' + message.text);
      });

    // listen to the stream of most current messages
    messagesService.messages
      .subscribe( (messages: Message[]) => {
        console.log('=> messages: ' + messages.length);
      });

```

```
38      messagesService.addMessage(m1);
39      messagesService.addMessage(m2);
40
41      // => messages: 1
42      // => newMessages: Hi!
43      // => messages: 2
44      // => newMessages: Bye!
45    });
46
47
48  });
```

Notice that even though we subscribed to `newMessages` first and `newMessages` is called directly by `addMessage`, our `messages` subscription is logged first. The reason for this is because `messages` subscribed to `newMessages` earlier than our subscription in this test (when `MessagesService` was instantiated). (You shouldn't be relying on the ordering of independent streams in your code, but why it works this way is worth thinking about.)

Play around with the `MessagesService` and get a feel for the streams there. We're going to be using them in the next section where we build the `ThreadsService`.

The `ThreadsService`

On our `ThreadsService` were going to define four streams that emit respectively:

1. A map of the current set of `Threads` (in `threads`)
2. A chronological list of `Threads`, newest-first (in `orderedthreads`)
3. The currently selected `Thread` (in `currentThread`)
4. The list of `Messages` for the currently selected `Thread` (in `currentThreadMessages`)

Let's walk through how to build each of these streams, and we'll learn a little more about RxJS along the way.

A map of the current set of `Thread`s (in `threads`)

Let's start by defining our `ThreadsService` class and the instance variable that will emit the `Threads`:

code/rxjs/rxjs-chat/src/app/thread/threads.service.ts

```
1  import { Injectable } from '@angular/core';
2  import { Subject, BehaviorSubject, Observable } from 'rxjs';
3  import { Thread } from './thread.model';
4  import { Message } from '../message/message.model';
5  import { MessagesService } from '../message/messages.service';
6  import * as _ from 'lodash';
7
8  @Injectable()
9  export class ThreadsService {
10
11   // `threads` is a observable that contains the most up to date list of threads
12   threads: Observable<{ [key: string]: Thread }>;
```

Notice that this stream will emit a map (an object) with the `id` of the `Thread` being the `string` key and the `Thread` itself will be the value.

To create a stream that maintains the current list of threads, we start by attaching to the `messagesService.messages` stream:

code/rxjs/rxjs-chat/src/app/thread/threads.service.ts

```
12   threads: Observable<{ [key: string]: Thread }>;
```

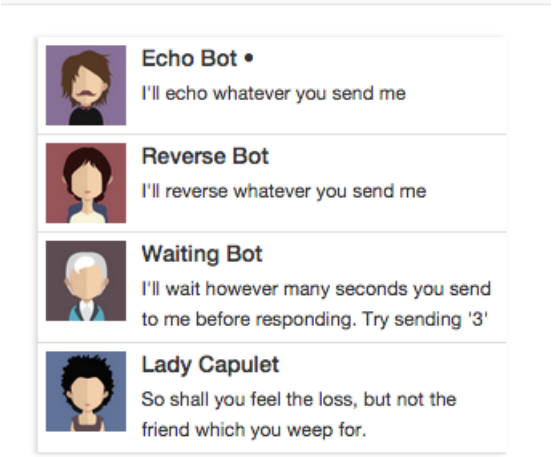
Recall that each time a new `Message` is added to the steam, `messages` will emit an array of the current `Messages`. We're going to look at each `Message` and we want to return a unique list of the `Threads`.

code/rxjs/rxjs-chat/src/app/thread/threads.service.ts

```
27     this.threads = messagesService.messages
28       .map( (messages: Message[]) => {
29         const threads: {[key: string]: Thread} = {};
30         // Store the message's thread in our accumulator `threads`
31         messages.map((message: Message) => {
32           threads[message.thread.id] = threads[message.thread.id] ||
33             message.thread;
```

Notice above that each time we will create a new list of `threads`. The reason for this is because we might delete some messages down the line (e.g. leave the conversation). Because we're recalculating the list of threads each time, we naturally will "delete" a thread if it has no messages.

In the threads list, we want to show a preview of the chat by using the text of the most recent `Message` in that `Thread`.



List of Threads with Chat Preview

In order to do that, we'll store the most recent `Message` for each `Thread`. We know which `Message` is newest by comparing the `sentAt` times:

code/rxjs/rxjs-chat/src/app/thread/threads.service.ts

```
34          // Cache the most recent message for each thread
35          const messagesThread: Thread = threads[message.thread.id];
36          if (!messagesThread.lastMessage ||
37              messagesThread.lastMessage.sentAt < message.sentAt) {
38            messagesThread.lastMessage = message;
39          }
40        });
41        return threads;
42      });
```

Putting it all together, `threads` looks like this:

code/rxjs/rxjs-chat/src/app/thread/threads.service.ts

```
27    this.threads = messagesService.messages
28      .map( (messages: Message[]) => {
29        const threads: {[key: string]: Thread} = {};
30        // Store the message's thread in our accumulator `threads`
31        messages.map((message: Message) => {
32          threads[message.thread.id] = threads[message.thread.id] ||
33            message.thread;
34
35          // Cache the most recent message for each thread
```

```
36            const messagesThread: Thread = threads[message.thread.id];
37            if (!messagesThread.lastMessage ||
38                messagesThread.lastMessage.sentAt < message.sentAt) {
39              messagesThread.lastMessage = message;
40            }
41          });
42          return threads;
43        });
```

Trying out the `ThreadsService`

Let's try out our `ThreadsService`. First we'll create a few models to work with:

code/rxjs/rxjs-chat/src/app/thread/threads.service.spec.ts

```
1  import { Message } from './../message/message.model';
2  import { Thread } from './thread.model';
3  import { User } from './../user/user.model';
4
5  import { ThreadsService } from './threads.service';
6  import { MessagesService } from './../message/messages.service';
7  import * as _ from 'lodash';
8
9  describe('ThreadsService', () => {
10   it('should collect the Threads from Messages', () => {
11
12     const nate: User = new User('Nate Murray', '');
13     const felipe: User = new User('Felipe Coury', '');
14
15     const t1: Thread = new Thread('t1', 'Thread 1', '');
16     const t2: Thread = new Thread('t2', 'Thread 2', '');
17
18     const m1: Message = new Message({
19       author: nate,
20       text: 'Hi!',
21       thread: t1
22     });
23
24     const m2: Message = new Message({
25       author: felipe,
26       text: 'Where did you get that hat?',
27       thread: t1
28     });
```

```
29
30      const m3: Message = new Message({
31        author: nate,
32        text: 'Did you bring the briefcase?',
33        thread: t2
34      });
```

Now let's create an instance of our services:

code/rxjs/rxjs-chat/src/app/thread/threads.service.spec.ts

```
36      const messagesService: MessagesService = new MessagesService();
37      const threadsService: ThreadsService = new ThreadsService(messagesService);
```

Notice here that we're passing `messagesService` as an argument to the constructor of our `ThreadsService`. Normally we let the Dependency Injection system handle this for us. But in our test, we can provide the dependencies ourselves.

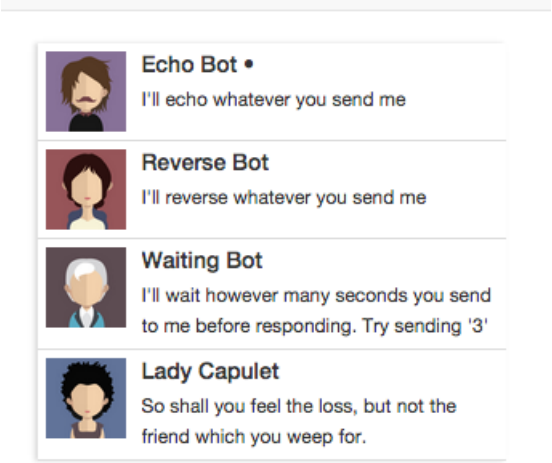
Let's subscribe to `threads` and log out what comes through:

code/rxjs/rxjs-chat/src/app/thread/threads.service.spec.ts

```
37      const threadsService: ThreadsService = new ThreadsService(messagesService);
38
39      threadsService.threads
40        .subscribe( (threadIdx: { [key: string]: Thread }) => {
41          const threads: Thread[] = _.values(threadIdx);
42          const threadNames: string = _.map(threads, (t: Thread) => t.name)
43                                       .join(', ');
44          console.log(`=> threads (${threads.length}): ${threadNames} `);
45        });
46
47      messagesService.addMessage(m1);
48      messagesService.addMessage(m2);
49      messagesService.addMessage(m3);
50
51      // => threads (1): Thread 1
52      // => threads (1): Thread 1
53      // => threads (2): Thread 1, Thread 2
54
55    });
56  });
```

A chronological list of `Thread`s, newest-first (in `orderedthreads`)

`threads` gives us a map which acts as an "index" of our list of threads. But we want the threads view to be ordered according the most recent message.



Time Ordered List of Threads

Let's create a new stream that returns an `Array` of `Threads` ordered by the most recent `Message` time:

We'll start by defining `orderedThreads` as an instance property:

code/rxjs/rxjs-chat/src/app/thread/threads.service.ts

```
14   // `orderedThreads` contains a newest-first chronological list of threads
15   orderedThreads: Observable<Thread[]>;
```

Next, in the `constructor` we'll define `orderedThreads` by subscribing to `threads` and ordered by the most recent message:

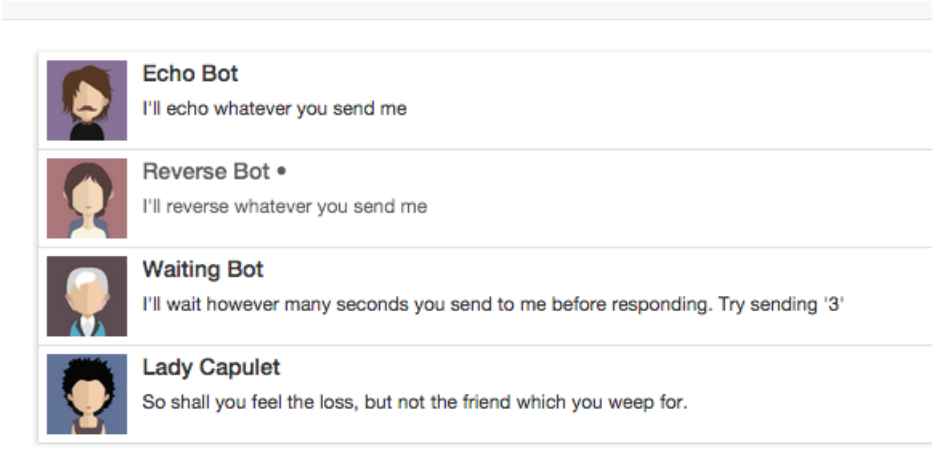
code/rxjs/rxjs-chat/src/app/thread/threads.service.ts

```
45     this.orderedThreads = this.threads
46       .map((threadGroups: { [key: string]: Thread }) => {
47         const threads: Thread[] = _.values(threadGroups);
48         return _.sortBy(threads, (t: Thread) => t.lastMessage.sentAt).reverse();
49       });
```

The currently selected `Thread` (in `currentThread`)

Our application needs to know which `Thread` is the currently selected thread. This lets us know:

1. which thread should be shown in the messages window
2. which thread should be marked as the current thread in the list of threads



The current thread is marked by a dot symbol

Let's create a `BehaviorSubject` that will store the `currentThread`:

code/rxjs/rxjs-chat/src/app/thread/threads.service.ts

```
17   // `currentThread` contains the currently selected thread
18   currentThread: Subject<Thread> =
19     new BehaviorSubject<Thread>(new Thread());
```

Notice that we're issuing an empty `Thread` as the default value. We don't need to configure the `currentThread` any further.

Setting the Current `Thread`

To set the current thread we can have clients either

1. submit new threads via `next` directly or
2. add a helper method to do it.

Let's define a helper method `setCurrentThread` that we can use to set the next thread:

code/rxjs/rxjs-chat/src/app/thread/threads.service.ts

```
70   setCurrentThread(newThread: Thread): void {
71     this.currentThread.next(newThread);
72   }
```

Marking the Current `Thread` as Read

We want to keep track of the number of unread messages. If we switch to a new `Thread` then we want to mark all of the `Messages` in that `Thread` as read. We have the parts we need to do this:

1. The `messagesService.makeThreadAsRead` accepts a `Thread` and then will mark all `Messages` in that `Threaad` as read
2. Our `currentThread` emits a single `Thread` that represents the current `Thread`

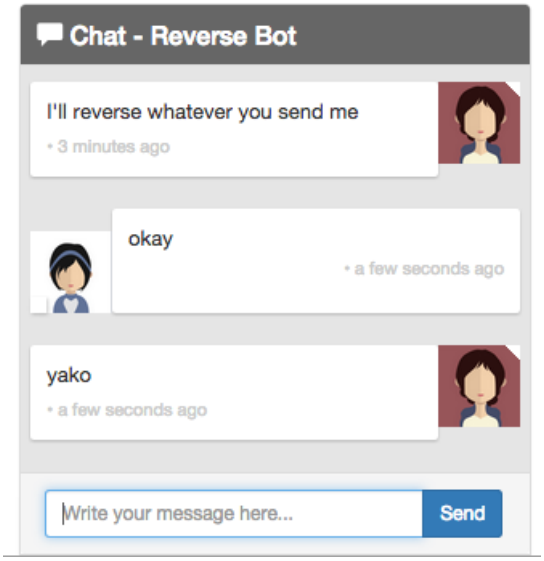
So all we need to do is hook them together:

code/rxjs/rxjs-chat/src/app/thread/threads.service.ts

```
67     this.currentThread.subscribe(this.messagesService.markThreadAsRead);
```

The list of `Message`s for the currently selected `Thread` (in `currentThreadMessages`)

Now that we have the currently selected thread, we need to make sure we can show the list of `Messages` in that `Thread`.



The current list of messages is for the Reverse Bot

Implementing this is a little bit more complicated than it may seem at the surface. Say we implemented it like this:

```
var theCurrentThread: Thread;

this.currentThread.subscribe((thread: Thread) => {
  theCurrentThread = thread;
})

this.currentThreadMessages.map(
  (mesages: Message[]) => {
    return _.filter(messages,
      (message: Message) => {
        return message.thread.id == theCurrentThread.id;
      })
  })
```

What's wrong with this approach? Well, if the `currentThread` changes, `currentThreadMessages` won't know about it and so we'll have an outdated list of `currentThreadMessages`!

What if we reversed it, and stored the current list of messages in a variable and subscribed to the changing of `currentThread`? We'd have the same problem only this time we would know when the thread changes but not when a new message came in.

How can we solve this problem?

It turns out, RxJS has a set of operators that we can use to **combine multiple streams**. In this case we want to say "if *either* `currentThread` **or** `messagesService.messages` changes, then we want to emit something." For this we use the `combineLatest`[95] operator.

code/rxjs/rxjs-chat/src/app/thread/threads.service.ts

```
    this.currentThreadMessages = this.currentThread
      .combineLatest(messagesService.messages,
                     (currentThread: Thread, messages: Message[]) => {
```

When we're combining two streams one or the other will arrive first and there's no guarantee that we'll have a value on both streams, so we need to check to make sure we have what we need otherwise we'll just return an empty list.

Now that we have both the current thread and messages, we can filter out just the messages we're interested in:

[95] https://github.com/Reactive-Extensions/RxJS/blob/master/doc/api/core/operators/combinelatestproto.md

code/rxjs/rxjs-chat/src/app/thread/threads.service.ts

```
51      this.currentThreadMessages = this.currentThread
52        .combineLatest(messagesService.messages,
53                       (currentThread: Thread, messages: Message[]) => {
54          if (currentThread && messages.length > 0) {
55            return _.chain(messages)
56              .filter((message: Message) =>
57                      (message.thread.id === currentThread.id))
```

One other detail, since we're already looking that the messages for the current thread, this is a convenient area to mark these messages as read.

code/rxjs/rxjs-chat/src/app/thread/threads.service.ts

```
55            return _.chain(messages)
56              .filter((message: Message) =>
57                      (message.thread.id === currentThread.id))
58              .map((message: Message) => {
59                message.isRead = true;
60                return message; })
61              .value();
```

Whether or not we should be marking messages as read here is debatable. The biggest drawback is that we're mutating objects in what is, essentially, a "read" thread. i.e. this is a read operation with a side effect, which is generally a Bad Idea. That said, in this application the `currentThreadMessages` only applies to the `currentThread` and the `currentThread` should always have its messages marked as read. That said, the "read with side-effects" is not a pattern I recommend in general.

Putting it together, here's what `currentThreadMessages` looks like:

code/rxjs/rxjs-chat/src/app/thread/threads.service.ts

```
51      this.currentThreadMessages = this.currentThread
52        .combineLatest(messagesService.messages,
53                       (currentThread: Thread, messages: Message[]) => {
54          if (currentThread && messages.length > 0) {
55            return _.chain(messages)
56              .filter((message: Message) =>
57                      (message.thread.id === currentThread.id))
58              .map((message: Message) => {
```

```
              message.isRead = true;
              return message; })
            .value();
        } else {
          return [];
        }
      });
```

Our Completed `ThreadsService`

Here's what our `ThreadService` looks like:

code/rxjs/rxjs-chat/src/app/thread/threads.service.ts

```
import { Injectable } from '@angular/core';
import { Subject, BehaviorSubject, Observable } from 'rxjs';
import { Thread } from './thread.model';
import { Message } from '../message/message.model';
import { MessagesService } from '../message/messages.service';
import * as _ from 'lodash';

@Injectable()
export class ThreadsService {

  // `threads` is a observable that contains the most up to date list of threads
  threads: Observable<{ [key: string]: Thread }>;

  // `orderedThreads` contains a newest-first chronological list of threads
  orderedThreads: Observable<Thread[]>;

  // `currentThread` contains the currently selected thread
  currentThread: Subject<Thread> =
    new BehaviorSubject<Thread>(new Thread());

  // `currentThreadMessages` contains the set of messages for the currently
  // selected thread
  currentThreadMessages: Observable<Message[]>;

  constructor(public messagesService: MessagesService) {

    this.threads = messagesService.messages
      .map( (messages: Message[]) => {
        const threads: {[key: string]: Thread} = {};
```

```
30          // Store the message's thread in our accumulator `threads`
31          messages.map((message: Message) => {
32            threads[message.thread.id] = threads[message.thread.id] ||
33              message.thread;
34
35            // Cache the most recent message for each thread
36            const messagesThread: Thread = threads[message.thread.id];
37            if (!messagesThread.lastMessage ||
38                messagesThread.lastMessage.sentAt < message.sentAt) {
39              messagesThread.lastMessage = message;
40            }
41          });
42          return threads;
43        });
44
45      this.orderedThreads = this.threads
46        .map((threadGroups: { [key: string]: Thread }) => {
47          const threads: Thread[] = _.values(threadGroups);
48          return _.sortBy(threads, (t: Thread) => t.lastMessage.sentAt).reverse();
49        });
50
51      this.currentThreadMessages = this.currentThread
52        .combineLatest(messagesService.messages,
53                       (currentThread: Thread, messages: Message[]) => {
54          if (currentThread && messages.length > 0) {
55            return _.chain(messages)
56              .filter((message: Message) =>
57                      (message.thread.id === currentThread.id))
58              .map((message: Message) => {
59                message.isRead = true;
60                return message; })
61              .value();
62          } else {
63            return [];
64          }
65        });
66
67      this.currentThread.subscribe(this.messagesService.markThreadAsRead);
68    }
69
70    setCurrentThread(newThread: Thread): void {
71      this.currentThread.next(newThread);
```

```
72    }
73
74  }
75
76  export const threadsServiceInjectables: Array<any> = [
77    ThreadsService
78  ];
```

Data Model Summary

Our data model and services are complete! Now we have everything we need now to start hooking it up to our view components! In the next chapter we'll build out our 3 major components to render and interact with these streams.