Data Architecture with Observables - Part 2: View Components

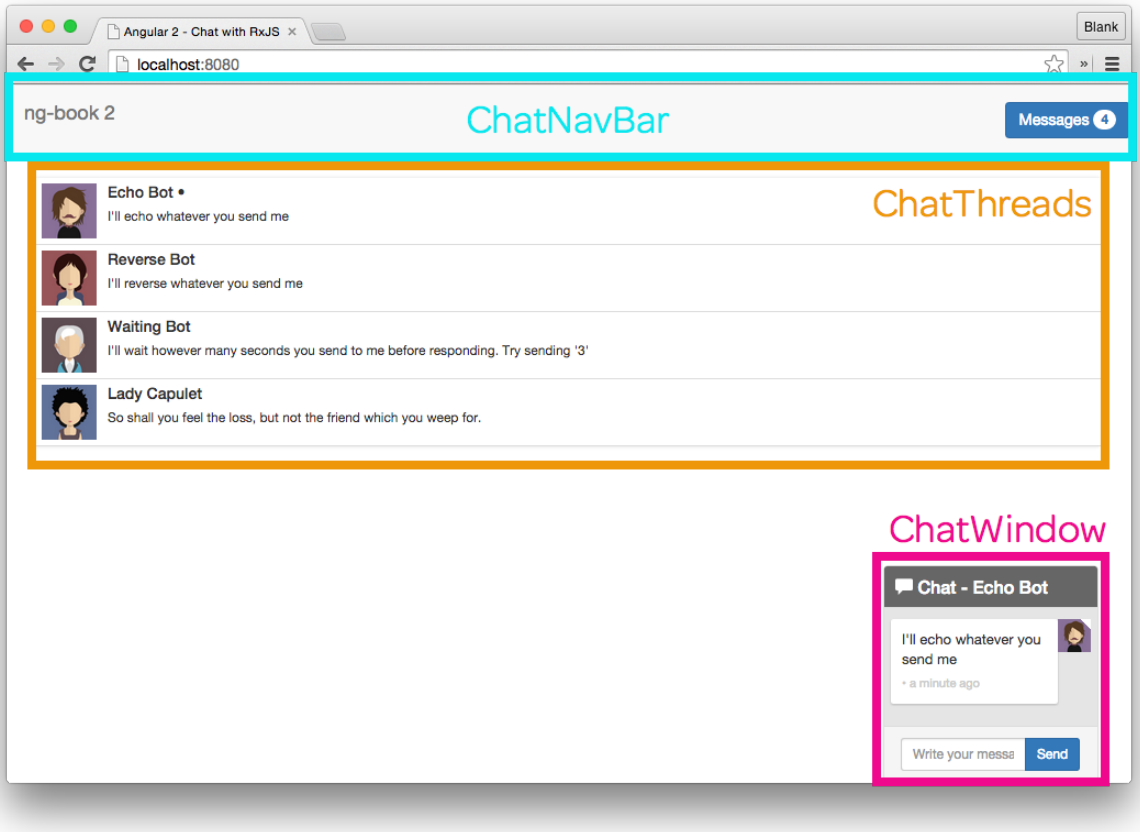
Building Our Views: The `AppComponent` Top-Level Component

Let's turn our attention to our app and implement our view components.

For the sake of clarity and space, in the following sections I'll be leaving out some `import` statements, CSS, and a few other similar things lines of code. If you're curious about each line of those details, open up the sample code because it contains everything we need to run this app.

The first thing we're going to do is create our top-level component `chat-app`

As we talked about earlier, 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

Here's what our top-level component looks like in code:

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

```
import { Component, Inject } from '@angular/core';
import { ChatExampleData } from './data/chat-example-data';

import { UsersService } from './user/users.service';
import { ThreadsService } from './thread/threads.service';
import { MessagesService } from './message/messages.service';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
    constructor(public messagesService: MessagesService,
                public threadsService: ThreadsService,
                public usersService: UsersService) {
    ChatExampleData.init(messagesService, threadsService, usersService);
  }
}
```

and the template:

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

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

In this chapter we are adding some style using the CSS framework Bootstrap[96]

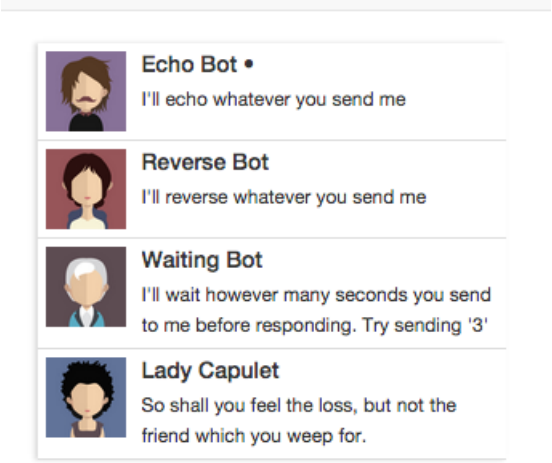
Take a look at the `constructor`. Here we're injecting our three services: the `MessagesService`, `ThreadsService`, and `UserService`. We're using those services to initialize our example data.

If you're interested in the example data you can find it in `code/rxjs/rxjs-chat/src/app/data/chat-example-data.ts`.

We'll build our `chat-page` in a moment, but first let's build our thread list in the `ChatThreadsComponent`.

[96] http://getbootstrap.com

The `ChatThreadsComponent`



Time Ordered List of Threads

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

```
import {
  Component,
  OnInit,
  Inject
} from '@angular/core';
import { Observable } from 'rxjs';
import { Thread } from '../thread/thread.model';
import { ThreadsService } from './../thread/threads.service';

@Component({
  selector: 'chat-threads',
  templateUrl: './chat-threads.component.html',
  styleUrls: ['./chat-threads.component.css']
})
export class ChatThreadsComponent {
  threads: Observable<any>;

  constructor(public threadsService: ThreadsService) {
    this.threads = threadsService.orderedThreads;
  }
}
```

Here we're injecting `ThreadsService` and then we're keeping a reference to the `orderedThreads` .

`ChatThreadsComponent` template

Lastly, let's look at the `template` and its configuration:

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

```
<!-- conversations -->
<div class="row">
  <div class="conversation-wrap">

    <chat-thread
         *ngFor="let thread of threads | async"
         [thread]="thread">
    </chat-thread>

  </div>
</div>
```

There's three things to look at here: `NgFor` with the `async` pipe, the `ChangeDetectionStrategy` and `ChatThreadComponent`.

The `ChatThreadComponent` directive component (which matches `chat-thread` in the markup) will show the view for the `Threads`. We'll define that in a moment.

The `NgFor` iterates over our `threads`, and passes the input `[thread]` to our `ChatThreadComponent` directive. But you probably notice something new in our `*ngFor`: the pipe to `async`.

`async` is implemented by `AsyncPipe` and it lets us use an RxJS `Observable` here in our view. What's great about `async` is that it lets us use our async observable as if it was a sync collection. This is super convenient and really cool.

On this component we specify a custom `changeDetection`. Angular has a flexible and efficient change detection system. One of the benefits is that if we have a component which has immutable or observable bindings, then we're able to give the change detection system hints that will make our application run very efficiently.

We talk more about various change-detection strategies in the Advanced Components Chapter

In this case, instead of watching for changes on an array of `Threads`, Angular will subscribe for changes to the `threads` observable - and trigger an update when a new event is emitted.

The Single `ChatThreadComponent`

Let's look at our `ChatThreadComponent`. This is the component that will be used to display a **single thread**. Starting with the `@Component`:

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

```
import {
  Component,
  OnInit,
  Input,
  Output,
  EventEmitter
} from '@angular/core';
import { Observable } from 'rxjs';
import { ThreadsService } from './../thread/threads.service';
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;
  selected = false;

  constructor(public threadsService: ThreadsService) {
  }

  ngOnInit(): void {
    this.threadsService.currentThread
      .subscribe( (currentThread: Thread) => {
        this.selected = currentThread &&
          this.thread &&
          (currentThread.id === this.thread.id);
      });
  }

  clicked(event: any): void {
    this.threadsService.setCurrentThread(this.thread);
    event.preventDefault();
  }
}
```

We'll come back and look at the `template` in a minute, but first let's look at the component definition controller.

`ChatThreadComponent` Controller and `ngOnInit`

Notice that we're implementing a new interface here: `OnInit`. Angular components can declare that they listen for certain lifecycle events. We talk more about lifecycle events here in the Advanced Components chapter.

In this case, because we declared that we implement `OnInit`, the method `ngOnInit` will be called on our component after the component has been checked for changes the first time.

A key reason we will use `ngOnInit` is because **our `thread` property won't be available in the `constructor`**.

Above you can see that in `ngOnInit` we subscribe to `threadsService.currentThread` and if the `currentThread` matches the `thread` property of this component, we set `selected` to `true` (conversely, if the `Thread` doesn't match, we set `selected` to false).

We also setup an event handler `clicked`. This is how we handle selecting the current thread. In our `template` (below), we will bind `clicked()` to clicking on the thread view. If we receive `clicked()` then we tell the `threadsService` we want to set the current thread to the `Thread` of this component.

`ChatThreadComponent` template

Here's the code for our template:

code/rxjs/rxjs-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.lastMessage.text}}</small>
  </div>
  <a (click)="clicked($event)" class="div-link">Select</a>
</div>
```

Notice we've got some straight-forward bindings like `{{thread.avatarSrc}}`, `{{thread.name}}`, and `{{thread.lastMessage.text}}`.

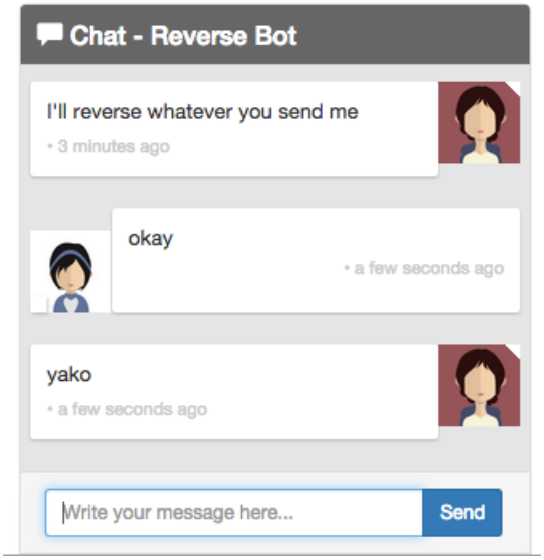
We've got an `*ngIf` which will show the `&bull;` symbol only if this is the selected thread.

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.

The `ChatWindowComponent`

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



The Chat Window

We start by defining our `@Component`:

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

```
17 @Component({
18   selector: 'chat-window',
19   templateUrl: './chat-window.component.html',
20   styleUrls: ['./chat-window.component.css'],
21   changeDetection: ChangeDetectionStrategy.OnPush
```

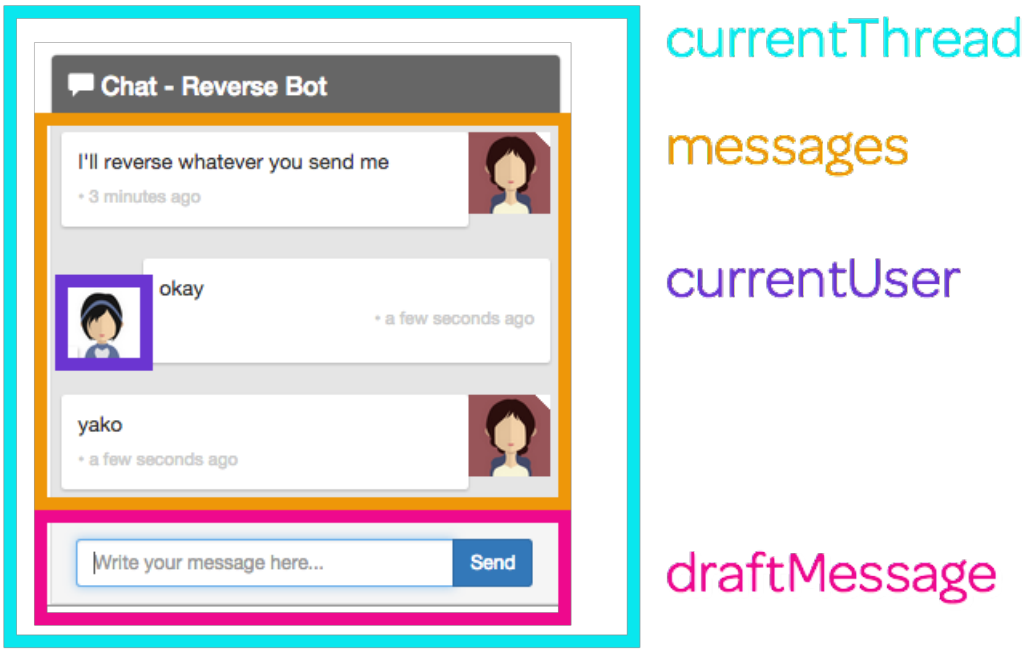
`ChatWindowComponent` Class Properties

Our `ChatWindowComponent` class has four properties :

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

```
23 export class ChatWindowComponent implements OnInit {
24   messages: Observable<any>;
25   currentThread: Thread;
26   draftMessage: Message;
27   currentUser: User;
```

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



Chat Window Properties

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

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

```
29   constructor(public messagesService: MessagesService,
30               public threadsService: ThreadsService,
31               public UsersService: UsersService,
32               public el: ElementRef) {
33   }
```

The first three are our services. The fourth, `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.

Remember: by using `public messagesService: MessagesService` in the constructor, we are not only injecting the `MessagesService` but setting up an instance variable that we can use later in our class via `this.messagesService`

ChatWindowComponent ngOnInit

We're going to put the initialization of this component in `ngOnInit`. The main thing we're going to be doing here is setting up the subscriptions on our observables which will then change our component properties.

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

```
35   ngOnInit(): void {
36     this.messages = this.threadsService.currentThreadMessages;
37
38     this.draftMessage = new Message();
```

First, we'll save the `currentThreadMessages` into `messages`. Next we create an empty `Message` for the default `draftMessage`.

When we send a new message we need to make sure that `Message` stores a reference to the sending `Thread`. The sending thread is always going to be the current thread, so let's store a reference to the currently selected thread:

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

```
40     this.threadsService.currentThread.subscribe(
41       (thread: Thread) => {
42         this.currentThread = thread;
43       });
```

We also want new messages to be sent from the current user, so let's do the same with `currentUser`:

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

```
45     this.UsersService.currentUser
46       .subscribe(
47         (user: User) => {
48           this.currentUser = user;
49         });
```

ChatWindowComponent sendMessage

Since we're talking about it, let's implement a `sendMessage` function that will send a new message:

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

```
65   sendMessage(): void {
66     const m: Message = this.draftMessage;
67     m.author = this.currentUser;
68     m.thread = this.currentThread;
69     m.isRead = true;
70     this.messagesService.addMessage(m);
71     this.draftMessage = new Message();
72   }
```

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.

Notice here that we're not updating the `draftMessage` text. That's because we're going to bind the value of the messages text in the view in a few minutes.

After we've updated the `draftMessage` properties we send it off to the `messagesService` and then **create a new `Message`** and set that new `Message` to `this.draftMessage`. We do this to make sure we don't mutate an already sent message.

`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 that event:

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

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

`ChatWindowComponent scrollToBottom`

When we send a message, or when a new message comes in, we want to scroll to the bottom of the chat window. To do that, we're going to set the `scrollTop` property of our host element:

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

```
74   scrollToBottom(): void {
75     const scrollPane: any = this.el
76       .nativeElement.querySelector('.msg-container-base');
77     scrollPane.scrollTop = scrollPane.scrollHeight;
78   }
```

Now that we have a function that will scroll to the bottom, we have to make sure that we call it at the right time. Back in `ngOnInit` let's subscribe to the list of `currentThreadMessages` and scroll to the bottom any time we get a new message:

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

```
50     this.messages
51       .subscribe(
52         (messages: Array<Message>) => {
53           setTimeout(() => {
54             this.scrollToBottom();
55           });
56         });
57   }
```

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` template

The opening of our `template` should look familiar, we start by defining some markup and the panel header:

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

```
<div class="chat-window-container">
  <div class="chat-window">
    <div class="panel-container">
      <div class="panel panel-default">

        <div class="panel-heading top-bar">
          <div class="panel-title-container">
            <h3 class="panel-title">
              <span class="glyphicon glyphicon-comment"></span>
              Chat - {{currentThread.name}}
            </h3>
          </div>
          <div class="panel-buttons-container">
            <!-- you could put minimize or close buttons here -->
          </div>
        </div>
```

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

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

```
        <div class="panel-body msg-container-base">
          <chat-message
          *ngFor="let message of messages | async"
          [message]="message">
        </chat-message>
      </div>
```

Lastly we have the message input box and closing tags :

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

```
      <div class="panel-footer">
        <div class="input-group">
          <input type="text"
          class="chat-input"
          placeholder="Write your message here..."
          (keydown.enter)="onEnter($event)"
          [(ngModel)]="draftMessage.text" />
          <span class="input-group-btn">
```

```
32              <button class="btn-chat"
33              (click)="onEnter($event)"
34              >Send</button>
35            </span>
36          </div>
37        </div>
38
39      </div>
40    </div>
41  </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, on the `input` tag above, 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)`.

Using ngModel

As we've talked about before, Angular doesn't have a general model for two-way binding. 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.

code/rxjs/rxjs-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" />
```

Clicking "Send"

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

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

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

The Entire `ChatWindowComponent`

We broke that up into a lot tiny pieces. So that we can get a view of the whole thing, here's the code listing for the entire `ChatWindowComponent`:

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

```
1  import {
2    Component,
3    Inject,
4    ElementRef,
5    OnInit,
6    ChangeDetectionStrategy
7  } from '@angular/core';
8  import { Observable } from 'rxjs';
9
10 import { User } from '../user/user.model';
11 import { UsersService } from '../user/users.service';
12 import { Thread } from '../thread/thread.model';
13 import { ThreadsService } from '../thread/threads.service';
14 import { Message } from '../message/message.model';
15 import { MessagesService } from '../message/messages.service';
16
17 @Component({
18   selector: 'chat-window',
19   templateUrl: './chat-window.component.html',
20   styleUrls: ['./chat-window.component.css'],
21   changeDetection: ChangeDetectionStrategy.OnPush
22 })
23 export class ChatWindowComponent implements OnInit {
24   messages: Observable<any>;
25   currentThread: Thread;
26   draftMessage: Message;
27   currentUser: User;
28
```

```
29    constructor(public messagesService: MessagesService,
30                public threadsService: ThreadsService,
31                public UsersService: UsersService,
32                public el: ElementRef) {
33    }
34
35    ngOnInit(): void {
36      this.messages = this.threadsService.currentThreadMessages;
37
38      this.draftMessage = new Message();
39
40      this.threadsService.currentThread.subscribe(
41        (thread: Thread) => {
42          this.currentThread = thread;
43        });
44
45      this.UsersService.currentUser
46        .subscribe(
47          (user: User) => {
48            this.currentUser = user;
49          });
50
51      this.messages
52        .subscribe(
53          (messages: Array<Message>) => {
54            setTimeout(() => {
55              this.scrollToBottom();
56            });
57          });
58    }
59
60    onEnter(event: any): void {
61      this.sendMessage();
62      event.preventDefault();
63    }
64
65    sendMessage(): void {
66      const m: Message = this.draftMessage;
67      m.author = this.currentUser;
68      m.thread = this.currentThread;
69      m.isRead = true;
70      this.messagesService.addMessage(m);
```

```
    this.draftMessage = new Message();
  }

  scrollToBottom(): void {
    const scrollPane: any = this.el
      .nativeElement.querySelector('.msg-container-base');
    scrollPane.scrollTop = scrollPane.scrollHeight;
  }
}
```

and template:

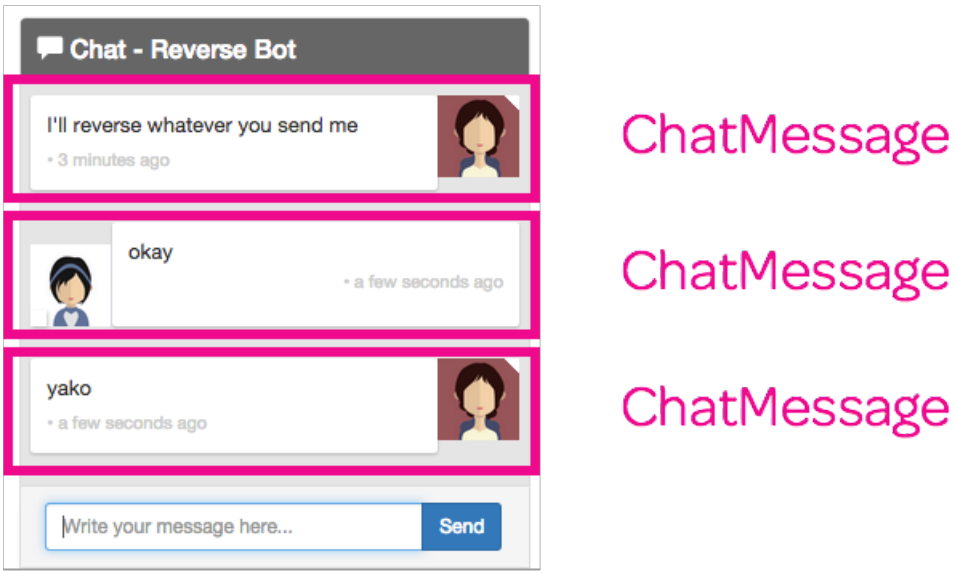
code/rxjs/rxjs-chat/src/app/chat-window/chat-window.component.html

```
<div class="chat-window-container">
  <div class="chat-window">
    <div class="panel-container">
      <div class="panel panel-default">

        <div class="panel-heading top-bar">
          <div class="panel-title-container">
            <h3 class="panel-title">
              <span class="glyphicon glyphicon-comment"></span>
              Chat - {{currentThread.name}}
            </h3>
          </div>
          <div class="panel-buttons-container">
            <!-- you could put minimize or close buttons here -->
          </div>
        </div>

        <div class="panel-body msg-container-base">
          <chat-message
          *ngFor="let message of messages | async"
          [message]="message">
        </chat-message>
      </div>

      <div class="panel-footer">
        <div class="input-group">
          <input type="text"
          class="chat-input"
          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 `ChatMessageComponent`

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`.

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

We start by defining the `@Component`

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

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

import { UsersService } from './../user/users.service';
import { ThreadsService } from './../thread/threads.service';
import { MessagesService } from './../message/messages.service';

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

@Component({
  selector: 'chat-message',
  templateUrl: './chat-message.component.html',
  styleUrls: ['./chat-message.component.css']
})
export class ChatMessageComponent implements OnInit {
  @Input() message: Message;
  currentUser: User;
  incoming: boolean;

  constructor(public UsersService: UsersService) {
  }

  ngOnInit(): void {
    this.UsersService.currentUser
      .subscribe(
        (user: User) => {
          this.currentUser = user;
          if (this.message.author && user) {
            this.incoming = this.message.author.id !== user.id;
          }
```

```
37          });
38    }
39  }
```

The `ChatMessageComponent` `template`

In our `template` we have two interesting ideas:

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

First, here's the code:

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

```
1  <div class="msg-container"
2       [ngClass]="{'base-sent': !incoming, 'base-receive': incoming}">
3
4    <div class="avatar"
5         *ngIf="!incoming">
6      <img src="{{message.author.avatarSrc}}">
7    </div>
8
9    <div class="messages"
10     [ngClass]="{'msg-sent': !incoming, 'msg-receive': incoming}">
11     <p>{{message.text}}</p>
12     <p class="time">{{message.sender}} • {{message.sentAt | fromNow}}</p>
13   </div>
14
15   <div class="avatar"
16        *ngIf="incoming">
17     <img src="{{message.author.avatarSrc}}">
18   </div>
19 </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`[97] library. If you'd like to learn about creating your own custom pipes read the source of the `FromNowPipe` in `code/rxjs/rxjs-chat/src/app/pipes/from-now.pipe.ts`

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

[97] http://momentjs.com/

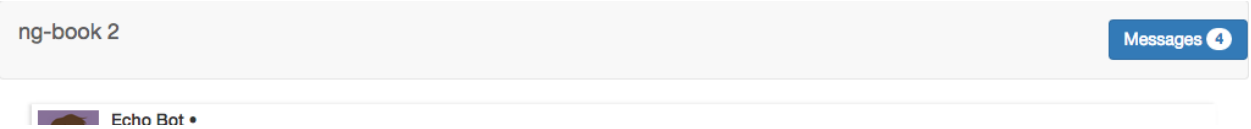
```
1     [ngClass]="{'msg-sent': !incoming, 'msg-receive': incoming}"
```

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.

The `ChatNavBarComponent`

The last component we have to talk about is 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.

The `ChatNavBarComponent @Component`

The only thing the `ChatNavBarComponent` controller needs to keep track of is the `unreadMessagesCount`. This is slightly more complicated than it seems on the surface.

The most straightforward way would be to simply listen to `messagesService.messages` and sum the number of `Messages` where `isRead` is false. This works fine for all messages outside of the current thread. However new messages in the current thread aren't guaranteed to be marked as read by the time `messages` emits new values.

The safest way to handle this is to combine the `messages` and `currentThread` streams and make sure we don't count any messages that are part of the current thread.

We do this using the `combineLatest` operator, which we've already used earlier in the chapter:

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

```
import {
  Component,
  Inject,
  OnInit
} from '@angular/core';
import * as _ from 'lodash';

import { ThreadsService } from './../thread/threads.service';
import { MessagesService } from './../message/messages.service';

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

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

  constructor(public messagesService: MessagesService,
              public threadsService: ThreadsService) {
  }

  ngOnInit(): void {
    this.messagesService.messages
      .combineLatest(
        this.threadsService.currentThread,
        (messages: Message[], currentThread: Thread) =>
          [currentThread, messages] )

      .subscribe(([currentThread, messages]: [Thread, Message[]]) => {
        this.unreadMessagesCount =
          _.reduce(
            messages,
            (sum: number, m: Message) => {
              const messageIsInCurrentThread: boolean = m.thread &&
                currentThread &&
                (currentThread.id === m.thread.id);
              // note: in a "real" app you should also exclude
```

```
42                // messages that were authored by the current user b/c they've
43                // already been "read"
44                if (m && !m.isRead && !messageIsInCurrentThread) {
45                  sum = sum + 1;
46                }
47                return sum;
48              },
49              0);
50        });
51    }
52  }
```

If you're not an expert in TypeScript you might find the above syntax a little bit hard to parse. In the `combineLatest` callback function we're returning an array with `currentThread` and `messages` as its two elements.

Then we `subscribe` to that stream and we're *destructuring* those objects in the function call. Next we `reduce` over the `messages` and count the number of messages that are unread and not in the current thread.

The `ChatNavBarComponent` template

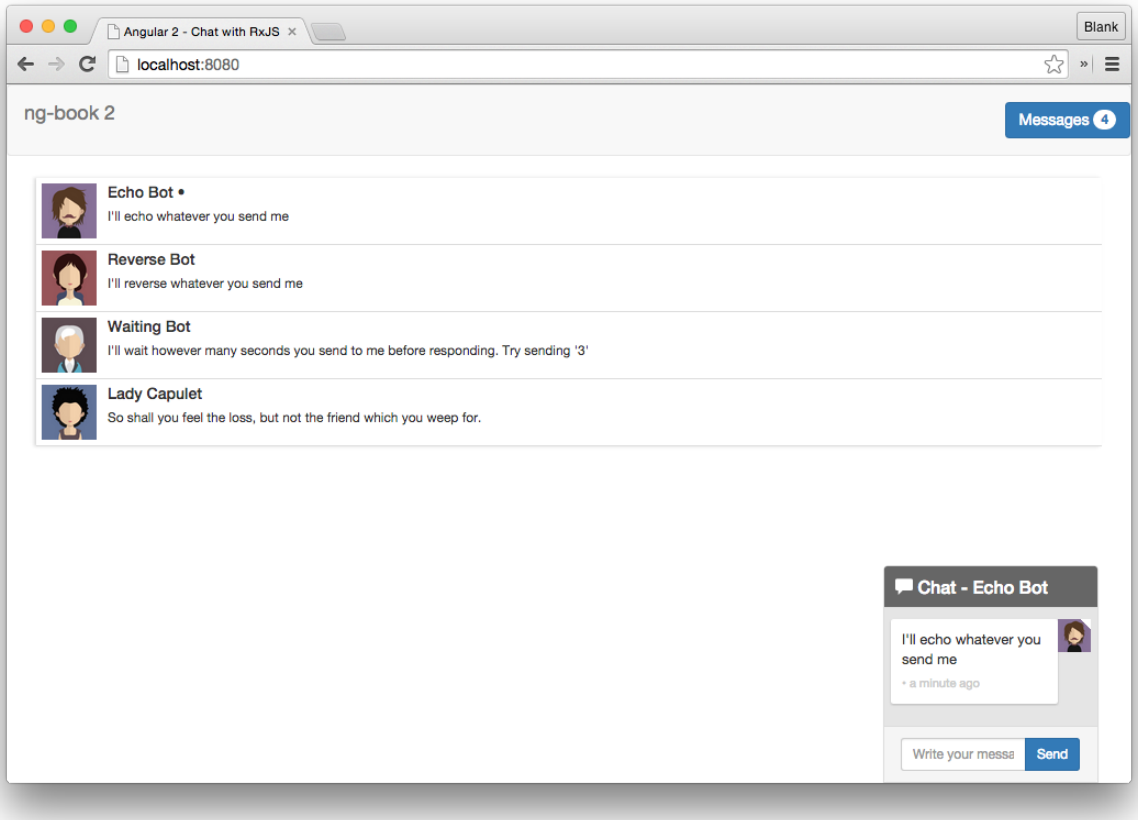
In our view, the only thing we have left to do is display our `unreadMessagesCount`:

code/rxjs/rxjs-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>
```

Summary

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



Completed Chat Application

If you checkout `code/rxjs/rxjs-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. Here's a code excerpt from the Reverse Bot:

```
let rev: User = new User("Reverse Bot", require("images/avatars/female-avatar-4.\
png"));
let tRev: Thread = new Thread("tRev", rev.name, rev.avatarSrc);
```

code/rxjs/rxjs-chat/src/app/data/chat-example-data.ts

```
    messagesService.messagesForThreadUser(tRev, rev)
      .forEach( (message: Message): void => {
        messagesService.addMessage(
          new Message({
            author: rev,
            text: message.text.split('').reverse().join(''),
            thread: tRev
          })
        );
      },
```

Above you can see that we've subscribed to the messages for the "Reverse Bot" by using `messagesForThreadUser`. Try writing a few bots of your own.

Next Steps

Some ways to improve this chat app would be to become stronger at RxJS and then hook it up to an actual API. We'll talk about how to make API requests in the HTTP Chapter. For now, enjoy your fancy chat application!