Extracted from:

Build Reactive Websites with RxJS

Master Observables and Wrangle Events

The Pragmatic Bookshelf
Raleigh, North Carolina

The Pragmatic Programmers

Build Reactive Websites with RxJS

Master Observables and Wrangle Events

Randall Koutnik

edited by Brian MacDonald

Build Reactive Websites with RxJS

Master Observables and Wrangle Events

Randall Koutnik

The Pragmatic Bookshelf

Raleigh, North Carolina

The team that produced this book includes:

Publisher: Andy Hunt
VP of Operations: Janet Furlow
Managing Editor: Susan Conant
Development Editor: Brian MacDonald
Copy Editor: Paula Robertson
Indexing: Potomac Indexing, LLC
Layout: Gilson Graphics

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

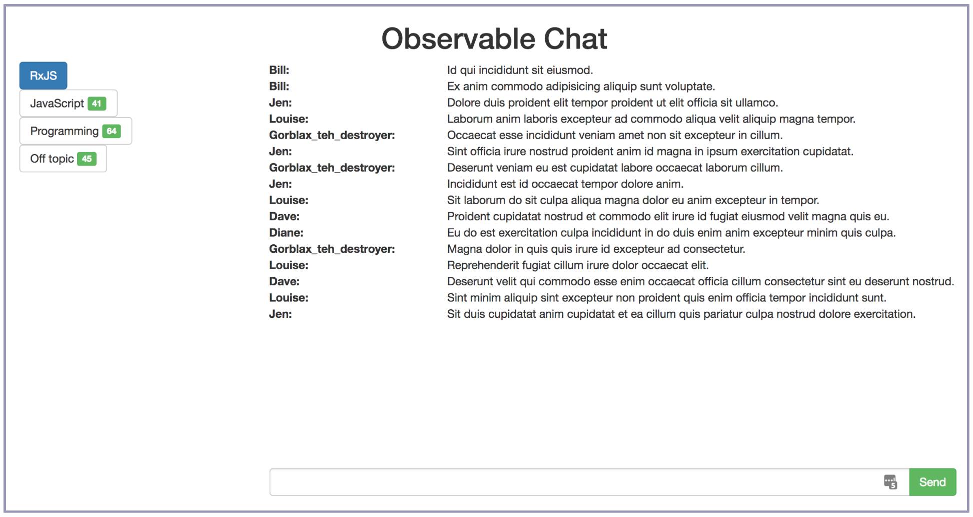
ISBN-13: 978-1-68050-295-4
Book version: P1.0—December 2018

Building a Chat Room

Chat systems are a favorite practice realm for programmers. You need to build out a two-way communications backbone, as well as a reactive UI on top of it. Subjects can help you enormously, handling both the real-time communication and data storage for use by components initialized after load. An example of this would be a subject recording the chat history in the background so that when the user opens that room, they'll see the most recent messages without needing an additional request.

This excercise is a capstone project for everything you've learned so far in this book. The goal is to connect many reactive streams to build an entire application. Extraneous functions to perform the raw DOM manipulation around displaying messages and modals are provided for you in chatlib.ts. While you're encouraged to take a look at these functions, they will not be discussed further, so we can keep the focus on learning RxJS.

When you're finished with this section, the chat application will have a login system, multiple rooms, and a chat history for each room.



Bill:	Id qui incididunt sit eiusmod.
Bill:	Ex anim commodo adipisicing aliquip sunt voluptate.
Jen:	Dolore duis proident elit tempor proident ut elit officia sit ullamco.
Louise:	Laborum anim laboris excepteur ad commodo aliqua velit aliquip magna tempor.
Gorblax_teh_destroyer:	Occaecat esse incididunt veniam amet non sit excepteur in cillum.
Jen:	Sint officia irure nostrud proident anim id magna in ipsum exercitation cupidatat.
Gorblax_teh_destroyer:	Deserunt veniam eu est cupidatat labore occaecat laborum cillum.
Jen:	Incididunt est id occaecat tempor dolore anim.
Louise:	Sit laborum do sit culpa aliqua magna dolor eu anim excepteur in tempor.
Dave:	Proident cupidatat nostrud et commodo elit irure id fugiat eiusmod velit magna quis eu.
Diane:	Eu do est exercitation culpa incididunt in do duis enim anim excepteur minim quis culpa.
Gorblax_teh_destroyer:	Magna dolor in quis quis irure id excepteur ad consectetur.
Louise:	Reprehenderit fugiat cillum irure dolor occaecat elit.
Dave:	Deserunt velit qui commodo esse enim occaecat officia cillum consectetur sint eu deserunt nostrud.
Louise:	Sint minim aliquip sint excepteur non proident quis enim officia tempor incididunt sunt.
Jen:	Sit duis cupidatat anim cupidatat et ea cillum quis pariatur culpa nostrud dolore exercitation.

This chat system is centered around a single subject hooked up to the chat websocket. You'll use the webSocket constructor for connecting and managing this connection to the server. Add the following snippet to the reader.ts file in the chapter directory. The chatStream$ variable will serve as the central source of information for your chat system.

vanilla/multiplexingObservables/reader-complete.ts
```
interface ChatRoom {
  name: string;
}
interface ChatMessage {
  room: ChatRoom;
}
let wsUrl = 'ws://localhost:3000/api/multiplexingObservables/chat-ws';
let chatStream$ = webSocket<ChatMessage>(wsUrl);
```

This chat application has four parts, and each one hooks into chatStream$ in a unique way. The first section handles the user providing a username as a rudimentary form of authentication.

Logging in

The first thing you will see when you load the page is a modal that asks for a username. In this section, you add code that allows the user to enter a username, connect to the server to log in, and close the modal to display the rest of the page. Let's add some code that listens in on the events from that modal to trigger login. To make it easy on the user, you'll create two different observables, one listening for the Enter key, and the other listening for a click on the Login button. All the code cares about at this point is that the user has filled in their name and wishes to submit it.

At this point, we map to the value of the input box (ignoring any value, the important thing is that an event happened), filter out empty strings, and display a handy loading spinner to indicate to the user that the backend is working hard on getting their chat ready. Finally, the subscription calls authenticateUser—a function you'll create in the next snippet.

vanilla/multiplexingObservables/reader-complete.ts
```
interface User {
  rooms: ChatRoom[];
}
let userSubject$ = new AsyncSubject<User>();
function authenticateUser(username) {
  let user$ = ajax(
        'http://localhost:3000/api/multiplexingObservables/chat/user/'
        + username)
    .pipe(
      map(data => data.response)
    );

  user$.subscribe(userSubject$);
}
```

Next, let's use an AJAX observable to tell the backend about the newly connected user. The AjaxObservable sends a request to the backend, and the AsyncSubject listens in, storing the resulting value for the rest of the application to use upon request.

vanilla/multiplexingObservables/reader-complete.ts

```
merge(
  fromEvent(loginBtn, 'click'),
  fromEvent<any>(loginInput, 'keypress')
  .pipe(
    // Ignore all keys except for enter
    filter(e => e.keyCode === 13)
  )
)
.pipe(
  map(() => loginInput.value),
  filter(Boolean),
  tap(showLoadingSpinner)
)
.subscribe(authenticateUser);
```

Joe asks:

What's with the Boolean Filter?

To review: The filter method expects to take a function that checks the latest value in the stream and returns true or false. filter only passes on a value if the function returns true.

JavaScript provides constructor functions for all of the primitives in the language, including booleans. The Boolean constructor takes any value, returning true if the value is truthy, and false otherwise. Sound familiar? .filter(Boolean) can be used as a shortcut for .filter(value => !!value) and carries a clearer intent for what you intend to do.

Now the code knows when the user has chosen a username and now it needs to close the modal and show the rest of the app. To do so, add a subscription to the user subject, calling the provided closeLoginModal function when the user request finishes and providing data about the current state of the chat room.

vanilla/multiplexingObservables/reader-complete.ts

```
userSubject$
.subscribe(closeLoginModal);
```

Now, you should be able to load the page, enter a username in the modal, and wait for the backend to respond with data about the current state of the chat. After the backend responds, nothing is listening in to render anything to the page. It's time to implement the code around viewing and switching chat rooms.

Rendering and Switching Rooms

After the user has logged in, they will want to see all of the rooms available to them and switch between them. To accomplish this, once the login modal has closed, start listening in for any new messages that come across the websocket. While it's possible to not keep any history and only show the latest messages, you can use the RxJS ReplaySubject to track room history. A ReplaySubject records the last n events and plays them back to every new subscriber. In this example, we'll create a new ReplaySubject for every chat channel and create a new subscription whenever the user switches rooms.

vanilla/multiplexingObservables/reader-complete.ts
```
function makeRoomStream(roomName) {
  let roomStream$ = new ReplaySubject(100);
  chatStream$
  .pipe(
    filter(msg => msg.room.name === roomName)
  )
  .subscribe(roomStream$);
  return roomStream$;
}
```

When the user authenticates, the server replies with the list of rooms the user is currently in. The room section needs to listen in on that, render the list of room buttons to the page, create room streams using the function above, and trigger an event loading the user into the first room on the list by default. Here, you'll use three separate subscribe functions to keep things compartmentalized:

vanilla/multiplexingObservables/reader-complete.ts
```
let roomStreams = {};
userSubject$
.subscribe(userObj => {
  userObj.rooms.forEach(room =>
    roomStreams[room.name] = makeRoomStream(room.name)
  );
});
userSubject$
  .subscribe(userObj => renderRoomButtons(userObj.rooms));
userSubject$
  .subscribe(userObj => roomLoads$.next(userObj.rooms[0].name));
```

For that code to work, you need to track when the user clicks one of the room buttons on the left, indicating they'd like to switch to a new room. A separate subject is created to track room loads so that we can trigger a room load from an event emitted by userSubject$. There's also a check to see whether the user clicked directly on the unread number, in which case, we pass on the parent element.

vanilla/multiplexingObservables/reader-complete.ts
```
let roomClicks$ = fromEvent<any>(roomList, 'click')
.pipe(
  // If they click on the number, pass on the button
  map(event => {
    if (event.target.tagName === 'SPAN') {
      return event.target.parentNode;
    }
    return event.target;
  }),
  // Remove unread number from room name text
  map(element => element.innerText.replace(/\s\d+$/, ''))
);

let roomLoads$ = new Subject();
roomClicks$.subscribe(roomLoads$);
```

Finally, now that you're tracking which room is active, it's time to start listening in on the streams and showing new messages on the page. The roomLoads$ stream listens for new room loads, updates the DOM classes on the buttons, switches to the new room stream through switchMap, and writes each event from the stream to the page as a message (writeMessageToPage and setActiveRoom are provided for you in chatLib.ts). Remember that each stream in roomStreams is a ReplaySubject, so as soon as switchMap subscribes to the subject, the last 100 messages are passed down the chain.

vanilla/multiplexingObservables/reader-complete.ts
```
roomLoads$
.pipe(
  tap(setActiveRoom),
  switchMap(room => roomStreams[room])
)
.subscribe(writeMessageToPage);
```

Now that you've completed this section of the application, a list of rooms to join appears on the left, and each room starts to display messages from other users. When a user clicks the button to switch to a new room, the chat history that's been collected so far is shown. While this is starting to look like a functional chat room, one critical feature is missing: the user still can't send a message to a chat room. Time to fix that.

Sending Messages

Now that the user can see the current rooms and the messages sent to them, it's time to let them send messages of their own. Compared to the two sections in the chat room so far, sending messages is fairly simple. It starts with the same technique as the login modal, using merge to listen for either a selection of the Send button or a press of the Enter key. Next, the stream plucks out the value of the message box, ensures the value is not an empty string, and resets the message box.

The following snippet introduces a new operator you haven't seen before: withLatestFrom. The stream in this snippet needs to send a new chat message (entered by the user) to the server and needs to annotate it with the user's name and current room so the server knows who sent the message and where it was sent.

Previously, you used combineLatest whenever you needed to combine the most recent value from multiple streams. combineLatest comes with a catch, though—it emits a new value when any of the streams emits a value. We don't want to send a new chat message when the user switches to a new room. Instead, withLatestFrom only emits new values when the observable stream that it's passed into through pipe emits a value. You can also add an optional projection function to combine the latest values from all of the streams.

vanilla/multiplexingObservables/reader-complete.ts

```
merge(
  fromEvent<any>(sendBtn, 'click'),
  fromEvent<any>(msgBox, 'keypress')
  .pipe(
    // Only emit event when enter key is pressed
    filter(e => e.keyCode === 13)
  )
)
.pipe(
  map(() => msgBox.value),
  filter(Boolean),
  tap(() => msgBox.value = ''),
  withLatestFrom(
    roomLoads$,
    userSubject$,
    (message, room, user) => ({ message, room, user })
  )
)
.subscribe(val => chatStream$.next(<any>val));
```

Finally, the chat room is feature complete. Users can log in, read incoming messages in all rooms in the system, and send messages of their own. Time to add a final flourish: let's display how many unread messages are waiting in each room.

Displaying Unread Notifications

While not strictly needed for a chat room, it can be interesting to see how many new messages have shown up in a room that the user doesn't have directly loaded. This feature is concerned with two streams: new messages and room loads. The tricky part is that, while we want to listen in to multiple streams and store state inside the observable stream (using merge and scan), we also want to perform different actions depending on which stream emits a new value. To make this simple, call map on the streams as they're passed into the merge constructor, so each new event tells us what type of event it is:

vanilla/multiplexingObservables/reader-complete.ts

```
merge(
  chatStream$.pipe(
    map(msg => ({type: 'message', room: msg.room.name}))
  ),
  roomLoads$.pipe(
    map(room => ({type: 'roomLoad', room: room}))
  )
)
```

Now that the stream is annotated, you can use scan to carry the state of all unread messages. Here the state contains two properties: rooms, an object storing the number of unread messages per room, and activeRoom, the most recently loaded room. Inside scan, we check to see what type of event has been emitted.

In case the event is a room load, the state is updated to record the new active room and set the number of unread messages in that room to 0. In the case that the websocket has sent a new message, scan first checks to see whether the message was sent to the current room. If it was, the current state is returned unmodified. Otherwise, we make sure that the current state has a record for the room in question (adding a new entry if this is the first time scan has seen this room, to allow for a dynamic room list). Finally, the room record is incremented.

vanilla/multiplexingObservables/reader-complete.ts

```
.pipe(
  scan((unread, event: any) => {
    // new message in room
    if (event.type === 'roomLoad') {
      unread.activeRoom = event.room;
      unread.rooms[event.room] = 0;
    } else if (event.type === 'message') {
      if (event.room === unread.activeRoom) {
        return unread;
      }
      if (!unread.rooms[event.room]) {
        unread.rooms[event.room] = 0;
      }
      unread.rooms[event.room]++;
    }
  return unread;
}, {
  rooms: {},
  activeRoom: ''
}),
```

The last step has two map operators to convert the state from scan into something easier to loop over, and the subscribe call passes each room object to setUnread, a function from chatlib.ts that updates the text in the room buttons.

vanilla/multiplexingObservables/reader-complete.ts

```
  map(unread => unread.rooms),
  map(rooms =>
    Object.keys(rooms)
    .map(key => ({
      roomName: key,
      number: rooms[key]
    }))
  )
)
.subscribe(roomArr => {
  roomArr.forEach(setUnread);
});
```

With that, your chat room is complete. If you're looking for a bit more of a challenge, try to update the code so that the user can change their name. Right now, this codebase assumes that the user can't change their username after entering it in the initial modal. Imagine if userSubject$ was an unbounded stream, adding an AJAX call for each username change. How would you change things to make them more flexible? Start with the pattern you used to track unread rooms, since that brought in two unbounded streams.