MENU

Integrating Angular 2 and SignalR - Part 2 of 2

20 MARCH 2016 on angular2, SignalR, WebAPI

Update - May 17, 2016

I have now updated the code examples to use the 2.0.0-rc.1

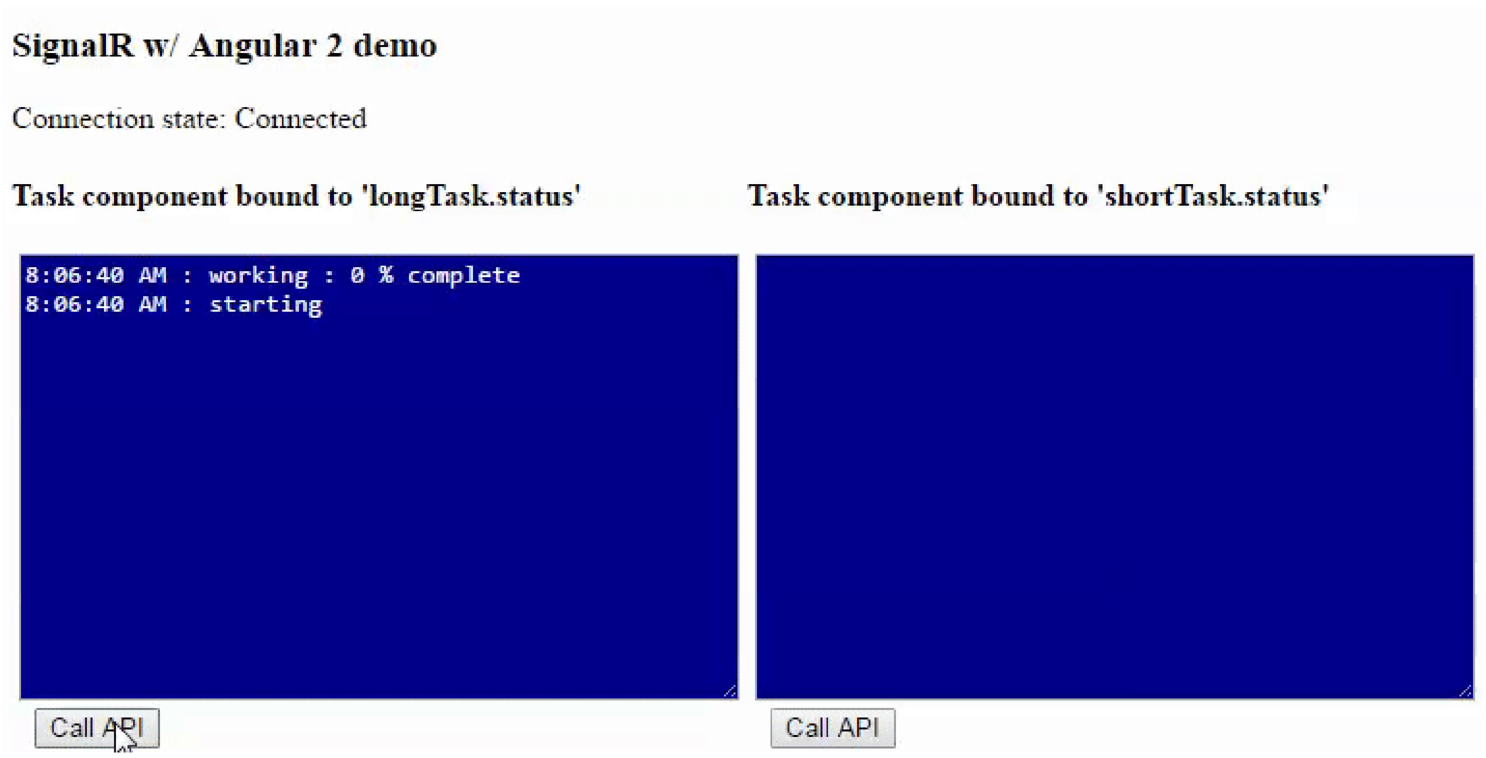
version of Angular 2. This in turn pulls in RxJS 5.0.0-beta.6, which I'm hoping addresses some issues a couple people were having in the comments. This also uses the new SystemJS config approach shown by the Angular 2 live example.

This post is part of a two-part series on integrating Angular 2 and SignalR. If you haven't read the first post, please check it out:

- Integrating Angular 2 and SignalR - Part 1 of 2
- Integrating Angular 2 and SignalR - Part 2 of 2 (this post)

In the first post of this series we walked through how to build a relatively simple API that used SignalR to provide real-time updates to clients calling API methods. In this article we'll tackle the Angular 2 side of things. All of the code shown is available in my github repo.

To set the stage we're going to build a simple Angular 2 app that can invoke the long and short tasks of the API, but also receive the status updates in real-time using SignalR. Visually the application will look like this:


SignalR w/ Angular 2 demo

Connection state: Connected

Task component bound to 'longTask.status'

```
8:06:40 AM : working : 0 % complete
8:06:40 AM : starting
```

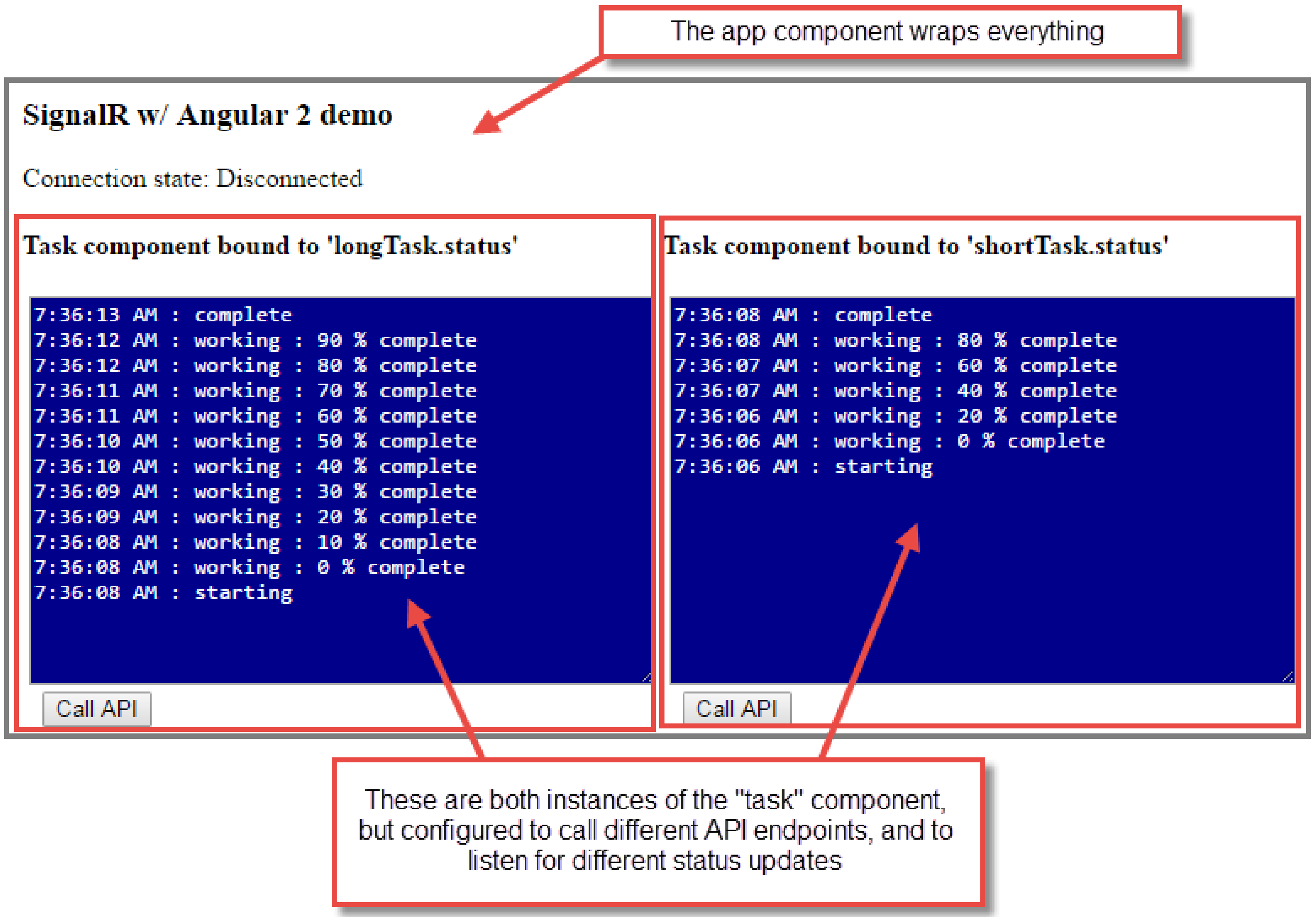
Task component bound to 'shortTask.status'

We have a couple components here that let us call the API using a button, and then any status updates related to that task are shown in the textareas. We also have some top-level information shown about the state of the SignalR connection.

The Angular 2 Components

To build this app I tried to keep things as simple as possible. There are only two components used:

1. The top-level "app" component
2. A "task" component


The app component wraps everything
SignalR w/ Angular 2 demo
Connection state: Disconnected
Task component bound to 'longTask.status'
Task component bound to 'shortTask.status'
Call API
Call API
These are both instances of the "task" component, but configured to call different API endpoints, and to listen for different status updates

```
7:36:13 AM : complete
7:36:12 AM : working : 90 % complete
7:36:12 AM : working : 80 % complete
7:36:11 AM : working : 70 % complete
7:36:11 AM : working : 60 % complete
7:36:10 AM : working : 50 % complete
7:36:10 AM : working : 40 % complete
7:36:09 AM : working : 30 % complete
7:36:09 AM : working : 20 % complete
7:36:08 AM : working : 10 % complete
7:36:08 AM : working : 0 % complete
7:36:08 AM : starting
```

```
7:36:08 AM : complete
7:36:08 AM : working : 80 % complete
7:36:07 AM : working : 60 % complete
7:36:07 AM : working : 40 % complete
7:36:06 AM : working : 20 % complete
7:36:06 AM : working : 0 % complete
7:36:06 AM : starting
```

Now, these rely on a single service that exposes the SignalR functionality, but we'll come back to that later on in this article.

The TaskComponent class

Here's what our task component looks like:

```
import {Component, OnInit, Input} from "@angular/core";
import {Http, Response} from "@angular/http";

import {ChannelService, ChannelEvent} from "./services/channel.s

class StatusEvent {
    State: string;
    PercentComplete: number;
}
```

```
@Component({
    selector: 'task',
    template: `
        <div>
            <h4>Task component bound to '{{eventName}}'</h4>
        </div>

        <div class="commands">
            <textarea
                class="console"
                cols="50"
                rows="15"
                disabled
                [value]="messages"></textarea>

            <div class="commands__input">
                <button (click)="callApi()">Call API</button>
            </div>
        </div>
    `
})
export class TaskComponent implements OnInit {
    @Input() eventName: string;
    @Input() apiUrl: string;

    messages = "";

    private channel = "tasks";

    constructor(
        private http: Http,
        private channelService: ChannelService
    ) {

    }

    ngOnInit() {
        // Get an observable for events emitted on this channel
        //
        this.channelService.sub(this.channel).subscribe(
            (x: ChannelEvent) => {
                switch (x.Name) {
                    case this.eventName: { this.appendStatusUpda
                }
            },
            (error: any) => {
                console.warn("Attempt to join channel failed!",
            }
        )
    }


    private appendStatusUpdate(ev: ChannelEvent): void {
        // Just prepend this to the messages string shown in the
        //
        let date = new Date();
        switch (ev.Data.State) {
            case "starting": {
                this.messages = `${date.toLocaleTimeString()} :
                break;
            }

            case "complete": {
                this.messages = `${date.toLocaleTimeString()} :
                break;
            }

            default: {
                this.messages = `${date.toLocaleTimeString()} :
            }
        }
    }

    callApi() {
        this.http.get(this.apiUrl)
            .map((res: Response) => res.json())
            .subscribe((message: string) => { console.log(messag
    }
}
```

We can see it has a pretty vanilla template, has two `@Input()` parameters, and is injected with both the `Http` and `ChannelService` services. The input parameters drive the behavior of this component like so:

- `eventName` - Determines what events are shown in the status window
- `apiUrl` - Determines what endpoint is called when the button is clicked

We also have code within `ngOnInit()` to react to events emitted by the channel service, but we'll come back to that later.

The AppComponent

The main application component is pretty simple, and is used mostly to start the SignalR connection, and to of course contain the child components. The code looks like this:

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

import {ChannelService, ConnectionState} from "./services/channe
import {TaskComponent} from "./task.component";

@Component({
    selector: 'my-app',
    template: `
        <div>
            <h3>SignalR w/ Angular 2 demo</h3>
        </div>

        <div>
            <span>Connection state: {{connectionState$ | async}}
        </div>

        <div class="flex-row">
            <task class="flex"
                [eventName]="'longTask.status'"
                [apiUrl]="'http://localhost:9123/tasks/long'"></
            <task class="flex"
                [eventName]="'shortTask.status'"
                [apiUrl]="'http://localhost:9123/tasks/short'"><
        </div>

    `,
    directives: [TaskComponent]
})
export class AppComponent implements OnInit {

    // An internal "copy" of the connection state stream used be
    //  we want to map the values of the original stream. If we
    //  need to do that then we could use the service's observab
    //  right in the template.
    //
    connectionState$: Observable<string>;

    constructor(
        private channelService: ChannelService
    ) {

        // Let's wire up to the signalr observables
        //
        this.connectionState$ = this.channelService.connectionSt
            .map((state: ConnectionState) => { return Connectior

        this.channelService.error$.subscribe(
            (error: any) => { console.warn(error); },
            (error: any) => { console.error("errors$ error", err
        );

        // Wire up a handler for the starting$ observable to log
        //  success/fail result
        //
        this.channelService.starting$.subscribe(
            () => { console.log("signalr service has been starte
            () => { console.warn("signalr service failed to star
        );
    }

    ngOnInit() {
        // Start the connection up!
        //
        console.log("Starting the channel service");

        this.channelService.start();
    }
}
```

We can see in the template we are using two instances of the `TaskComponent`, and providing them different inputs. We can also see in the template that we're subscribing to a stream that provides updates about the connection state of the channel service. We'll describe this more in a bit.

Outside of the template this component mostly subscribes to observables provided by the channel service, and inside of `ngOnInit()` it starts the channel service.

Bootstrapping

With Angular 2 you need to bootstrap the application, and here's the code in this example:

```
import {bootstrap} from '@angular/platform-browser-dynamic';
import {provide} from "@angular/core";
import {HTTP_PROVIDERS} from "@angular/http";

import "rxjs/add/operator/map";
import {AppComponent} from './app.component';

import {ChannelService, ChannelConfig, SignalrWindow} from "./se


let channelConfig = new ChannelConfig();
channelConfig.url = "http://localhost:9123/signalr";
channelConfig.hubName = "EventHub";

bootstrap(AppComponent, [
    HTTP_PROVIDERS,
    ChannelService,
    provide(SignalrWindow, {useValue: window}),
    provide("channel.config", { useValue: channelConfig })
]);
```

Here we're pulling in the services we'll need, but also providing two token-based providers. One provides a type called `SignalrWindow` that is used to call the SignalR methods within our ChannelService class without relying on global variables. The other drives the behavior of the ChannelService itself.

The ChannelService

ok, so here's the meat of this example. The ChannelService is meant to wrap the internals of how SignalR works, and expose it using the channel/event model discussed in the first part of this series.

Publicly it exposes the following items:

- `starting$` – An observable that emits an event when the connection is first established
- `connectionState$` – An observable for changes in the state of

the SignalR connection

- `error$` – An observable of errors that can occur within SignalR
- `start()` – A function to start the connection
- `sub(channel: string)` – A function to subscribe to events on a specific channel. This returns an observable that the caller can subscribe to
- `publish(ev: ChannelEvent)` – A function to push out events on a given channel (this isn't used in this particular demo though)

`<caveat>` I am in no way an expert in observables, and am trying really hard to better understand them. If you see something really dumb please tell me in the comments! `</caveat>`

With that, here's the code for the ChannelService:

```
import {Injectable, Inject} from "@angular/core";
import {Subject} from "rxjs/Subject";
import {Observable} from "rxjs/Observable";

/**
 * When SignalR runs it will add functions to the global $ varia
 * that you use to create connections to the hub. However, in th
 * class we won't want to depend on any global variables, so thi
 * class provides an abstraction away from using $ directly in h
 */
export class SignalrWindow extends Window {
    $: any;
}

export enum ConnectionState {
    Connecting = 1,
    Connected = 2,
    Reconnecting = 3,
    Disconnected = 4
}

export class ChannelConfig {
    url: string;
    hubName: string;
    channel: string;
}

export class ChannelEvent {
    Name: string;
    ChannelName: string;
    Timestamp: Date;
    Data: any;
    Json: string;

    constructor() {
        this.Timestamp = new Date();
    }
}

class ChannelSubject {
    channel: string;
    subject: Subject<ChannelEvent>;
}

/**
 * ChannelService is a wrapper around the functionality that Sig
 * provides to expose the ideas of channels and events. With thi
 * you can subscribe to specific channels (or groups in signalr 
 * use observables to react to specific events sent out on those
 */
@Injectable()
export class ChannelService {

    /**
     * starting$ is an observable available to know if the signa
     * connection is ready or not. On a successful connection th
     * stream will emit a value.
     */
    starting$: Observable<any>;

    /**
     * connectionState$ provides the current state of the underl
     * connection as an observable stream.
     */
    connectionState$: Observable<ConnectionState>;

    /**
     * error$ provides a stream of any error messages that occur
     * SignalR connection
     */
    error$: Observable<string>;

    // These are used to feed the public observables
    //
    private connectionStateSubject = new Subject<ConnectionState
    private startingSubject = new Subject<any>();
    private errorSubject = new Subject<any>();

    // These are used to track the internal SignalR state
    //
    private hubConnection: any;
    private hubProxy: any;

    // An internal array to track what channel subscriptions exi
    //
    private subjects = new Array<ChannelSubject>();

    constructor(
        @Inject(SignalrWindow) private window: SignalrWindow,
        @Inject("channel.config") private channelConfig: Channel
    ) {
        if (this.window.$ === undefined || this.window.$.hubConn
            throw new Error("The variable '$' or the .hubConnect
        }

        // Set up our observables
        //
        this.connectionState$ = this.connectionStateSubject.asOb
        this.error$ = this.errorSubject.asObservable();
        this.starting$ = this.startingSubject.asObservable();

        this.hubConnection = this.window.$.hubConnection();
        this.hubConnection.url = channelConfig.url;
        this.hubProxy = this.hubConnection.createHubProxy(channe

        // Define handlers for the connection state events
        //
        this.hubConnection.stateChanged((state: any) => {
            let newState = ConnectionState.Connecting;

            switch (state.newState) {
                case this.window.$.signalR.connectionState.conne
                    newState = ConnectionState.Connecting;
                    break;
                case this.window.$.signalR.connectionState.conne
                    newState = ConnectionState.Connected;
                    break;
                case this.window.$.signalR.connectionState.recon
                    newState = ConnectionState.Reconnecting;
                    break;
                case this.window.$.signalR.connectionState.disco
                    newState = ConnectionState.Disconnected;
                    break;
            }

            // Push the new state on our subject
            //
            this.connectionStateSubject.next(newState);
        });

        // Define handlers for any errors
        //
        this.hubConnection.error((error: any) => {
            // Push the error on our subject
            //
            this.errorSubject.next(error);
        });
        this.hubProxy.on("onEvent", (channel: string, ev: Channe
            //console.log(`onEvent - ${channel} channel`, ev);

            // This method acts like a broker for incoming messa
            //  check the interal array of subjects to see if or
            //  for the channel this came in on, and then emit t
            //  on it. Otherwise we ignore the message.
            //
            let channelSub = this.subjects.find((x: ChannelSubje
                return x.channel === channel;
            }) as ChannelSubject;

            // If we found a subject then emit the event on it
            //
            if (channelSub !== undefined) {
                return channelSub.subject.next(ev);
            }
        });

    }

    /**
     * Start the SignalR connection. The starting$ stream will e
     * event if the connection is established, otherwise it will
     * error.
     */
    start(): void {
        // Now we only want the connection started once, so we h
        //  starting$ observable that clients can subscribe to k
        //  if the startup sequence is done.
        //
        // If we just mapped the start() promise to an observabl
        //  a client subscried to it the start sequence would be
        //  again since it's a cold observable.
        //
        this.hubConnection.start()
            .done(() => {
                this.startingSubject.next();
            })
            .fail((error: any) => {
```

```
            this.startingSubject.error(error);
        });
}

/**
 * Get an observable that will contain the data associated w
 * channel
 * */
sub(channel: string): Observable<ChannelEvent> {

    // Try to find an observable that we already created for
    //  channel
    //
    let channelSub = this.subjects.find((x: ChannelSubject)
        return x.channel === channel;
    }) as ChannelSubject;

    // If we already have one for this event, then just retu
    //
    if (channelSub !== undefined) {
        console.log(`Found existing observable for ${channel
        return channelSub.subject.asObservable();
    }

    //
    // If we're here then we don't already have the observab
    //  caller, so we need to call the server method to joir
    //  and then create an observable that the caller can us
    //  messages.
    //

    // Now we just create our internal object so we can trac
    //  in case someone else wants it too
    //
    channelSub = new ChannelSubject();
    channelSub.channel = channel;
    channelSub.subject = new Subject<ChannelEvent>();
    this.subjects.push(channelSub);

    // Now SignalR is asynchronous, so we need to ensure the
```

```
        //  established before we call any server methods. So we
        //  the starting$ stream since that won't emit a value u
        //  is ready
        //
        this.starting$.subscribe(() => {
            this.hubProxy.invoke("Subscribe", channel)
                .done(() => {
                    console.log(`Successfully subscribed to ${ch
                })
                .fail((error: any) => {
                    channelSub.subject.error(error);
                });
        },
            (error: any) => {
                channelSub.subject.error(error);
            });

        return channelSub.subject.asObservable();
    }

    // Not quite sure how to handle this (if at all) since there
    //  more than 1 caller subscribed to an observable we create
    //
    // unsubscribe(channel: string): Rx.Observable<any> {
    //     this.observables = this.observables.filter((x: Channe
    //         return x.channel === channel;
    //     });
    // }

    /** publish provides a way for calles to emit events on any
     * production app the server would ensure that only authoriz
     * actually emit the message, but here we're not concerned a
     */
    publish(ev: ChannelEvent): void {
        this.hubProxy.invoke("Publish", ev);
    }

}
```

So there's a fair bit of code, but all this really does is create the SignalR connection, and attempt to map what it provides to observables that users of the service can observe. The `sub()` function works by creating new Subjects (internally) that are used to broker events sent by the SignalR hub.

The idea is that a user of the service just calls `sub("myChannel")`, and they are provided an observable they can subscribe to to get any events sent on that channel. They then need to decide what to do with any events emitted on the observable. With this knowledge we can return to our `TaskComponent`:

```
    ngOnInit() {
        // Get an observable for events emitted on this channel
        //
        this.channelService.sub(this.channel).subscribe(
            (x: ChannelEvent) => {
                switch (x.Name) {
                    case this.eventName: { this.appendStatusUpda
                }
            },
            (error: any) => {
                console.warn("Attempt to join channel failed!",
            }
        )
    }
```

We can now see that in our task component we're calling `sub(this.channel)` and immediately subscribing to the observable we get back. In our subscription we simply check the name of the event provided and append it to the message window if it matches the event the task component was configured for.

Pulling in SignalR itself

SignalR's "magic" is provided by some javascript files provided by Microsoft, so we need to ensure they're included in `index.html`

```
<!DOCTYPE html>
<html>
  <head>
     <!-- omitted for brevity -->

    <!-- Include SignalR (& jQuery since it requires it) -->
    <script src="https://ajax.aspnetcdn.com/ajax/jQuery/jquery-2
    <script src="https://ajax.aspnetcdn.com/ajax/signalr/jquery.

     <!-- omitted for brevity -->
</html>
```

Wrapping up

In this two-part series I explored how to create a simple API that used SignalR to provide real-time updates on long running tasks. I then created an Angular 2 client to call the API methods and process the updates using SignalR.

I am still very new to Angular 2, and in particular how to properly use RxJS, so if you have any feedback or suggestions on how this could be improved please let me know in the comments.

Sam Storie

Read more posts by this author.

Share this post

22 Comments The Blog 1 Login

Recommend 1 Share Sort by Best

Join the discussion…

John • 23 days ago

Very helpful write-up **@Sam Storie**. One question that I'm trying to reason about is why do you call a REST endpoint to invoke SignalR instead of just calling a Hub directly from the client (on the button clicks)? I know this is a sample to illustrate the technologies but I'm trying to understand it from a best practice standpoint. Would the separation be to use the REST API for public consumable APIs and calling SignalR directly for in-app communication that is tightly coupled?

• Reply • Share ›

Sam Storie Mod John • 23 days ago

That's a great question **@John**. In this case I was trying to illustrate an API that provides some functionality that takes a long time to complete. In this case I want to tell the client that I've started the work, so I return immediately from the API call. Now I also want to provide the caller periodic updates on the progress, so this is why the SignalR call is invoked within the API itself.

On my personal website I use this approach to let family members select a bunch of photos they want to download. Once they have them selected they make an API call that returns right away, but kicks off a back-end process to start building a single zip file they can eventually download. As that zip file is being built I push back updates to the client so they can see stuff like "1 of 20 done", "2 of 20 done", etc. If you check out the end of this post you can see this use case in action: https://blog.sstorie.com/how-t...

Another use case is when I want to broadcast updates to data within an API that can be called from many different clients. Most of those clients may not use SignalR at all, but I can inform subscribers anyway if the API talks to the hub itself.

Whether you use an API like this, or rely instead on SignalR exclusively relies on your specific application...I don't know of a best practice that applies in all cases.

Does that help at all?

• Reply • Share ›

John Sam Storie • 23 days ago

Yeah, what you are saying makes sense. I think the implementation of the API call doesn't actually return right away but runs the long running process and then returns once that is complete. That is the part that feels like it should live outside in it's own process (maybe a windows service or something).

• Reply • Share ›

Sam Storie Mod John • 23 days ago

Oh, doh! You're right that the API call doesn't return right away in my example. That's why you don't rush to reply in a Friday afternoon :)

• Reply • Share ›

Testermax • 2 months ago

Hi,
I really like the way you have added the extra layer over SignalR and wrapped the whole thing into "channel/event".
I have played a little with your example, and I figured I wanted to use the task.component in another component (template) than the main AppComponent template.

What I did was that I just cut out the html code from the AppComponent into another component which I`m showing.
But doing so, I can osly render the task, but not firering the subscription to the channel (the message "Sucessfully subscribed to task channel" is never shown in the console.)

BUT, if left the original AppComponent template AND copied it to the other template, everything works.

It seems that the task component need to be generated as the AppComponent loads, for this to work?

• Reply • Share ›

Sam Storie Mod Testermax • 2 months ago

Thanks for the comment Testermax. Without seeing some specific code it's a bit tough to know exactly what's happening, but a couple things come to mind:

1. Are you still specifying the TaskComponent as a directive in your new component (the "copy" of AppComponent)?
2. Is the ChannelService itself starting up and running? If not, then there might be a bigger issue happening.

If possible, throw your code up in a plunkr or something and I can try to take a look to see what might be going on.

• Reply • Share ›

Testermax ➦ Sam Storie • 2 months ago

1: yes
2: yes
I should mention that Im using routing to get to the new component, dont think that really matters. All I really did (or want to acomplish) is to show the task component on a different coponent than the AppCoponent.
I will upload some code when Im at my computer.
Thanx

• Reply • Share ›

Testermax ➦ Testermax • 2 months ago

Hi,
I got it to work, but I had to run the angular app from a different dev server...dont know why.
I just added the task component to another component, and started up the other dev server, and everything works great.

• Reply • Share ›

Sam Storie Mod ➦ Testermax • 2 months ago

That's great to hear. Please let me know if you find anything to improve as you keep working with the code.

• Reply • Share ›

Testermax ➦ Sam Storie • 2 months ago

I will :-)
Thanx for your time.
You do contribute alot to the SW community.

• Reply • Share ›

Michael Martinez • 2 months ago

In the Channel.Service.ts, I'm getting the following compile errors..

1. Property 'asObservable' does not exist on type 'Subject<any>'. --> this.error$ = this.errorSubject.asObservable();
2. Supplied parameters do not match any signature of call target --> this.startingSubject.next();

Any clue?

• Reply • Share ›

Zaidc ➦ Michael Martinez • 2 days ago

What did you tweak?

• Reply • Share ›

Michael Martinez ➦ Michael Martinez • 2 months ago

Whew... with some dependency tweeking.. I got the demo to work.

Thanks ! Very helpful exercise.

• Reply • Share ›

Sam Storie Mod ➦ Michael Martinez • 2 months ago

Good to hear Michael. It's fun when things fall into place, and incredibly frustrating the entire time before that finally happens ;)

• Reply • Share ›

Sam Storie Mod ➦ Michael Martinez • 2 months ago

Hey Michael! That seems strange, but a couple things come to mind:

1. Are you sure you're using RxJS beta 2? It's needed to get access to the .asObservable() function - http://stackoverflow.com/a/352...
2. How are you importing the RxJS code into your code? If you're not using the method I did in the example (which you shouldn't in production apps), you need to specifically import each of the rxjs components you plan to use.

Other than that I'm a bit stumped, sorry!

• Reply • Share ›

Michael Martinez ➦ Sam Storie • 2 months ago

Looks like I only have access to beta.6 in Visual Studio. I'll install BETA 2..

• Reply • Share ›

Michael Martinez ➦ Michael Martinez • 2 months ago

I installed beta.2.. problem #1 is fixed.

Problem #2 is still present. Supplied parameters do not match any signature of call target --> this.startingSubject.next();

• Reply • Share ›

Sam Storie Mod ➦ Michael Martinez • a month ago

Michael, did you ever get past problem #2? I'm going to try to make some time and update the code to use the current RC version of Angular, which should bring in a newer version of rxjs. I wonder if that will eliminate these issues for you.

^ | ˅ • Reply • Share ›

READ THIS NEXT

Self-hosting an Angular 2 website in a windows service

Lately I've really wanted to explore the possibilities of self-hosting an Angular 2 website within a Windows service that...

YOU MIGHT ENJOY

Importing IIS logs into Elasticsearch with Logstash

Logstash is a tool for processing log files that tries to make it easy to import files of varying...