PHIL LEGGETTER

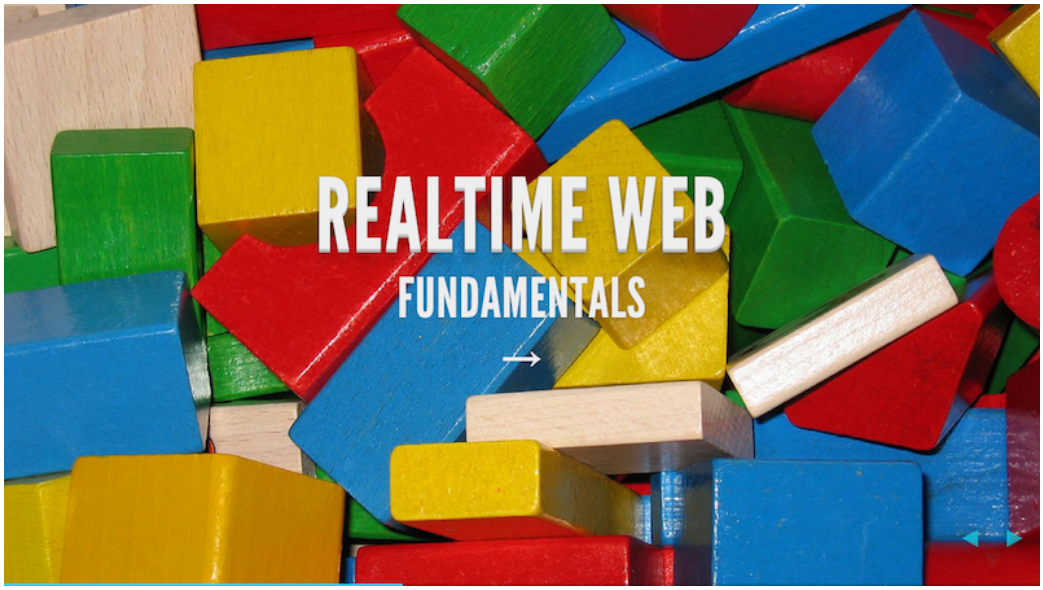
HEAD OF DEVELOPER EVANGELISM AT PUSHER
& REALTIME WEB CONSULTANT

Fundamentals of the Realtime Web & Realtime Web Functionality

31 Oct 2013

In part 1 of this 3-part write-up from my talk at FOWA London 2013 I covered the History, Background, Benefits & Use Cases of Realtime. In this 2nd part I'm going to provide details on some fundaments about realtime web technology and an overview of the basic types of functionality that they offer.

- Part 1: History, Background, Benefits & Use Cases of Realtime
- Part 3: Choosing your Realtime Web App Tech Stack
- The FOWA 2013 Choosing your Realtime Web App Tech Stack video



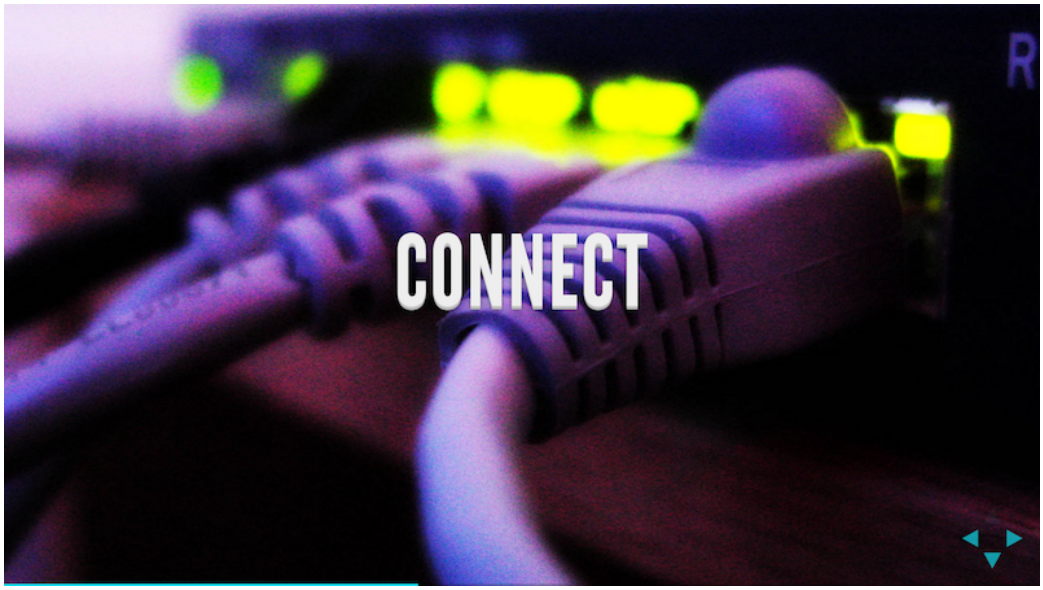
What are the core concepts used in realtime web technologies? What are the fundamental raw technologies? And, is it possible to break the types of functionality up into categories?

My name is **Phil Leggetter**. I'm He
Developer Evangelism at Pusher.

I frequently write articles and give
Realtime Web Technologies and g
technology. Get in touch if you'd l
talk at your event or write an artic
I'm also very interested in develop
experience and productivity, APIs
customer service.

I'm also the co-author of the APres
beginners title "Realtime Web App

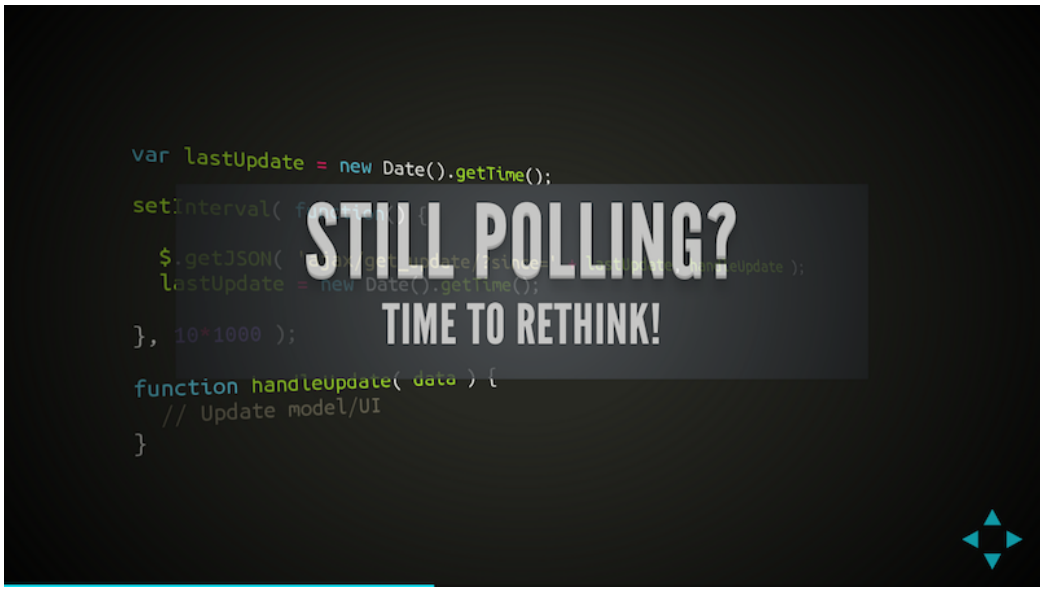
**Realtime Web Apps: With H
WebSocket, PHP, and jQue**



Buy the book I co-write with Jaso via Amazon.com or Amazon

This one's pretty obvious! You need to connect to the source of data in order to receive new information from it. Without that connection there's no way of the new data being pushed to the intended recipients.

Ideally this will be a long-held persistent connection.



So, **STOP POLLING**.

If you check *very infrequently* for new data via a polling strategy that may still be okay. But in general, if you're polling at short intervals you are very likely to:

- Be using resources unnecessarily
- Have code complexity that isn't required

Realtime web solutions are now very solid and efficient so you really should be using them. In fact, if you use a realtime web server instead of polling a standard web server you will:

- Use less client and server resources
- Reduce the complexity of your code

Here's the server code that is executed when new data is available and updates the database:

```
<?php
$update = $_POST['update'];
$data = json_decode($update, true);

$query = sprintf("INSERT INTO updates (key, value) VALUES ('%s', '%s')",
   mysql_real_escape_string($data['key'], $data['value']));

$result = mysql_query($query);

if($result) {
 header("HTTP/1.1 201 Created")
}
else { header("HTTP/1.1 418 I'm a teapot"); }
```

Here's some JavaScript that polls the server every 5 seconds to see if any new data is available:

```
var lastUpdate = new Date();
setInterval(function() {
 $.getJSON('ajax/get_update.php?since=' + lastUpdate, function(data) {
  var items = [];

  $.each(data, function(key, val) {
   items.push('&lt;li id="' + key + '"&gt;' + val + '&lt;/li&gt;');
  });

  $('#my_list').prepend(items);
 });
 lastUpdate = new Date();
}, 5*1000);
```

Here's the server code that handles that polling request:

```
<?php // get_update.php
$since = $_GET['since']; // assume format is correct

$query = sprintf("SELECT updates WHERE created > '%s'",
   mysql_real_escape_string($since));

$result = mysql_query($query); // assume no errors

$rows = array();
while($r = mysql_fetch_assoc($result)) {
 $rows[] = $r;
```

```
}
return json_encode($rows);
```

Now, let's look at a usage scenario:

- Site average of 10,000 users online at any one time
- Over 1 Hour, with a 5 second polling interval
- Pages load + resources estimate (HTML, CSS, JS, Images) = 50,000
- Poll requests per minute = (60 / 5) = 12
- Poll requests per hour = (12 * 60) = 720
- Poll requests site-wide per hour = (360 * 10,000) = 7,200,000.
- **Total: 7.2 Million requests**

Get scaling!

By using a realtime web technology you have dedicated and specialised resource to maintain those 10,000 persistent connections.

And the code would look something like this:

The JavaScript:

```
var connection = new Connection( 'realtime.example.com' );
var channel = connection.subscribe( 'my-channel' );
channel.bind( 'my-event', function( data ) {
  var items = [];
  $.each( data, function( key, val ) {
    items.push( '<li id="' + key + '">' + val + '</li>' );
  } );

  $( '#my_list' ).prepend( items );
});
```

On the server we can completely remove the code that handles the polling request. All we need to do is update the code after we've updated the database. This additional code publishes the new data to all interested recipients:

```
<?php
require('Realtime.php');

$update = $_POST['update'];
$data = json_decode($update, true);

$query = sprintf("INSERT INTO updates (key, value) VALUES ('%s', '%s')",
    mysql_real_escape_string($data['key'], $data['value']));

$result = mysql_query($query);
```

```
 if($result) {
  $realtime = new Realtime();
  $realtime->trigger('my-channel', 'my-event', $data);

  header("HTTP/1.1 201 Created")
 }
 else { header("HTTP/1.1 418 I'm a teapot"); }
```

Note: this code hasn't been checked - there may be a few errors. It's here to demonstrate a point

Here the call to trigger may push the data onto a message queue to be processed by the dedicated realtime infrastructure, or it may be to make a call to a hosted service to handle the distribution of that data.

There is much less code in the realtime scenario and it's also less complex.

In this scenario we have:

- Site average of 10,000 users online at any one time
- Over 1 Hour, **no polling**
- Pages load + resources estimate (HTML, CSS, JS, Images) = 50,000
- **That's it! Our standard web servers handle 50,000 requests**
- During the hour our realtime infrastructure maintains 10,000 persistent connections

Many developers have understood the benefits of the timely delivery of data, the business value, the resource savings and the reduced complexity that realtime could offer (reduced complexity is a relatively recent thing).

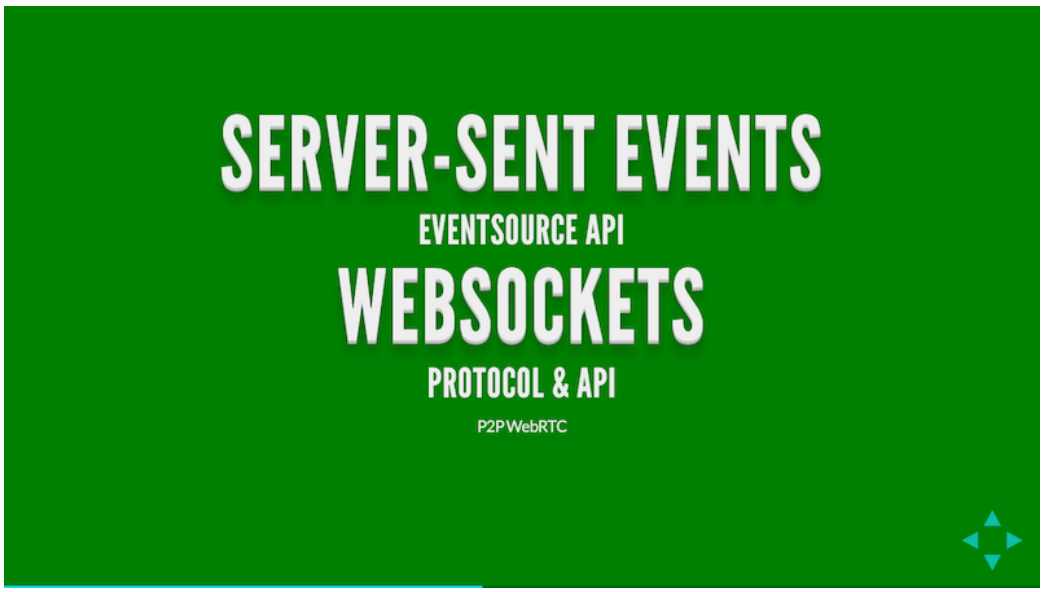
But in order to achieve a persistent connection we had to use technology in ways it hadn't been intended to be used. Not only that, but the solutions were different depending on the web browser. **We hacked the solutions**.

Terms you may have heard before include **HTTP Streaming** and **HTTP Long-Polling**. Traditionally these have been hacks. And, just like the UK Comet Superstore, we want those hacky solutions to go away.

And thanks to standardisation, they eventually will.

The fact that so many developers were trying to achieve "realtime push" meant that the standards body had to take note. After a number of years, and contribution from many of those developers who contributed to the hacked solutions, we now finally have two solutions that let us natively support realtime communication in our web browsers.

But, it's not just web browsers! Because these are defined standards it means that they can be implemented in any technology. From web browsers, native mobile runtimes to constituents of the Internet of Things (search "Arduinos" for some great examples).



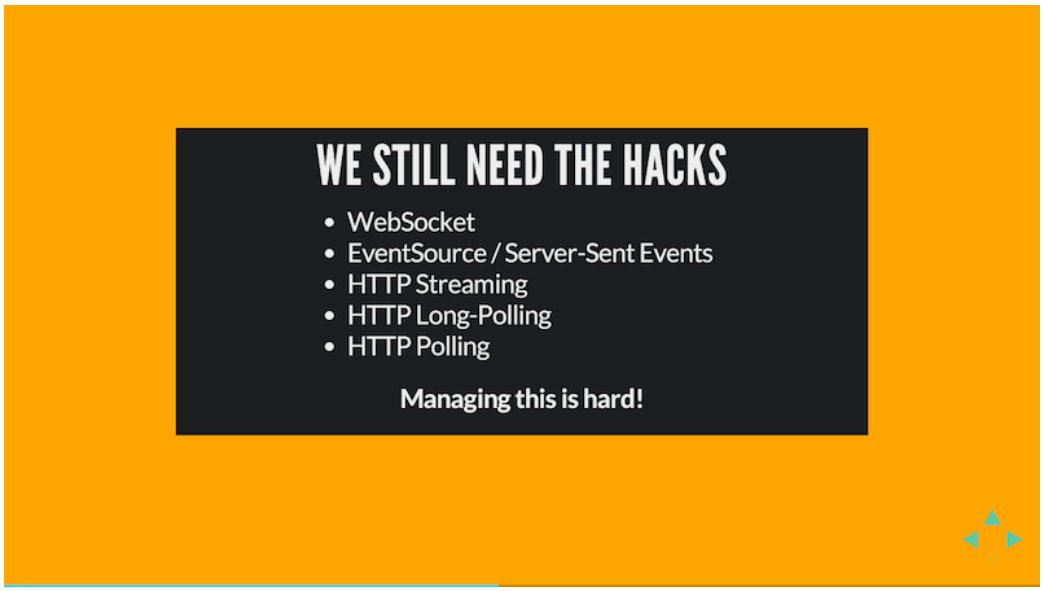
The two solutions we now have are:

- Server-Sent Events and the EventSource API - this is an HTTP-based solution, but taking a standardised approach and avoiding the "hacks".
- WebSocket - The WebSocket protocol and API is probably the most exciting standard. It allows for a persistent, bi-directional, full-duplex connection to be established between a client and a server.

When you are building truly interactive applications you are going to need bi-directional communication. And - as we'll see - even if you're building apps focusing on live content only you'll probably need a level of functionality that will still require bi-directional communication.

Hence, **WebSockets are the best realtime client-server communication transport available**.

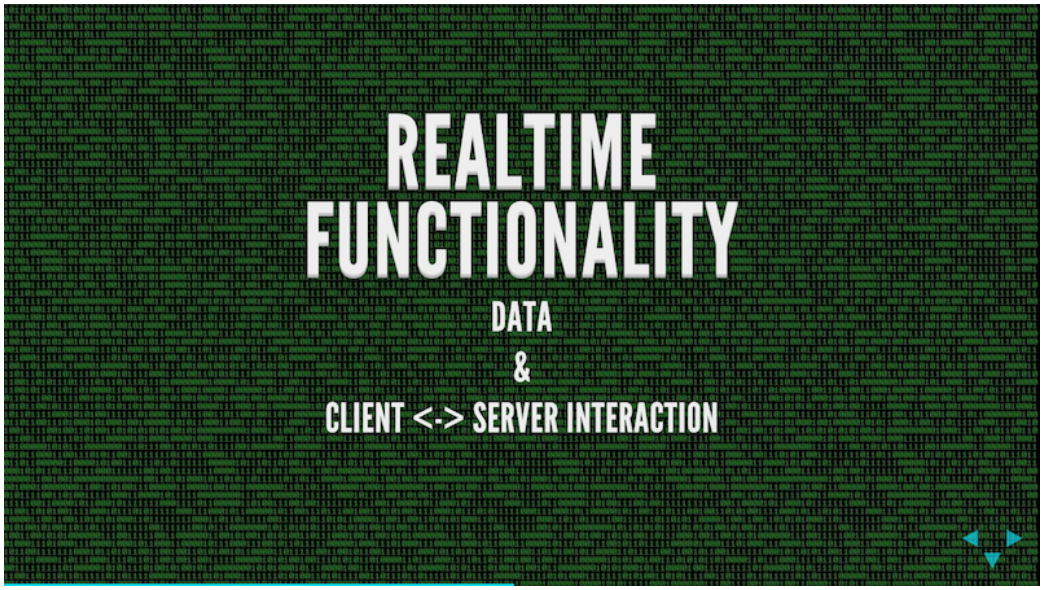
Do not let anybody tell you otherwise!



Unfortunately we do still need the hacks for scenarios such as:

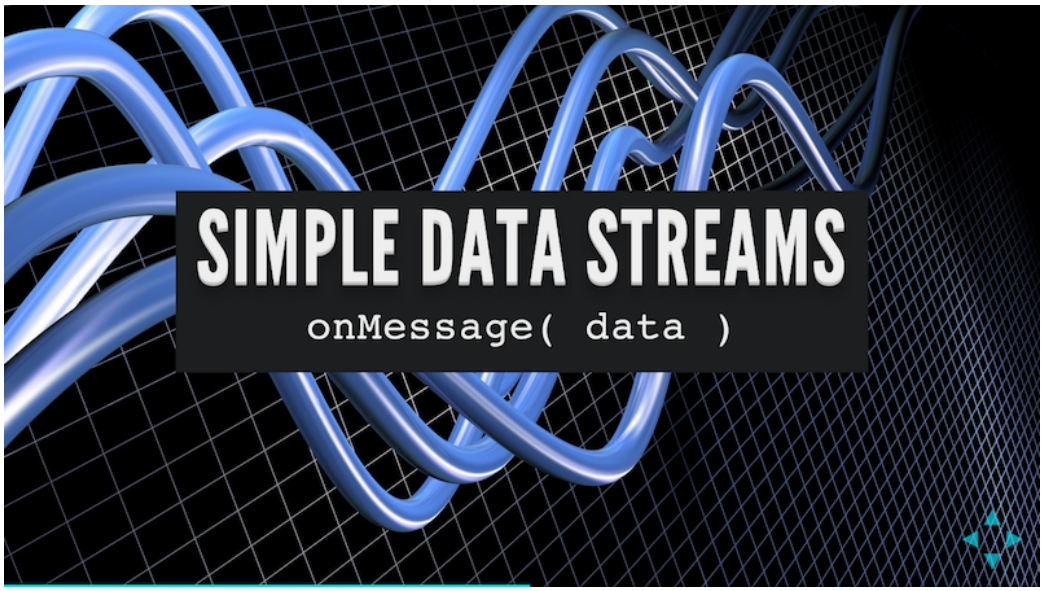
- Where older browsers don't support these new standards
- When tricky networks get in the way. These normally tend to be locked down corportate or educational networks, and some mobile network operators

Therefore, the realtime technology you choose needs to handle these "fallback" scenarios for you.



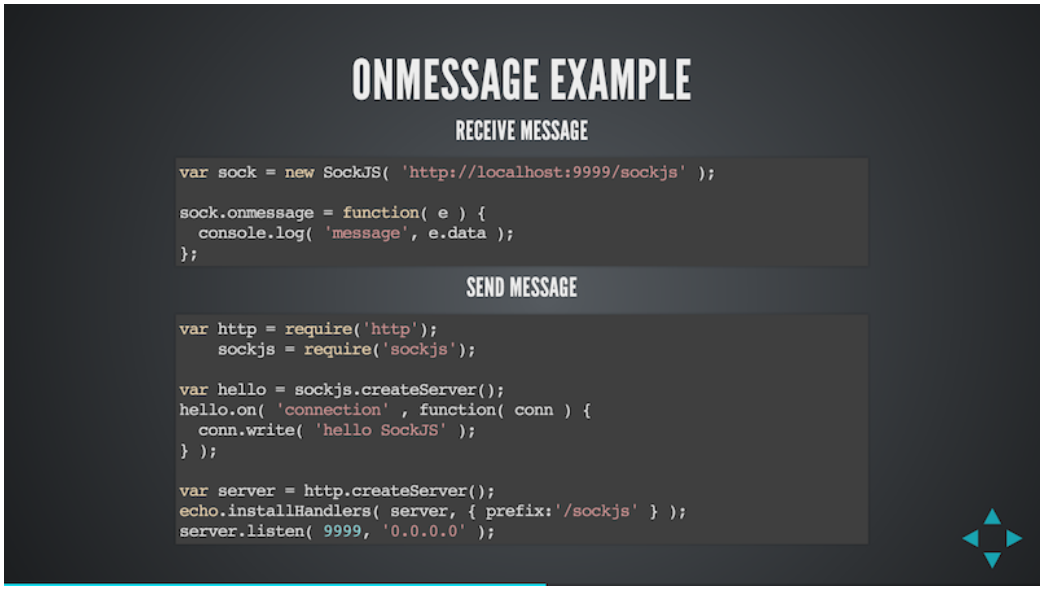
We now understand that we need to connect. But what about receiving data, and what sort of functionality should a realtime solution provide? How should the client and server interact with each other in order for you to add realtime functionality to your application?

I've broken this down into four types of functionality. This isn't perfect, but it's a good starting point.



If you are pushing relatively simple pieces of data from the server to your client then a simple onMessage type of functionality may well be good enough.

Note: I'd like a better name for this but I've gone with onMessage as it reflects the basic level of functionality offered by the EventSource API and is also on the WebSocket API.



In the example above we use a SockJS realtime solution. We connect to our realtime server, assign a function handler to the onmessage property, and that's it.

On the server we do some setup, but in order to push data to the client we simply use the write function on the connection object to push the data.

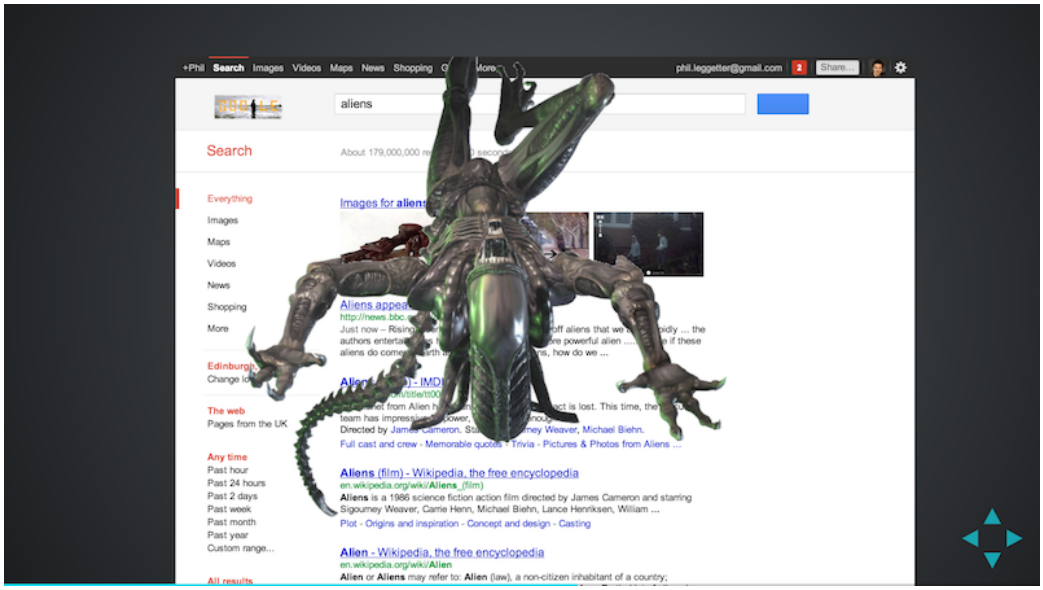
This is the most basic level of functionality. Here SockJS is a great solution as it focuses on basic messaging and connectivity. If you want to, you can layer on much more complex functionality.

The **Publish/Subscribe** pattern is something that the majority of developers understand. In realtime web technology it's the most common form of functionality offered and is very data-centric.

With a PubSub solution you normally subscribe to data using a string identifier. This is normally called a Channel, Topic or Subject.

As the data within your application becomes more complex and you wish to receive updates when

subsets of data change, or you have naturally partitioned your data, **PubSub** works very well.



Although most of us understand PubSub I still like to provide a silly example to try and get us thinking about scenarios that we don't generally associate with subscriptions. In doing that it can get us in the mindset of considering more scenarios as data subscriptions. And in turn, get us thinking about exposing changes in data to users and systems, where we presently let that **change event** go unnoticed.

My silly example is:

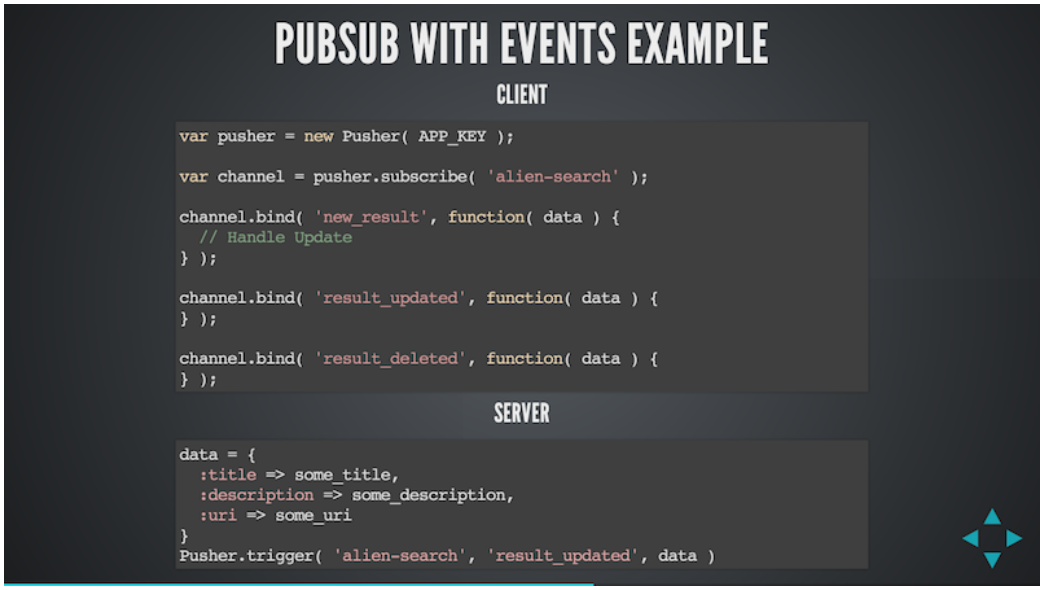
If I do a search for "aliens" in Google I get a relatively fresh set of results returned and displayed to me. Once that data is returned it instantly becomes historical.

If we were then to experience an Independence Day (movie) moment, and the sky around our planet were suddenly filled with Alien spacecraft, this Google search would not be showing me the most relevant information **right now**.

Google should instead be pushing updates of reports of Alien sightings from news channels, pictures of Aliens that have been posted to social networks (because, of course we'd stop to snap that Alien with a ray-gun instead of running), and as the Alien's invade our digital space I'd want to see them in my search results!

Now, Google did have a realtime search solution in the past. I'm not sure quite why they dropped it. Maybe it was because they didn't want to renew their subscription to the Twitter Firehose? I think the lack of this feature is a mistake.

So, start thinking about scenarios in your applications that represent subscriptions. A user being on a URL, where they have an obvious interest in the data on that page, could easily be considered a subscription.



The example above uses Pusher.

On the client I connection and then subscribe to an alien-search channel. This channel name is simply a name I've chosen to use within my application. On that channel object, that represents the subscription, I then bind to different types of events:

- new_result - a new search result has become available
- result_updated - an existing result I already have has been updated
- results_deleted - an existing result has been deleted

This maps really nicely with CRUD operations. As with the channel name, the event names are entirely up to me.

When these events are received by the client the appropriate callback function is called and we can update our application accordingly.

On the server the functionality is really simple. When the event occurs (a new search result is available, one is updated etc.) we trigger the appropriate event on the channel.

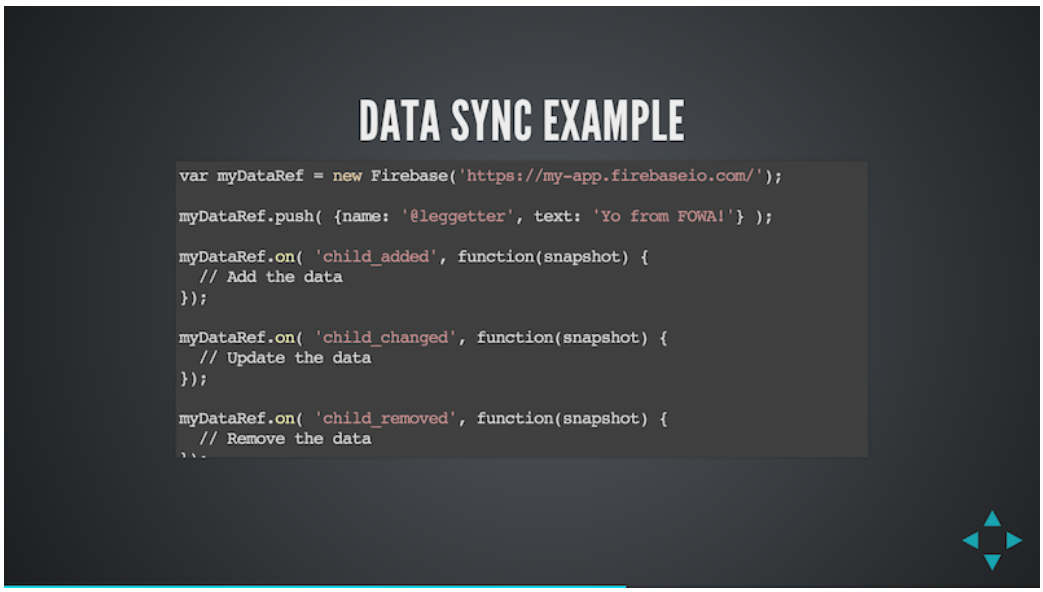
Using a realtime PubSub solution should - and is - this simple.

Not all PubSub solutions will offer the event abstraction that Pusher do. I think it's really powerful and maps very nicely to the evented nature of realtime applications.

The next type of functionality that is available is **data synchronisation**. In data sync solutions the focus is manuipulating data structures and ensuring that all applications get a synchronised version of that data structure.

An obvious example of this is Google Docs where the data, that represents the document that multiple users are editing, is synchronised between all users.

In these scenarios it's also worth being aware of operational transforms - basically detecting and handling data edit collisions.

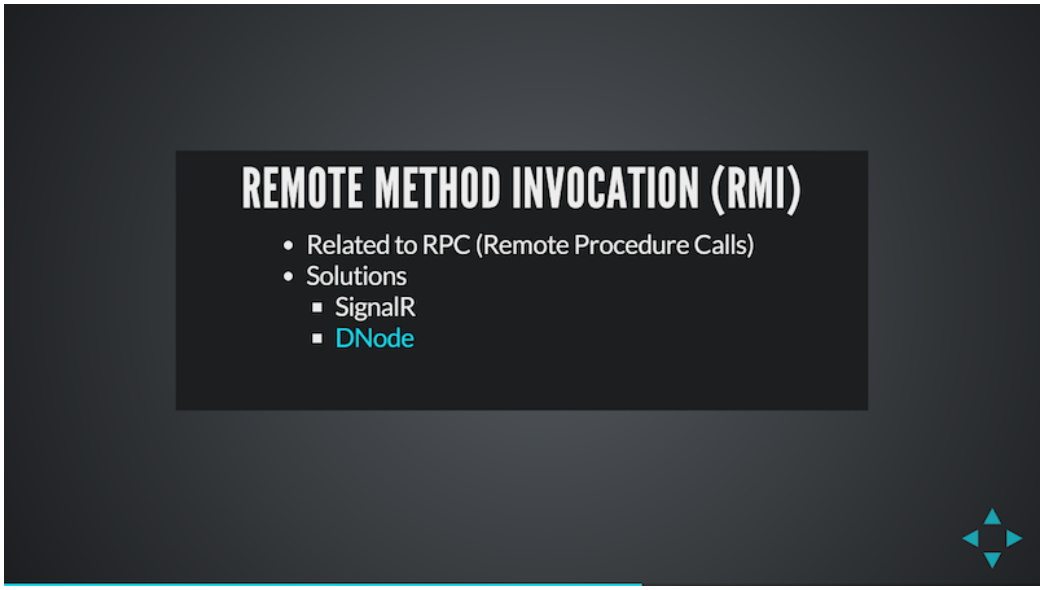


Firebase offer a synchronisation solution. In the code above a structure is referenced and a piece of data added to the collection.

Event handlers are bound on the data reference so that any time new data (a child) is added, a child is changed, or removed, the application is informed. This is very powerful and **very easy** as I found out when I won the ESRI 100 Lines of JS challenge - yeah, 5 lines of code to add realtime

collaborative functionality!

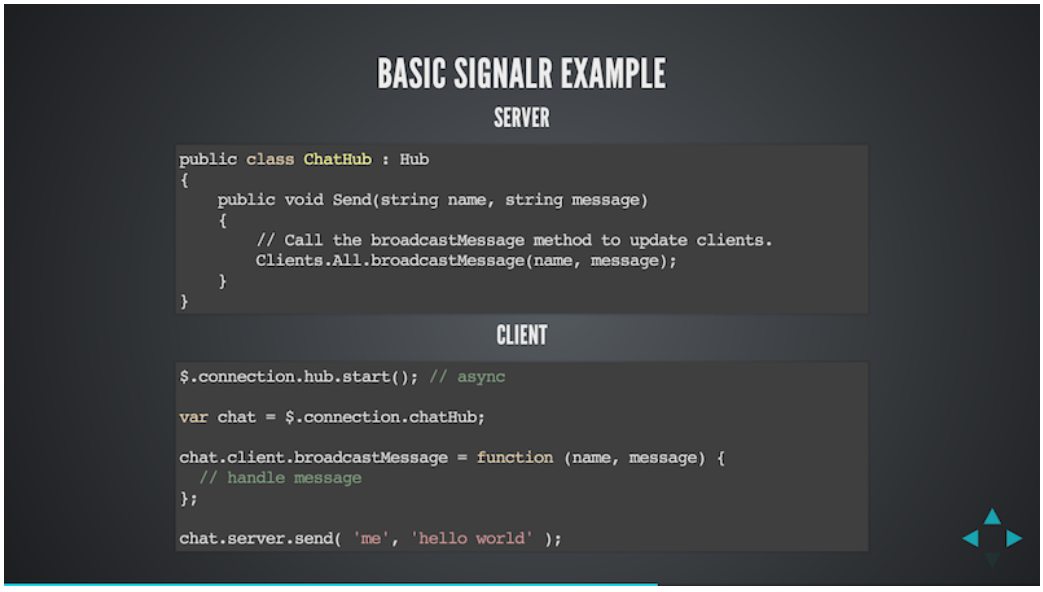
You can see how this could be mapped to a PubSub solution, and maybe it's an abstraction on top of PubSub, but the point here is that it's all about working with a data structure which is a relatively complex abstraction and a specific style of functionality.



The final style of functionality is RMI - **Remote Method Invocation**. Some refer to this as the Object Oriented cousin of RPC. It is ultimately about interacting with objects, and calling methods on them, over a network.

This is amazingly powerful.

It's important to remember that a interacting with an object could result with a network call. PubSub makes that relatively clear, but the RMI abstraction can hide this away to the extent that you may forget you're making network calls.



The above examples using SignalR which is now part of ASP.NET. The SignalR framework does provide simple low-level messaging functionality, but the framework, and their docs, primarily focuses on RMI functionality using something they've called Hubs.

Hubs provide a way of defining objects on the server and the framework creates client-side JavaScript proxy objects to that exposes the server-side functionality.

In this example, a ChatHub extends a Hub object in order to offer the RMI functionality. On the client the chat hub proxy object can be reference via $.connection.chatHub.

The server defines a Send method which can be accessed on the client via chat.server.send.

On the server the SignalR framework exposes a way of calling a function on all connected clients via the Clients.All property. In this case it calls a broadcastMessage function.

The client defines a chat.client.broadcastMessage function which can, and is, invoked from the server.

As I said, this is very powerful!

Next: Choosing your Realtime Web App Tech Stack

By now you should have a good idea of why realtime rocks (is beneficial to users and for your business), have an understanding of some of the raw fundamentals (hacks, Server-Sent Events and WebSocket) and that the functionality can roughly be broken down to onMessage, PubSub, Data Synchronisation and RMI.

In part 3 I'll demonstrate how you can use all this information when **Choosing your Realtime Web App Tech Stack** (wow, that title still isn't catchy!).

GET IN TOUCH

Social

Email

phil@leggetter.co.uk