Building Real Time Web Apps

Using Node.js, Pub/Sub Networks, and WebRTC

Table of Contents


BUILDING
REAL TIME
WEB APPS
Using Node.js
Pub/Sub networks
and WebRTC
BUILDING REALTIME WEB APPS

1. Introduction - Real-Time is the future

As developers, entrepreneurs, and technologists, we all want to know what the next "big thing" is on the internet. Whether you seek to keep your technical skills up to par, or anticipate the next great business opportunity, it’s important that you are ready for whatever that big thing is going to be.

By the time you read this, perhaps the list of trendy technologies and applications will have changed, but for now, these are some of the hottest topics out there:

- Google Glass, FitBits, smart watches and other wearable technologies that allow you to access and record data as you move about in the real world.
- Geo-location based applications like Uber where the users' coordinates have to be continuously exchanged with other users and applications.
- Oculus Rift and virtual reality devices that allow you to interact differently with the virtual and real worlds.
- Social media applications that enable you to communicate with friends anywhere in the world.
- Video based communication applications like Facetime, Google Hangouts, Skype, and a plethora of video conferencing applications that bring people together like was never possible on the internet just a few years ago.
- Massive amounts of data being gathered from factory sensors, home sensors like Nest, and other devices not only connected to the "Internet of Things", but also streaming data and accepting commands from other devices on the internet.
- Raspberry Pi and other devices that make it simpler for anyone to build their own intelligent devices or sensors and attach them to the internet.

We’ve come a long way on the internet from just building static PHP web sites!

These glimpses into the present and future all have one common thread between them: Real-Time communications.

Whether you’re dealing with data streaming over the internet in real-time, or video/audio/text communications, or a combination of the two, you’re dealing with some form of real-time communications.

There are a number of advances in programming that enable this. NodeJS makes it easier to build event driven applications that support real-time handling of data and commands. HTML5 brings together a number of technologies like improved socket communication and WebRTC for peer-to-peer video, audio, and data channels. Cloud based real-time publish/subscribe messaging services are making exchange of data between many browsers much more scalable to the average developer.

The best part is that all of this is now possible in the browser using only javascript code. Cross browser and mobile concerns remain in some cases (especially with WebRTC), but the concepts of this book are the future of web development, even if the exact technologies and code syntax will change over the years.

We believe the best way to learn is by doing, and so this book takes a very tutorial driven approach to build a complete application using a couple different real-time technologies. By the end, you will have built a simple webinar tool that incorporates video chat as well as real-time messaging for comments, voting, and remote synchronization of the webinar presentation.

All the code you need is provided in the book and in online code repositories along the way.

Like you, we're learning too. This book is based on what we've learned in our development teams about the best way to design and build real-time web applications. But we don't know everything, especially in such a bleeding edge area of technology. We are publishing this only as an e-book so that we can continue to give you updates over time as the material changes and is improved upon. Your feedback is welcome anytime!

1.1. About the Authors

Arin Sime, Lead Author

Arin Sime is a software developer, entrepreneur, and the lead author and disturbed visionary behind this book. Arin is also editor of RealTimeWeekly.com, a free weekly newsletter focused on real-time technologies. Arin holds two degrees from the University of Virginia - a Bachelors in Electrical Engineering and a Masters in Management of IT. Arin co-founded software development firm AgilityFeat along with David Alfaro and splits his time between the US and Costa Rica when he is not speaking at conferences and user groups about software development. All the grammar mistakes, occasional bad jokes, and convoluted explanations of this book are the sole errors of Arin.

Allan Naranjo, Lead Developer

Allan is a talented NodeJS developer with experience building applications for social media, training, music, and gaming industries. He has developed an intimate knowledge of WebRTC through many

test sessions of "Can you hear me now?" and canceling echoes. Allan led the architecture and development of the sample applications in this book, and patiently put up with late night testing sessions and feature changes from Arin. Allan lives in Costa Rica but would welcome your invitations to visit San Francisco or New York anytime.

Mariana Lopez, Lead Interaction Designer

Mariana Lopez is the lead User Experience and Interaction Designer behind the webinar tutorial in this book. She started her career as a software developer in Costa Rica and then earned her Masters degree in Human Computer Interaction from Carnegie Mellon University, studying in the US and Portugal along the way. She has led design projects for a wide variety of industries including social media applications, logistics, online education, training, and more. Mariana is also a professor at the Universidad Veritas in Costa Rica.

Daniel Phillips, Lead Designer

Daniel Phillips is the lead designer at AgilityFeat, and is an avid surfer in Costa Rica when he is not designing web and mobile applications or starting CrossFit gyms. Daniel is responsible for the CSS and design of the tutorials in this book, in particular creating the responsive designs to allow the webinar application to work on a variety of devices. Daniel studied marketing at the University of Barcelona.

And the list goes on

Writing a book and building an application are rarely the product of a single person, and in this case, it’s not even the product of only the four people listed above. AgilityFeat co-founder David Alfaro is an evil genius behind much of what our team does. Andrea Phillips, Oscar Phillips, and Esteban Cordero also contributed code to the project. And credit goes to our whole team for joining in testing sessions.

January 2014: Our whole gang at a team building event in the cloud forest of Costa Rica. Yes, there are places in Costa Rica where it gets cold.

1.2. Who this book is for

This book is very tutorial driven, so web developers and technical architects will get the most out of the extensive coding examples presented here. As long as you have a basic understanding of javascript, you should be able to follow the code provided here. It's not necessary that you have previous NodeJS experience.

The chapters on real-time messaging and WebRTC are general introductions suitable for a wider range of technologists and technical managers. These chapters can stand alone to give you an understanding of the technical concepts involved so that you can lead and inspire your team to move towards more real-time applications.

Designers will get the most out of the early chapters where we explain the interaction design of the application, as well as the sections on use cases for these technologies. This not the web of a few years ago that you may be used to designing for, so be ready to challenge old interaction patterns.

Entrepreneurs will particularly benefit from the chapters explaining the current state and limitations of WebRTC, and sample use cases to apply these concepts. When you come up with even more exciting uses for WebRTC, please drop us a line so we can share it with our readers.

There's also something for testers in this book. While we didn't mean to leave any bugs for you to find, but when you do, please let us know! It's also important for you to understand real-time applications because testing them is not so easy. You'll typically need multiple browsers open, and perhaps a variety of ways to put in data to simulate the real-time nature of your specific application. This book will help you understand how real-time applications are built so that you can also determine how to test them.

1.3. Code Samples in this book

This book is designed to be a hands-on tutorial explaining everything you need to know to build the sample webinar application. We try not to assume you know too much already, although you

definitely need to be familiar with Javascript, Git version control, and you should be comfortable with the command line for executing the code. In other words, some basic web development knowledge is going to be helpful if you plan to build the sample application, but it's not necessary that you already be an expert in Node.js or any of the other technologies we cover.

The latest version of the sample application is available publicly on GitHub:

```
https://github.com/agilityfeat/webinar-tool/
```

All of the chapters in this book which include work on the webinar application have the prefix "Coding: " in the chapter title. For each of those chapters, we have created a separate branch of the code on GitHub which matches the code developed at the end of that chapter.

This allows you to jump around in the book based on your interest or knowledge. If you are already comfortable with front-end development and Node.js, then you don't really need us to explain it to you. You're welcome to jump ahead to the chapters publish/subscribe real time messaging. By starting with the appropriate code branch for that chapter, you can skip around the book as you like, as well as cross check your code against our completed code for that chapter.

At the beginning of each chapter is a section titled "Code Samples". This brief section will include two links:

- Code branch for pre-requisites to start this chapter – This is the branch of code on GitHub you want to start with if you have jumped ahead to this chapter. This branch will include a working copy of all the chapters prior to the current one. If you are following the book sequentially and have successfully completed the code in each prior chapter, then you probably don't need this except possibly as comparison.

- Code branch for completed code from this chapter – This is the branch where we have placed the completed code from this chapter. You can use this to compare your code to. In most of the chapters we are including all the necessary code in the chapter text and explaining it as we go, but it may still be helpful to refer to our completed code if you get confused about the placement of the code we are explaining.

1.4. Additional Book Materials and Courses

The official website for this book is http://www.RealTimeWeb.co

You can buy additional copies of this book there, or upgrade to packages that include additional video interviews with other experts in the development of real-time web applications. The upgraded packages of this book also includes access to an online course that follows the patterns set forth in this book, so please check that out if you learn better by video than by reading.

In addition to that, our team coordinates occasional workshops on real-time web application development. Those will be posted on RealTimeWeb.co, or you are welcome to request a public workshop in your area or a private course at your company by contacting Arin@AgilityFeat.com [mailto:Arin@AgilityFeat.com].

1.5. Feedback

We welcome your feedback on any aspect of this book. Please contact Arin@AgilityFeat.com [mailto:Arin@AgilityFeat.com] with any comments, suggestions, and especially success stories about your applications!

1.6. Book revision history and notes

Initial Release 1.0 – April 2014

Release 1.1 - May 2014 - Improvements to the WebRTC calling chapter, code fixes

2. Our application: Real Time Webinar

Most of this book will guide you through the development of a simple, but fairly comprehensive real time application incorporating both data messaging and use of WebRTC. If you follow the book through chapter by chapter, then by the end you will have built the complete application. As noted in the code samples section previously, if you're already experienced in some of the technologies used in our example, then you can also skip ahead to the chapters most relevant to you, and download code that is already developed up to that point in the book.

2.1. A typical webinar experience

Have you ever attended a webinar? Most of us have, and the experience is pretty consistent among the major tools like GoToMeeting or WebEx. They are great tools, which I have used many times.

A typical webinar experience looks like this:

- Presenter setups an account, and creates a link to send to attendees
- Attendees download desktop software to view the webinar
- Presenter shares desktop when the webinar starts
- Attendees join in and can see Presenter's desktop
- Attendees see the slides change in the presentation
- A chat box or "ask a question" feature is probably in the desktop tool, allowing you a way to interact with the presenter

Many of these tools allow for the presenter to turn on their video camera if they choose, though most webinar presenters do not. In a webinar setting, the questions are usually taken via the chat application, and it's unusual that you see a webinar where the attendee asking the question is visible or audible to the rest of the group.

How engaged are attendees in the webinar? If you're like me, then you probably read emails, skim blogs, and do all sorts of things while a webinar plays in the background.

2.2. A more interactive webinar experience

The most important aspect of an engaging webinar is the presentation itself. The content must be compelling and the speaker must deliver it in a way that doesn't put you to sleep. We're not going to fix that with a coding exercise!

For fun though, we might be able to make the webinar technology itself a little more engaging. A presentation on bulgier wheat manufacturing may still be just as dry in this tool, but at least we'll give attendees some widgets to play with if they haven't found a good blog post to read yet!

Our webinar tool is going to allow users to post comments and questions at any point in the presentation. They will be able to vote on slides. And when they ask questions or raise their hand, with a simple mouse click the presenter will be able to start an immediate video chat with that person.

But wait, there's more! We're also going to create a "second screen" experience here. Attendees can view the presentation in real time from their mobile device or tablet, as well as their browser. On the "second screen", they will see the current slide of the presentation updated in real time as the presenter changes slides.

As if that weren't enough, we're going to do all of this in the browser using Responsive Design techniques and WebRTC for the video chat capabilities. No downloads or plugins are necessary!

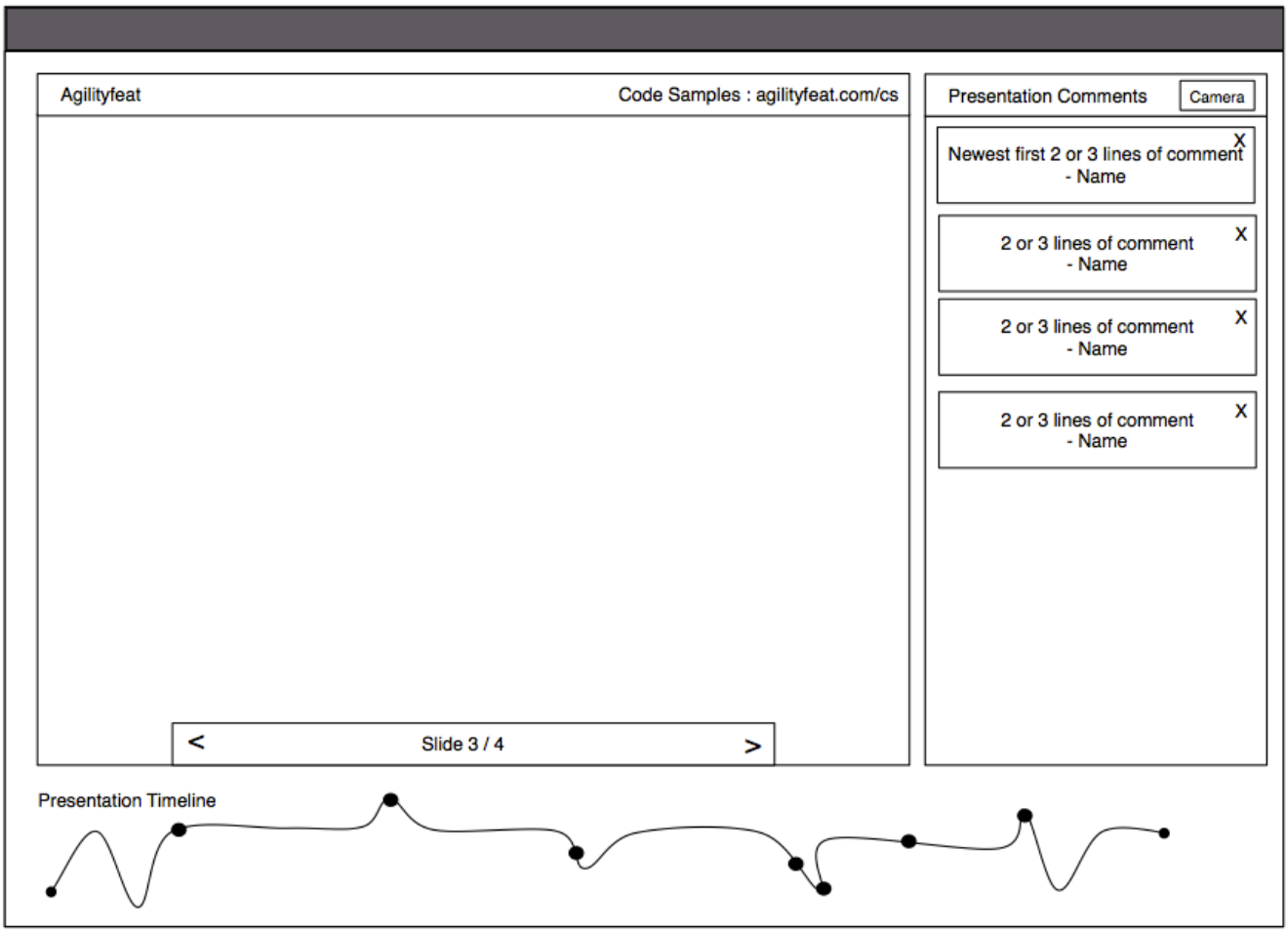
Sidebar: This app will even allow you to lead a conference talk and a webinar together at the same time! We've used this tool to present at conferences before, while others not at the conference logged into the web application and were able to join us virtually. We've used the video chat part to bring in other parts of our team for remote Q&A at the conference too. The ways you will be able to use and abuse this code are limited only by the wifi at the conference you are attending!

Here is what our new and improved webinar experience will look like:

- Presenter sends a link to send to attendees
- Attendees login to the webinar application from their browser or mobile device
- Presenter shares desktop when the webinar starts
- Attendees make comments and cast votes
- Attendees see the slides change in the presentation from their mobile device
- Presenter can call attendees who make comments

2.3. Webinar application wireframes for the Presenter on desktops

To help illustrate the way this application is going to work, let's take a look at the wireframes for what we will build in this book.



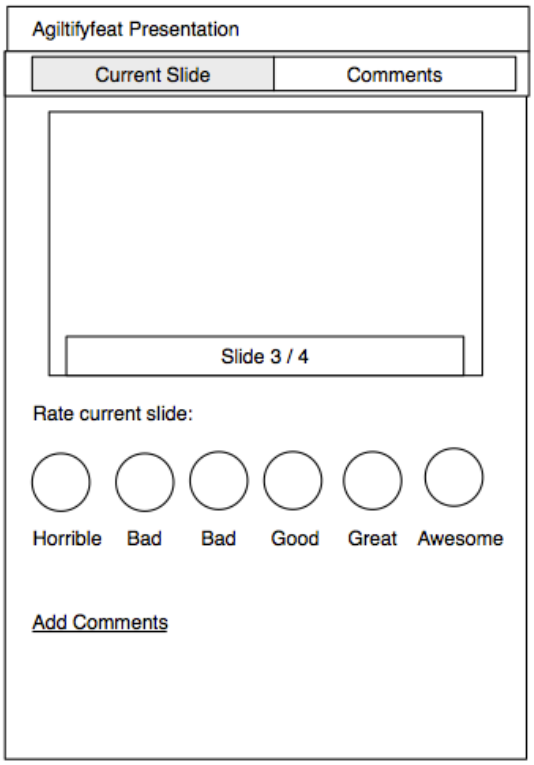
In this wireframe we see how the presenter on their laptop browser will see the webinar application. The chart along the bottom is our real time view of attendee votes as they come in. The right hand panel is where the comments from attendees will be displayed for anyone to see.

In the main part of the screen, we have real estate available to show our presentation slides, with a navigation bar to change the slides. Only the presenter will be able to change the slides.

2.4. Webinar application wireframes for the Attendees on mobile devices

A key use case for this webinar tool is that attendees may not always be sitting at their laptop, they may want to catch the webinar on the train ride home or during lunch. In addition, we are also going to use this tool for in-person presentations at conferences, and we would like attendees to be able to pull out their phones and have a “second screen” experience in a responsive mobile design.

For a tablet form factor, we are going to leave the experience the same as for a desktop to keep it simple. For a mobile device however, we want to add in a responsive view. This is the default view that attendees will see.



In this mobile view of the webinar tool, the UX is very important. We expect to attendees to focus on two things on their second screen: Viewing the current slide and casting votes on the quality of that slide.

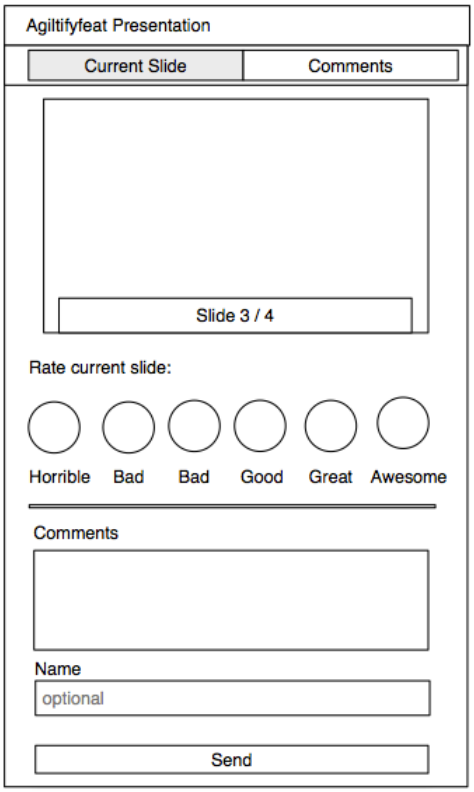
The top of the screen is dominated by an image of the same slide that is currently displayed on the main presentation screen. When the presenter changes the slide, a message will be sent to all attached devices so that they also change to the same slide simultaneously.

Underneath the slide we have a simple interface for casting a vote. Press the large button above your rating for this slide, and your vote will be sent as a message to the main application and the vote added into the timeline of other votes on the big screen. If lots of attendees are casting bad votes, then that is going to provide serious incentive to the presenter, since those votes are being projected to everyone!

Tip

If the pressure of audience members' votes on your material is too much for you to handle, we understand. Don't sweat it! This is just a fun way for us to demonstrate sending real time messages in both directions. The presenter is sending a message to attendees when the slide changes, and attendees' devices are sending messages back to the presenter when they cast a vote. As an alternative exercise, you could change this functionality to allow attendees to answer polls or quizzes during the presentation. Now your webinar application has been converted into an online education tool! That's got to be worth some venture capital funds.

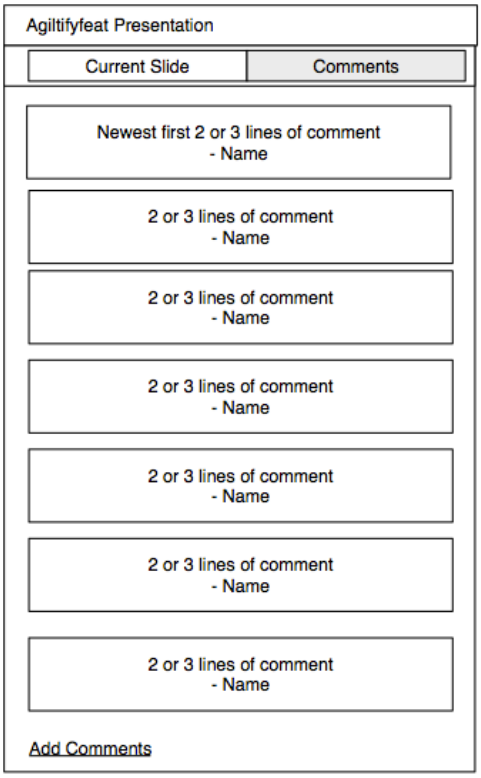
What about comments? We also want attendees to make comments or ask questions during the presentation via their mobile devices. But we consider this to be a secondary task, and so the user experience reflects that. If you want to make a comment, just press the "Add Comment" link or button at the bottom of the mobile view, and you'll see a screen like this:



In this case, we have now made visible a couple fields to enter your comment and a name. Because we think the anonymous nature of the Internet is fun, we're going to leave the name field optional, and we'll assign a random username if you don't supply one.

Note that you are still able to see the current slide while you type your comment. Some of us still haven't figured out how to type fast on a phone!

There's one final thing to consider for our attendees' mobile experience. Despite how awesome this webinar is going to be, you might not be riveted in your seat the whole time. If you get bored, or you just want to test the real time nature of all this, then you should be able to watch the insightful comments and brilliant questions of other attendees stream in.



Just click on the comments tab on the top of the screen, and you will see the most recent comments from other attendees. When an attendee sends a message to the main screen, it is also sent to all other connected devices. Therefore attendees who are at an in-person presentation should see new comments appear here at the same time as they do on the presenters' projected screen.

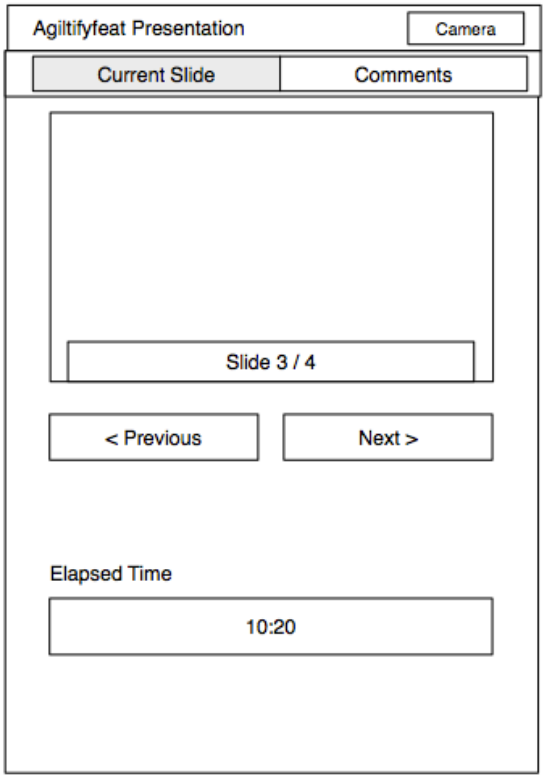
2.5. Webinar application wireframes for the Presenters on mobile devices

We've covered everything about the attendees' views of the application on mobile and desktop, as well as that main screen that might be projected on the wall for an in-person presentation. We could stop here, but since we're having fun with responsive layouts, let's add in one more to create a cool feature for the presenter.

I like to walk around while I make presentations. It eases the nervousness, and helps me get nice and Steve-Ballmer-sweaty. Wouldn't it be great if we could build in our own presentation remote, so I don't have to be standing by my laptop to advance the next slide?

Through the power of mobile devices and real time messaging, we can do exactly that! This is the stuff of science fiction, so buckle up before you look at the next wireframe. [1]

[1] It might be slightly sarcastic to say science fiction, but this is pretty cool, right?



In this view, the presenter is also seeing the current slide. This is comforting for me as the presenter because it assures me that attendees on their mobile devices are also seeing the current slide updated correctly.

Underneath the current slide the presenter has Previous and Next buttons to change the slides. This allows the presenter to change the slides for everyone from their phone, including the main screen if they are projecting at a conference. When I'm using this tool to talk about real time messaging, this is the first way that I demonstrate it working.

Another key to a great webinar is sticking to your schedule. No matter how important that last nugget of information is, you should always end on time! To help with that, we've added a little timer at the bottom of the screen that shows the presenter how long it has been since they advanced from the first slide. There's nothing particularly real time about this since we are only showing it in the presenter's view, but as an additional coding exercise you could extend this timer to work on all mobile views of the application, and keep it in synch via real time messaging.

3. NodeJS Primer

NodeJS is basically just javascript that you can use on the server side as well as the client side. Well, that's a pretty big understatement actually. It's not just about being able to learn a single language for client and server side development. It's also not just about writing in a language that can truly be used on any server operating system. It's also not just the fact that you don't need to configure Apache anymore in order to run the server software itself.

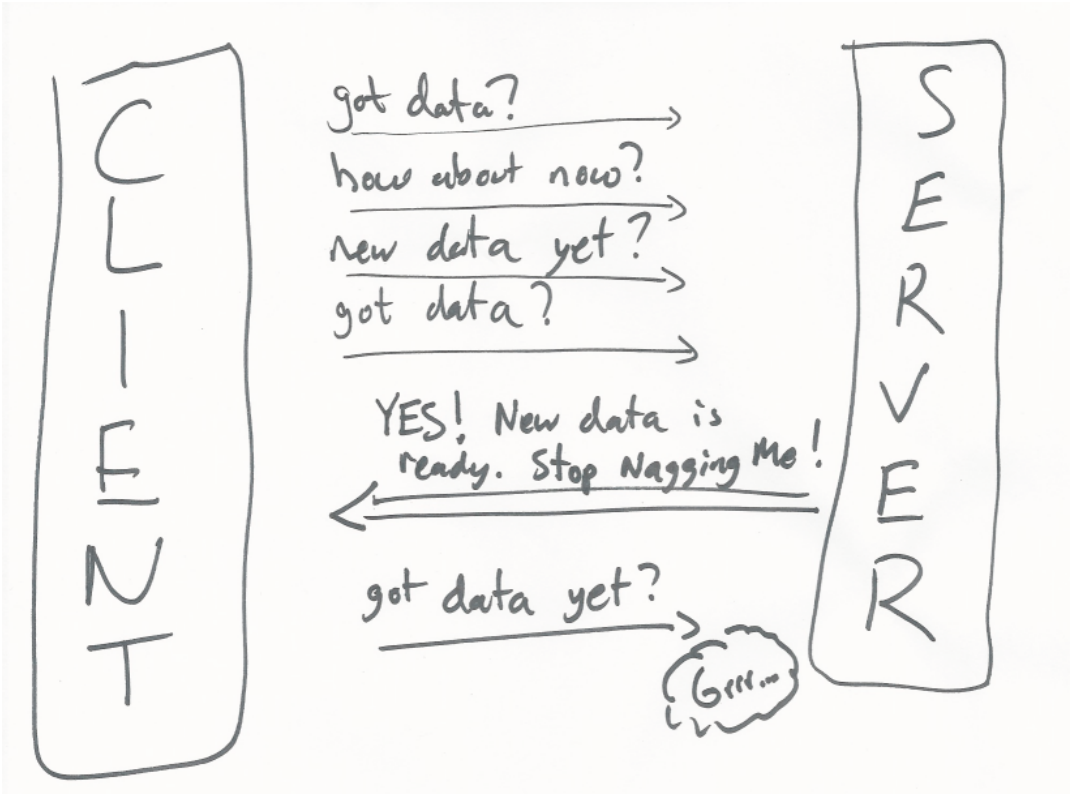
The biggest advantage of NodeJS, certainly for those of us interested in real time applications, is that it has non-blocking I/O, and a single threaded event loop. That means that it is an event driven language.

NodeJS was first created in 2009 by Ryan Dahl, and is now maintained by cloud infrastructure company Joyent. [2] NodeJS is based on the V8 Javascript engine developed by Google.

3.1. Polling solutions are wasteful

Ryan's original desire was to build a web based event driven language, so that you don't have to do polling back to a server in order to get updates for the browser page. Event driven languages are inherently faster performing and lower load on the server than a polling solution, because they allow the server to push updates to the client as they are ready.

This is what an older polling solution might look like. It's not pretty.



Client Polling the Server for Data Updates

[2] http://en.wikipedia.org/wiki/Nodejs

In a polling solution, the client has to repeatedly ask the server for data updates. The server has no idea where the client is, or how to reach it, and so it is incumbent upon the client to ask for data as often as it wants that new data.

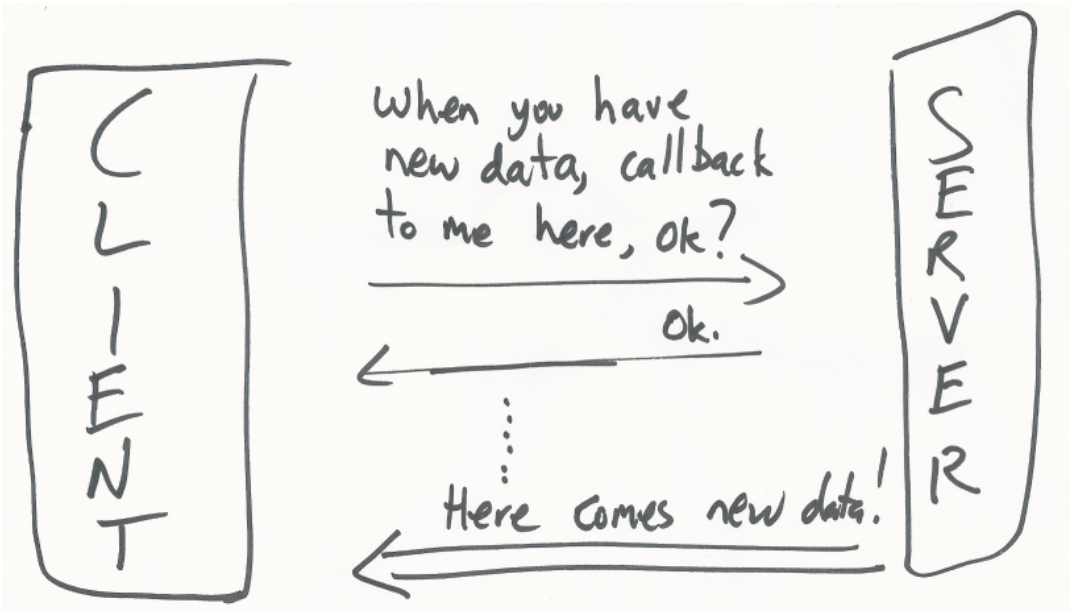
This is a very inefficient process for both client and server. On the client side, if you want to give users a service level agreement where the data you show is never more than a second old, that would mean you need to poll the server at least every second in order to get new data. That requires a lot of calls from the client side.

On the server side, this is also very inefficient. The server is constantly replying to data requests from every client connected to it, and this puts a lot of load on the server, especially as the application becomes more popular and the number of connected clients scales up dramatically.

If servers had a personality, they would find polling to be extremely annoying.

3.2. Event driven is more efficient

An event driven model is much less noisy and more efficient.



Event-Driven communication between Client and Server

In this model, each client just needs to tell the server how to find it. They pass a reference to a “callback method”, which the server stores. When the server has new data available, it calls back to that method to let the client know about the data.

In this model, the client does not need to make repeated requests of the server, and so that frees up a lot of bandwidth in the client side browser. The client just needs to trust the server that as soon as it does have new data available, it will push that data back to the client immediately.

Not only is less traffic generated between the client and server, but the data is more fresh. It should be very near real time at this point. It’s also a much more scalable architecture because the server can handle more clients connecting to it.

3.3. NodeJS is an event driven language

NodeJS was built from the ground up to be event-driven. While you can add libraries to other languages like Ruby to make them also be more event-driven, the fact that NodeJS is truly designed to be event-driven makes it a compelling choice for real time applications.

NodeJS itself is just an application engine written in javascript. You can write console apps, web applications, worker threads, whatever you want. In its simplest form, you just put your javascript in a .js file, and then execute it from the command line like this:

```
Node <script name>.js
```

That's it, but of course you can do much more powerful things with NodeJS, and there are many add on frameworks you can use. If you're familiar with the Ruby language, then you know that in Ruby the most common web framework is Rails (thus the common phrase "Ruby on Rails"). Ruby is the language, Rails is the framework to build web applications.

In NodeJS, there are a number of web application frameworks that you can choose from. Perhaps the most commonly used, and the one we will use in this book, is called Express. We'll walk you through a few basics of that in the next chapter, but you should know that there is much more to it than we will dive into this text, and you can learn more at ExpressJS.com.

Using Express gives you some familiar patterns to your NodeJS code, such as separating view logic from data models, and a public folder for all static assets. In the next two chapters, we'll get our NodeJS application setup using Express, and then we'll apply a bootstrap based theme to it. After that, we get into the real-time coding.

4. Coding: Setting up the NodeJS application

4.1. Code Samples

In order to allow you to jump around this book as you like, at the beginning of each chapter we are going to include links to two branches of code on GitHub:

- **Code branch for pre-requisites to start this chapter** – This is the branch of code on GitHub you want to start with if you have jumped ahead to this chapter. This branch will include a working copy of all the chapters prior to the current one. If you are following the book sequentially and have successfully completed the code in each prior chapter, then you probably don't need this except possibly as comparison.

- **Code branch for completed code from this chapter** – This is the branch where we have placed the completed code from this chapter. You can use this to compare your code to. In most of the chapters we are including all the necessary code in the chapter text and explaining it as we go, but it may still be helpful to refer to our completed code if you get confused about the placement of the code we are explaining.

Prerequisite code for this chapter: None

Completed code for this chapter:

https://github.com/agilityfeat/webinar-tool/tree/nodesetup

4.2. Installation Steps

Setting up NodeJS is pretty straightforward, but of course will vary for everyone. We're following the installation steps below on a Mac OSX, and your steps for other operating systems will be similar, but different. If you have trouble following the following steps for your operating system or specific situation, then google will hopefully be your friend. You might also want to check out this post that shows a few simple ways to install NodeJS for different operating systems.

http://bevry.me/learn/node-install

4.3. Prerequisites for installing NodeJS

If you have a brand new Mac that you've never done any software development on, you're going to need to do the slightly dreaded Xcode installation before you go any further. The good news is it's free, the bad news is it's a very large download. More info on Xcode here:

https://developer.apple.com/xcode/

The first step is you're going to need git installed. We'll assume that you already know all about git, and it's superpowers for file versioning and branching. If not, you should take a little time to

get familiar with it since the code samples in this book are all hosted at github and you'll need to understand git commands in order to access them.

If you don't already have git installed on your computer, check out this site for installation advice:

http://git-scm.com/download

If you're new to git commands, then this snappy little tutorial from GitHub and built by Code School will guide you through learning all the most important git commands:

http://try.github.io/

Installing Node Version Manager

We're going to use a tool called Node Version Manager (NVM) in order to install NodeJS. If you're a Ruby developer, then you're probably familiar with the Ruby tool called RVM. NVM is the basically the same thing, but for NodeJS.

NVM will allow us to install multiple versions of NodeJS on our computer, and easily switch between them when using different projects. If this is your first NodeJS project, then that may seem like overkill, but trust us, it's a life saver once you become a pro and have multiple NodeJS projects going. Not every project will be on the same version of NodeJS, and NVM makes it super easy to manage this.

Here again, installing NVM will be a little different on each operation system. Your best bet is to look at the project's page on GitHub for up to date installation instructions for your situation:

https://github.com/creationix/nvm

For our situation, we're on a new Mac running OSX and we're going to install NVM with a simple curl command:

```
curl https://raw.github.com/creationix/nvm/master/install.sh | sh
```

If your installation worked, then you should be able to type "nvm" at the command line to see the Node Version Manager help page come up with all the beautiful NVM commands you can use.

4.4. Installing NodeJS

Now that we have NVM installed, actually installing NodeJS is going to be a snap. First, we want to run this command to see the latest versions of node available for download:

```
nvm ls-remote
```

That's going to give you a long list of version numbers from v0.1.14 all the way up to the latest. We're going to install version v0.11.11 in this book, but you may choose to use a later version as you're reading this. It's possible there will be inconsistencies if you do use a later version, but we're not doing any crazy NodeJS in this book, so it will likely work on the latest versions too.

To install our specific version of NodeJS, run this command:

```
nvm install 0.11.11
```

You'll see output similar to this:

```
Arins-MacBook-Pro:~ arinsime$ nvm install 0.11.11
######################################################################## 100.0%
Now using node v0.11.11
```

You should now be using NodeJS version 0.11.11, and you can verify this by typing "node –v":

```
Arins-MacBook-Pro:rtbook arinsime$ node -v
v0.11.11
```

Whenever you want to start using NodeJS in the future, you will likely need to tell NVM which version of NodeJS to start using. To do this, first list all the versions that you have installed locally with the command "nvm list":

```
Arins-MacBook-Pro:~ arinsime$ nvm list
  v0.11.11
current:        v0.11.11
```

In this case, we just have one version installed and we're already using it. But if you come back to your project later and the node commands don't work, it may be that you don't have a version of NodeJS selected. You can do this by running this command:

```
nvm use 0.11.11
Now using node v0.11.11
```

4.5. Using barenaked NodeJS

Now we have our base installation of NodeJS complete. The next step will be to quickly show you how to execute a simple NodeJS file [3]. (After that, we'll get you setup with a web framework called Express.)

Open up your favorite text editor and create a file named 'http_example.js' with these contents.

```
var http = require('http');
var server = http.createServer(function (request, response) {
  response.writeHead(200, {"Content-Type": "text/plain"});
  response.end("Never laugh at live dragons!\n");
});
server.listen(3000);
console.log("See the output at http://localhost:3000/");
```

After saving the file with those contents, just type the following command to start your node http server:

```
node http_example.js
```

Note that the file name can be whatever you want, just execute that file with the node command. Now if you visit localhost:3000 in a browser, you'll see the output:

```
Never laugh at live dragons!
```

[3] More examples like this at http://howtonode.org/hello-node

That's it! If you've made it this far then you have successfully installed NodeJS and proved that you can run a simple http server with it. The next step in our project is to install Express.

4.6. Adding the Express web framework to NodeJS

Let's start a new directory for our project. This is going to become the root directory for our project in the rest of this book.

```
mkdir webinar-tool
```

In this directory, we're going to install a framework called Express [4], which will give you a familiar model-view-controller (MVC) pattern for defining your application in.

In the webinar-tool directory, create a file called package.json, and put this code in it:

```
{
  "name": "webinar-tool",
  "description": "RealTimeWeb.co webinar tool",
  "version": "0.0.1",
  "private": true,
  "dependencies": {
    "express": "3.x",
    "ejs": "0.8.4"
  }
}
```

You may want to change the version of Express to whatever is most recent as you read this. We're not doing anything fancy with Express here so they changes in subsequent code examples are likely to be minimal.

After creating your package.json file in the webinar-tool root directly, run this command in the same place:

```
npm install
```

This will install Express and all the dependencies you need. You can see that list of installations by running:

```
npm list
```

You should now also notice a node_modules directory in your application with an express folder underneath.

Now that we have Express installed, we can configure a simple "hello world" just to prove to ourselves that Express is working and we can serve up web pages.

We need to setup a couple standard directories and files for Express and Node. First, create a file in the root directory called server.js. This is the main application file, and you'll need to put the following code in it:

[4]These instructions are based on: http://expressjs.com/guide.html

```
var     express         = require('express');
var port = process.env.PORT || 3000;
var app = express();
app.configure(function(){
        app.set('views', __dirname + '/views');
        app.set("view options", {layout: false});
        app.engine('.ejs', require('ejs').__express);
        app.use(express.cookieParser());
        app.use(express.session({ secret: "webinar-tool" }));
        app.use(express.json());
        app.use(express.urlencoded());
        app.use(express.methodOverride());
        app.use(app.router);
        app.use(express.static(__dirname + '/public'));
});
/*********** Routes ***********/
require('./routes/routes')({
        app     : app,
        port    : port
});
/*********** Start listening to requests ***********/
app.listen(port, function(){
        console.log("Express server listening on port %d", port);
});
```

This requires the Express library, and tells the app to run on port 3000 unless otherwise specified. It also configures Express to look for the a views directory (for .ejs files that will contain all the visual layouts), a public directory (which will contain static assets like images, javascript files, and CSS files), and a routes file that will contain the information for routing specific URL's to the right view files.

You'll need to create each of these directories

```
mkdir views
mkdir public
mkdir routes
```

In the new routes directory, you also need to create the routes.js file, and put this in it to start:

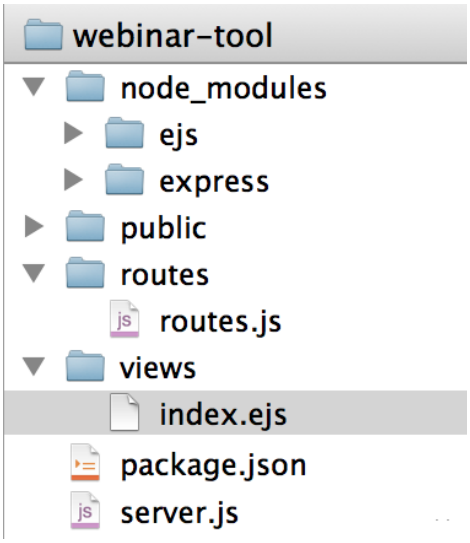
```
var routes = function (params) {
        var app = params.app;
        app.get('/', function(req, res){
                        res.render('index.ejs');
                })
}
module.exports = routes;
console.log("All routes registered");
```

We'll add in more routes over time, but for now this will tell the server to route all root URL requests to the index.ejs file in the views directory.

That means we better go create that file before we try to run our server, so create a file called index.ejs in the views directory, and put some simple html code in it:

```
<!doctype html>
<html lang="en">
        <head>
                <title>Webinar-tool RealTimeWeb.co</title
        </head>
        <body>
                Greetings from your future webinar-tool
        </body>
</html>
```

We're just about ready to run our NodeJS website! Confirm that your app structure looks like this:



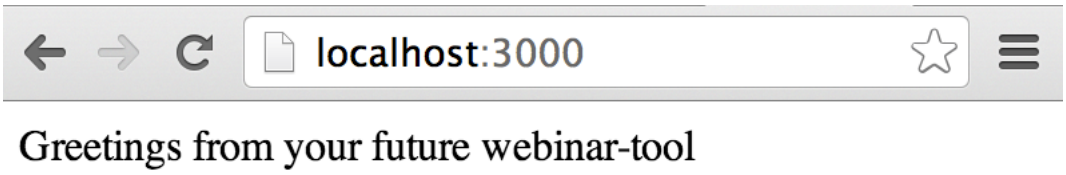
Ready to start your site? Just type this command from the root directory:

```
node server
```

You should get an output that looks like this:

```
All routes registered
Express server listening on port 3000
```

Navigate to http://localhost:3000/ in your browser and you will see your (very simple) site!



Did you have any trouble getting this far? You can view our complete code so far on github:

https://github.com/agilityfeat/webinar-tool/tree/nodesetup

5. Coding: Front end design setup

5.1. Code Samples

Prerequisite code for this chapter:

https://github.com/agilityfeat/webinar-tool/tree/nodesetup

Completed code for this chapter:

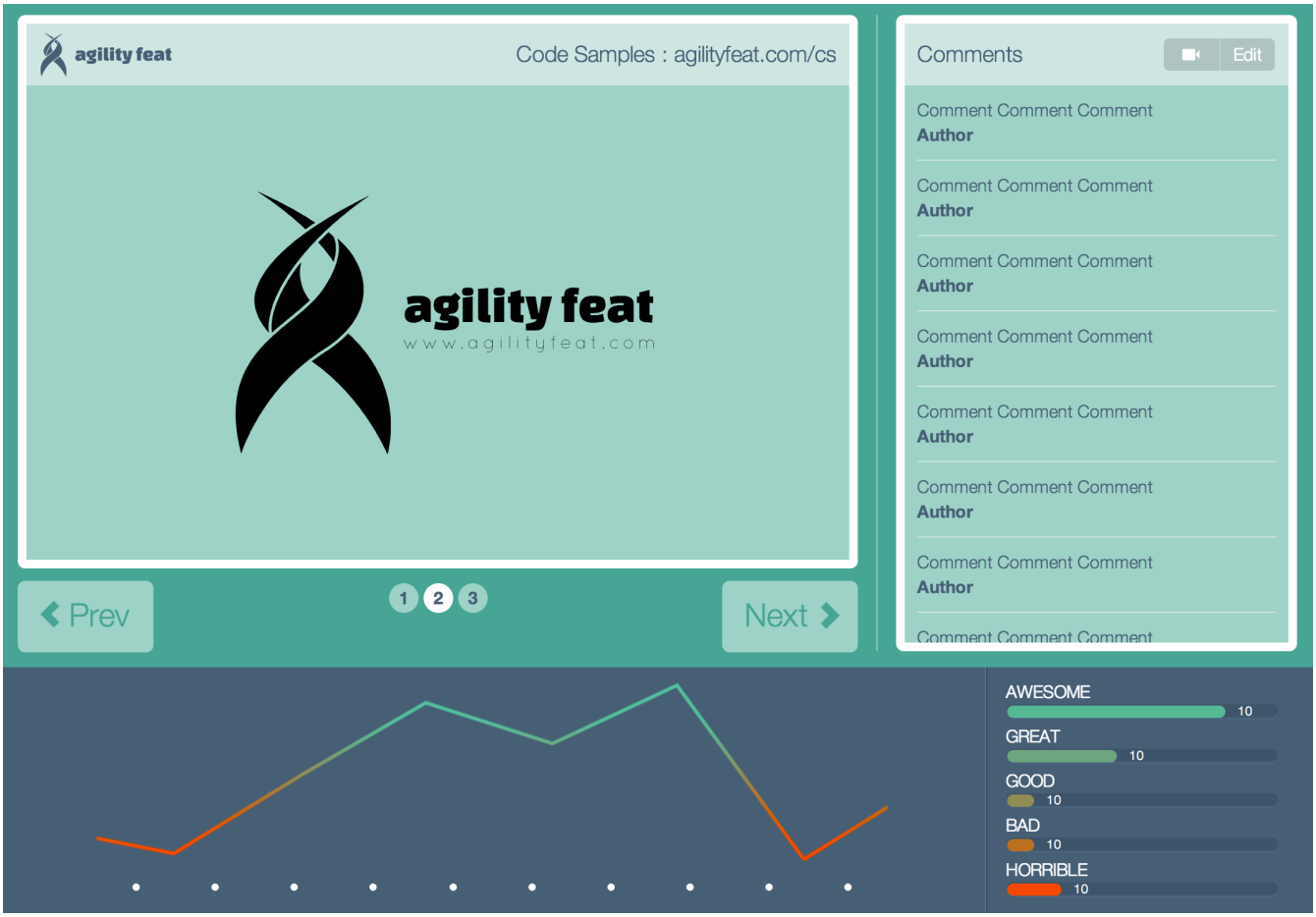
https://github.com/agilityfeat/webinar-tool/tree/designsetup

5.2. Introduction

Up to this point, we've designed the user interactions with our site, and setup a node and Express project. The next step for our webinar tool is to get the design elements in place. The wireframes we looked at previously were designed by our User Experience (UX) lead Mariana Lopez. She has now handed those off to Daniel Phillips, our Visual Designer to produce mockups. Let's take a look at a couple of Daniel's mockups.

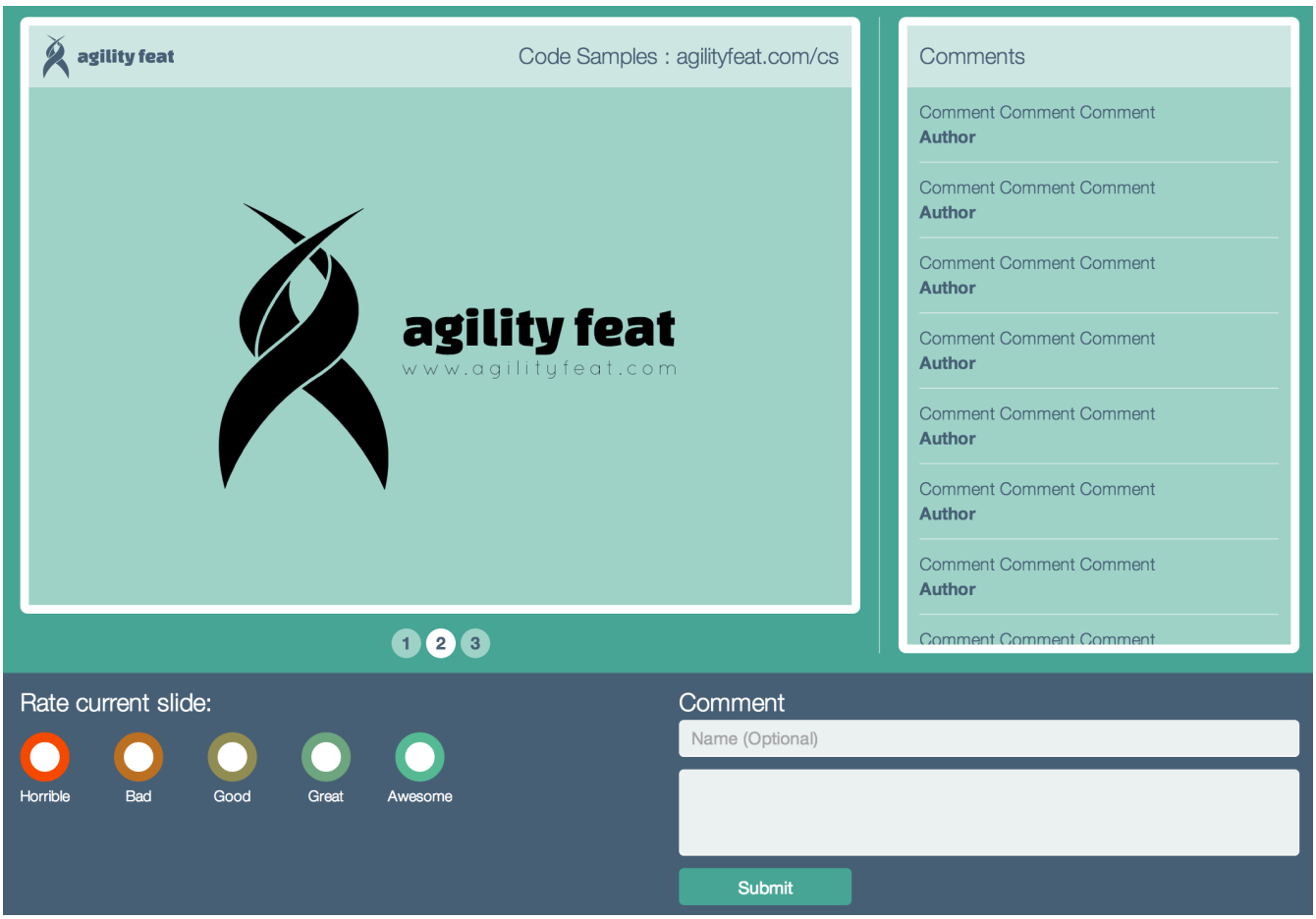
5.3. Visual mockups review

First, we will look at the Presenter's View, as they see it on a desktop. Remember that this is the screen that the Presenter would project on the wall during an in-person presentation, and so it contains the comments coming in from attendees, the latest votes from attendees, and it shows the current slide.



Notice that in this view, if we are logged in as a presenter, then we have access to the Next and Previous buttons to move between slides. We can also click on the slide number to jump to a particular slide.

If a regular attendee brings up the main page of the webinar application, they see a few key differences, just as the mockups suggested. Here is the design that Daniel produced.

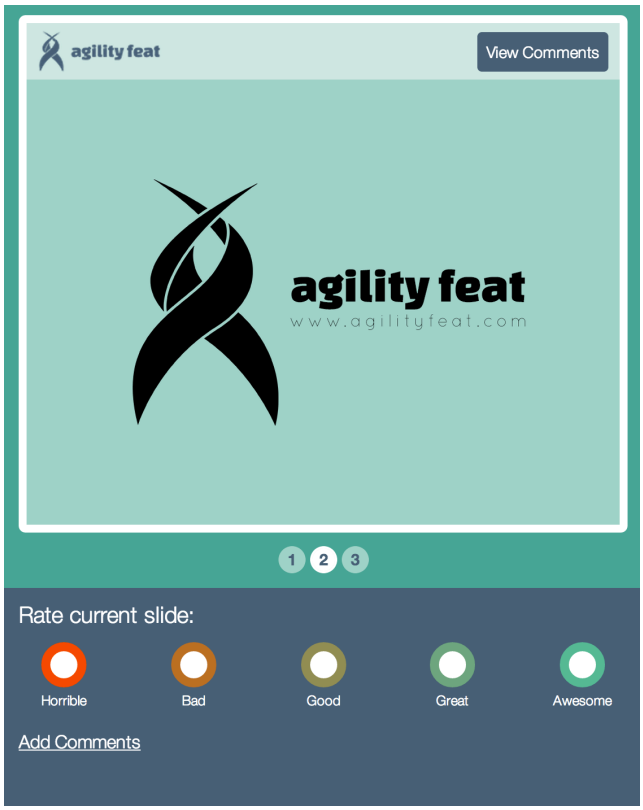


In the desktop view, the attendee has a comment box, and a set of controls for rating the current slide. Since we have enough real estate in the browser window, these are displayed at the same time as the slides and the comments streaming in from other users. Notice that the Previous and Next buttons to control the slides are gone now – we don't want the attendees to change slides on us.

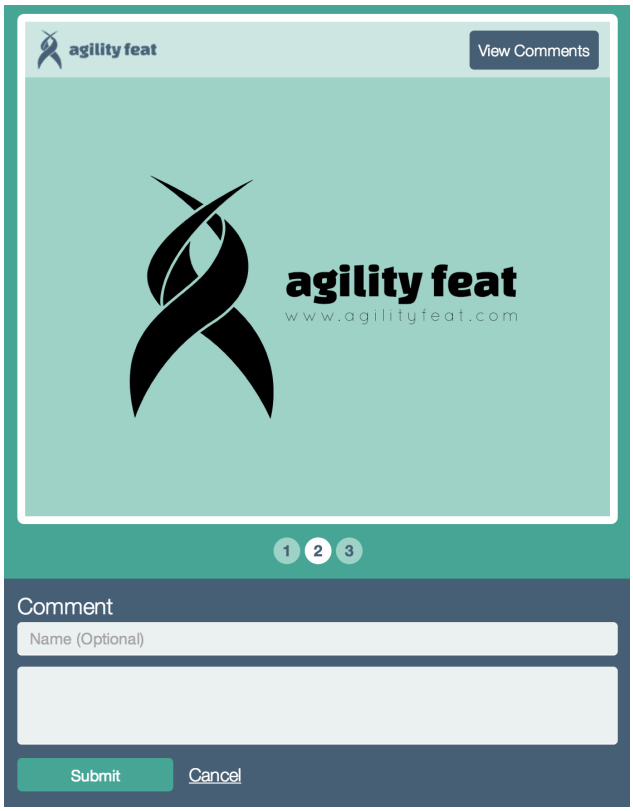
Next, let's take a quick look at the designs that Daniel came up with for the mobile views.

If the presenter brings up the webinar tool on their mobile device and logs in as a presenter, then they see the slides and the Previous / Next slide controls. This allows the presenter to use their phone as a "remote" to control the presentation.

If an attendee brings up the webinar tool on their phone, then they will not get the controls, just like in the desktop view. They will however be given a prominent set of controls for voting, as shown in the next design.



If the attendee wants to make a comment from their mobile device, they can do that by clicking on Add Comments, and then the voting controls are replaced with a comment form.



Notice that in either view, we are keeping the current slide displayed front and center on the attendee's mobile device. This is the most important part of the application, so we want to keep that the focus.

As explained previously when we went through the wireframes, we chose to encourage attendees to vote more often than comment, and so the voting tools are given precedence in the attendee's mobile view, but both features are easily accessible. Deciding on how to guide the users to do the actions you want them to do most frequently is especially important in a real-time application, because you may be presenting them with so much data or visual stimuli that it will be easy for them to get distracted from the other functionality of your application.

5.4. Setting up the Responsive Design

Because we want to use the same code base to support the presenter's laptop, the presenter's smartphone, and the smartphones or tablets of attendees, we need a responsive layout.

If you're not familiar with what that means, here is the short explanation. We are going to have one set of code do the functionality for all different devices. Everything is still HTML, Javascript, and NodeJS. You'll see a tag in the HTML and CSS that talks about "viewport", which is a way to determine what size device the user is looking at our website with. Based on that size, we can make some assumptions about whether the user is on a laptop, a tablet (which we will treat the same as a laptop in this application), or a mobile device.

If the application is run on a mobile device instead of anything else, we will present a different stylized view of the application. The underlying code has no idea nor cares on what device the app is being presented, it behaves the same way.

Tip

This is not a book on design, and so we are not going to explain Responsive Design in depth, and all the design artifacts for this webinar application will be provided at the end of this chapter. But we should briefly explore how this works, so that those of you new to responsive design will at least have an idea where to look if you want to change something in the code.

In the main view for our application (/views/index.ejs), we're going to have a couple of meta tags like this:

```
<head>
        <title>Webinar-tool RealTimeWeb.co</title>
        <meta name="viewport"
          content="width=device-width, initial-scale=1.0, user-scalable = no">
        <meta name="format-detection" content="telephone=no">
</head>
```

In the viewport meta tag, we are setting our application to use the full width of whatever device we are on, with an initial-scale of 1. The scale just tells the browser that 1 CSS pixel is equal to 1 viewport pixel.

The user-scalable attribute is often not used, and some will recommend strongly against using it. By setting this to no, we are saying that users on a mobile device are not going to be able to zoom in and out on their phone. That is very common behavior for a mobile user, and you should think carefully

before restricting it. In our case, our page will be displayed very simply on mobile devices, and there are no complicated forms or images that users would want to zoom in on. Therefore, we've chosen to not allow user scaling.

The second meta tag for format detection doesn't have anything specifically to do with the layout of the page, we are just telling mobile browsers not to try and make telephone numbers in the application hyperlink directly to the phone dialer. You can leave this out if you prefer.

The actual changes of layouts happens in the media-query.css file, which you'll see when you pull down all the design artifacts for this project from github. In this CSS file, you will see CSS definitions like the following:

```
/* Smaller than standard 960 (devices and browsers) */
  @media only screen and (max-width: 959px) {
    .rateOptionsWrap{
      margin-right: 120px;
    }
  }
/* Tablet Portrait size to standard 960 (devices and browsers) */
  @media only screen and (min-width: 768px) and (max-width: 959px) {
    .rateOptionsWrap{
      margin-right: 80px;
    }
  }
```

In these two CSS definitions, we see that the right margin wrapping around our "cast votes" control is changing based on whether the @media being used is likely to be a mobile phone (in the top example) as opposed to a tablet or laptop (in the second example).

Rather than explain all of Daniel's CSS decisions, we're recommending in this chapter you just download our code to use as a starting point. If you're an experienced designer than you can of course modify it from there however you like. We'll include all the design artifacts in a link at the end of this chapter. For now, if you want to see the complete media query CSS file, you can check it out here:

https://github.com/agilityfeat/webinar-tool/blob/designsetup/public/stylesheets/media-queries.css

5.5. Bootstrap design framework

The other design framework worth mentioning here is Bootstrap. Bootstrap is a CSS and Javascript framework that you can download for free and use as the basis of your responsive design. It's designed with responsiveness and mobile in mind, and is widely used in websites.

We'll include bootstrap in the design assets that you download at the end of this chapter, but you can learn more about it in the meantime here:

http://getbootstrap.com/

The bootstrap download also includes a few fonts that will need to go into our project. Here is what you will see included with Bootstrap:

```
bootstrap/
├── css/
│   ├── bootstrap.css
│   ├── bootstrap.min.css
│   ├── bootstrap-theme.css
│   └── bootstrap-theme.min.css
├── js/
│   ├── bootstrap.js
│   └── bootstrap.min.js
└── fonts/
    ├── glyphicons-halflings-regular.eot
    ├── glyphicons-halflings-regular.svg
    ├── glyphicons-halflings-regular.ttf
    └── glyphicons-halflings-regular.woff
```

Files included with Bootstrap. Image from GetBootstrap.com

Bootstrap is dependent on JQuery, so we will also be including that in our project.

5.6. Viewing the HTML mockups

To see how the responsive design and bootstrap work together to implement the visual designs that we saw at the beginning of this chapter, we have provided you with a complete copy of the HTML/CSS mockups.

Different development teams handle implementation of the visual design in different ways. Sometimes visual designers will give the developers a set of PDF files showing how the design should look, and they “cut up” the images from that design into the individual files that developers will need. It is up to the developers to then write the necessary CSS and HTML to implement that visual design.

This method works well if your developers have strong front-end skills and can create well-structured CSS and clean HTML. It works well when your development team is small, but can fall apart in larger development teams because the developers may not maintain consistent CSS practices, and the design aspects of the code may get messy.

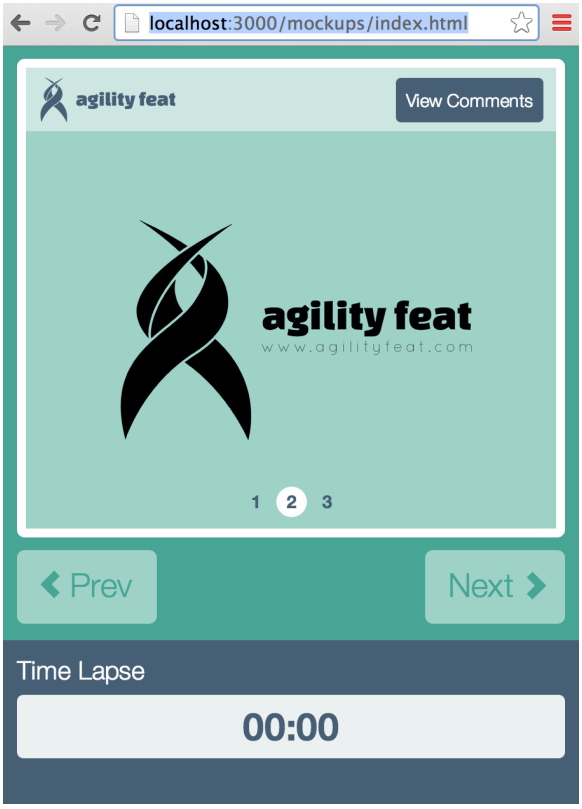
Other development teams prefer to have the visual designer implement their own designs, or they have a single front-end developer do it before passing the implementation off to the development team. This method works well when your visual designer also has front-end coding skills, since they can ensure that best practices in the HTML and CSS are followed.

This second method is what our team prefers, especially because our visual designer Daniel Phillips is also excellent at front-end coding and HTML/CSS implementation. Because of that, Daniel has already implemented the most important aspects of the design for us.

For your convenience, we’ve put the HTML mockups in a folder called “mockups”, under the public directory of the application. That means that (after downloading the code at the end of this chapter), you can start your local node server (using “node server”), and then navigate to the mockups here:

http://localhost:3000/mockups/index.html

Now you're seeing the default presenter view. Resize your browser to a smaller size, and you'll see the presenter's mobile view, complete with the session timer beneath the slides.



Similarly, you can also see the attendee view from your local site by navigating to:

http://localhost:3000/mockups/spectators.html

If you want to experiment with changing the CSS or HTML for your version of the webinar tool, then these mockups may serve as a good sandbox for you to work in.

You can also grab the mockups directly from github here:

https://github.com/agilityfeat/webinar-tool/tree/designsetup/public/mockups

5.7. Breaking down the front end code

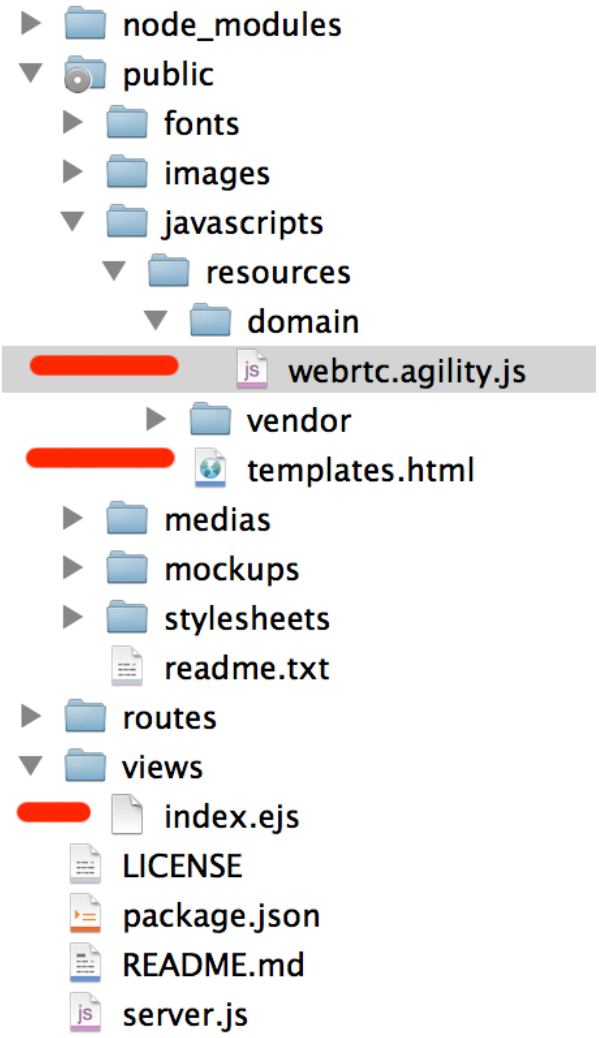
Ok, if you're ready to get a copy of the webinar-tool that contains all the design artifacts, you should pull down this branch of our code:

https://github.com/agilityfeat/webinar-tool/tree/designsetup

We have added in a lot of code since the last chapter, and we won't be able to explain all of it line by line. As mentioned previously, this is not a book about visual design specifically, so we won't be able to explain all the CSS and responsive techniques being used here. But with a high level overview, you should know where to look when you want to change the foundation we've given you.

Most of our additions are in the public directory, where we have added in a large number of images, fonts, the mockups directory previously mentioned, and our stylesheets. Many of the images and styles contained in those won't be used until later parts of this book.

The most important files for us to examine for now are going to be our main javascript file (webrtc.agility.js), a templates file, and an updated version of our main view, index.ejs. Those files are highlighted in the directory structure shown below.



Updated directory structure after adding in design artifacts, mainly under the public directory. Not all new files are shown.

First, let's take a look at the updated index.ejs file in our views directory.

```
<!doctype html>
<html lang="en">
 <head>
  <meta charset="utf-8">
  <title>Webinar Tool RealTimeWeb.co</title>
  <meta name="viewport"
     content="width=device-width, initial-scale=1.0, user-scalable = no">
  <meta name="format-detection" content="telephone=no">
  <link rel="stylesheet" type="text/css" href="stylesheets/bootstrap.min.css" />
  <link rel="stylesheet" type="text/css" href="stylesheets/bootstrap-theme.css"/>
  <link rel="stylesheet" type="text/css" href="stylesheets/main.css" />
  <link rel="stylesheet" type="text/css" href="stylesheets/media-queries.css" />
  <link rel="stylesheet" type="text/css" href="stylesheets/line_graph.css" />
  <link rel="stylesheet" type="text/css" href="stylesheets/tipsy.css" />
  <link rel="shortcut icon" href="img/favicon.ico">
 </head>
 <body style="">
  <div class="top-container" id="content">
   <!--RENDERED CONTENT HERE -->
  </div>
  <script src="javascripts/resources/vendor/jquery.min.js"></script>
  <script src="javascripts/resources/vendor/easing.js"></script>
  <script src="javascripts/resources/vendor/underscore-1.5.2.js"></script>
  <script src="javascripts/resources/vendor/bootstrap.min.js"></script>
  <script src="javascripts/resources/vendor/d3.v3.min.js"></script>
  <script src="javascripts/resources/vendor/jquery.tipsy.js"></script>
  <script src="javascripts/resources/vendor/line_graph.js"></script>
  <script src="javascripts/resources/vendor/timer.js"></script>
  <script src="javascripts/resources/domain/webrtc.agility.js"></script>
 </body>
</html>
```

Notice that we have added in a number of stylesheets at the top, and a number of javascript files at the bottom of the page. Most of them are needed now just to get bootstrap working with our base styles.

The other important change to notice is the addition of the div tag with id “content”, which contains the text “Rendered Content Here.” This is where the main layout code of our application will be inserted at run time.

Next, we’re going to look at our main javascript file. This is where the code is that will render content into that div tag. Look for this file in your local project:

/public/javascripts/resources/domain/webrtc.agility.js

Tip

You will find a number of things in the webrtc.agility.js file, and we will be adding a lot more to it in the future. So get used to visiting this file, this is where most of the logic will live!

This file is the central processor of our application, so let’s start by looking at the init() function:

```
init : function(){
        var self = agility_webrtc;
        self.loadTemplates({
                templates_url : "javascripts/resources/templates.html"
        }, function(){
                self.showPresentationScreen();
        })
}
```

The initialization function calls a method to load in some html templates that we have defined. That method, shown below, will then add the contents of a templates file to the body of our main page.

```
loadTemplates : function(options, callback){
        $("#templatesContainer").empty().remove();
        $('<div id="templatesContainer"></div>').appendTo('body');
        $('#templatesContainer').load(("javascripts/resources/templates.html?r="
          + Date.now()),
        function(){
                if(typeof callback === "function"){
                        callback();
                }
        })
}
```

Next the init() function called a method to show the main presentation screen. That method, shown below, in turn renders the template for our main presentation screen.

```
showPresentationScreen : function(){
        agility_webrtc.render({
                container       : "#content",
                template        : "#presentation_template",
                data            : {
                        user            : agility_webrtc.currentUser,
                        slide_moods : agility_webrtc.slide_moods,
                        slides  : agility_webrtc.slide_pics
                }
        })
}
```

In this method we see that the render method is told to attached the presentation_template from the templates file to the content div tag in the index.ejs file.

In this way, our base templates for the main presentation view are in the templates file:

/public/javascripts/resources/templates.html

We've chosen to store this file in the javascript resources directory because we're going to manipulate it from our main javascript file, webrtc.agility.js.

5.8. Cheat sheet to the app design files

At this point, you should have an idea of where to go if you want to change anything about the design in this application. Even if you don't want to change anything, it was worth your time to familiarize yourself with what we're doing in this app because we're going to add to it as the book goes on. Each new feature will require the addition of some html elements.

So here's your cheat sheet of where to look when you want to change something:

File	Purpose
Public/stylesheets/media-queries.css	CSS for the responsive design
Public/stylesheets/main.css	General CSS for the application
Public/javascripts/domain/webrtc.agility.js	Core javascript for the application
Public/javascripts/templates.html	Base view that will contain the application
Views/index.ejs	HTML templates for each part of the application

As a reminder, this is the branch in GitHub to use in order to get all the design assets you need to run the application.

https://github.com/agilityfeat/webinar-tool/tree/designsetup

Tip

I recommend that you use the branch above as your starting point for the rest of the book because it has all the design assets you will need already integrated into the project. Consider any Node.js code you wrote on your own before this chapter to have been a learning exercise, and use ours from here on out.

Assuming you use our code, then run “node server” and you should see this:

6. A comparison of WebRTC and Publish/Subscribe patterns

In this book, there are two major real-time technologies we are working with:

1. Real-Time Messaging using a Publish/Subscribe pattern

2. Real-Time Communications using WebRTC

6.1. WebRTC and Publish/Subscribe messaging are two different things

These two concepts are both important, and both very useful. But they are two separate, albeit complementary, concepts. When we speak to user groups or customers about what we do, we often need to introduce both concepts to people for the first time, and because we are introducing them together, we have found that people get easily confused which one is which.

So before we dive into a primer on real-time messaging using a publish/subscribe pattern, we need to explain the difference between the two technologies you will encounter in this book.

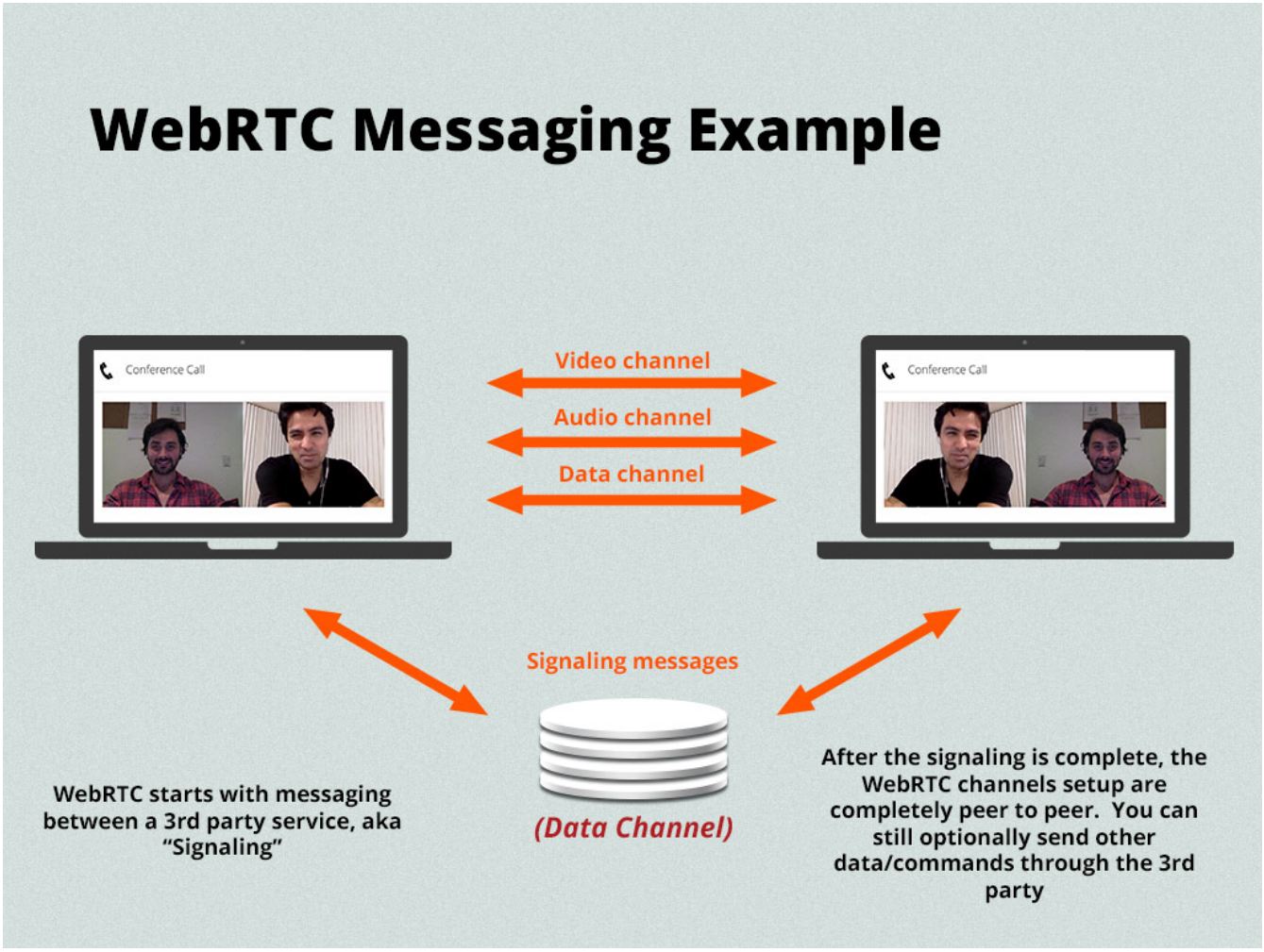
6.2. A brief overview of WebRTC

Let's start by explaining WebRTC very briefly – we'll go into this topic in more depth in a future chapter. WebRTC stands for Web Real-Time Communications. The shortest way to describe it is this:

WebRTC is an HTML5 standard for encrypted peer-to-peer (P2P) exchange of video, audio, and data, directly between two clients.

Although you can do a lot more with WebRTC than build video chat applications, the simplest way to think of WebRTC is this is how you build your own personal version of a Skype client, directly in your browser application. The most touted feature of WebRTC is that it is built directly into the browser, and you only need to use HTML5 and Javascript to use it. The users of your application do not need to download any plugins (like was necessary to build similar applications using Flash).

The next figure shows a diagram of how this works.



Imagine that you are building a video chat application in the browser. In the figure above, we see Daniel and Allan chatting in a browser client. For them to talk, both users need to be on the same website at the same time. There is no way using WebRTC alone that you can call a user who is not on your same web page.

For Allan to call Daniel, there must be a connection setup between their two browsers. We'll talk more in a later chapter about how this is done, but for the moment, just accept that this is called "signaling" in the WebRTC world, and it must be handled by a third party server. This 3rd party server could be your web server, or you could use a paid-service to handle the signaling for you. Again, we'll come back to that later, but this is important to be aware of, because Allan can't call Daniel's browser out of the blue, there is some handshaking that needs to happen first.

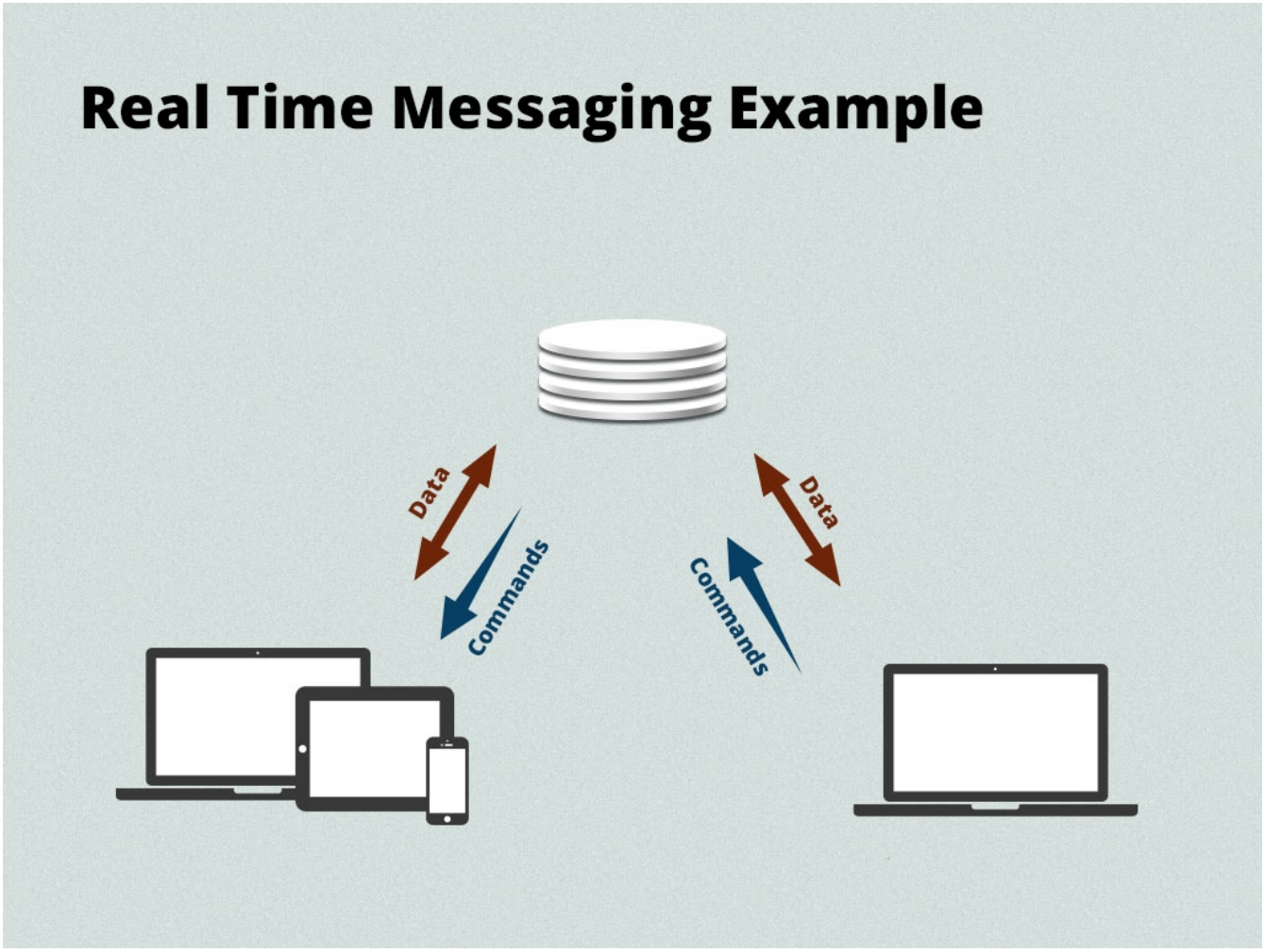
The key point about WebRTC is what happens after the handshaking or signaling is complete. Once Allan and Daniel have established a browser connection, the rest of their conversation and data exchange is all done in a Peer-to-Peer encrypted connection. The signaling server never receives any of the video, audio, or data traffic between the two browsers.

For now, that's all you need to know about WebRTC. We'll see later how you can do more than just video chat with it, but for now just know that until we get to the video and audio aspects of this webinar tool, we are not using WebRTC yet.

6.3. A brief overview of Publish/Subscribe real-time messaging

The first things we are going to build in our webinar application don't need WebRTC. We are going to synchronize the changing of slides across logged in clients, send chat comments and cast votes. All of these will be done using a publish/subscribe model of real-time messaging.

In a publish/subscribe model, there is never a peer-to-peer connection made, and we are only exchanging data. There is no video or audio component to it. The next figure shows how this communication works.



In this model, we have a central messaging server (shown as the discs at the top of the diagram) which is responsible for relaying all messages between different users of our application.

In our application, imagine that the single laptop on the right is our Presenter in the webinar. On the left of the diagram are all of our webinar attendees, and they might be using laptops, tablets, or smartphones to view the webinar.

The presenter's laptop will send data and commands to all the webinar attendees, mainly about the current slide being viewed.

The webinar attendees will receive those commands (i.e., "advance to slide #2"), and they will also send data back to the presenter for display to everyone. When a webinar attendee makes a comment

or casts a vote, that data will be “published” to the central messaging server, and anyone who is “subscribed” to that channel will receive a copy of the message. This is how the comments and votes will be displayed on the presenter’s screen, as well as on other attendee’s screens.

In our application, even though we might think of the presenter as someone with special privileges, that is not really the case. From the messaging server’s perspective, they are all just publishers and subscribers to the same messaging channel on the server. Both presenters and attendees have data to be published and shared with others on the same channel.

6.4. WebRTC and Publish/Subscribe messaging are complementary

We could build an application based purely on a publish/subscribe model. A basic text chat application is a common example. Everyone is just sending simple text data to a channel that others subscribe to and receive updates from. Another example is perhaps a real-time data dashboard of stock quotes. As new stock prices come in, they are published to a data channel. Everyone subscribed to that channel receives an update about the stock price at the same time.

A WebRTC application, using video and audio chat, is likely to use both WebRTC and some other messaging protocol like publish/subscribe. While this is not a hard requirement, it is very common that the two will complement each other. This is because in addition to the video and audio chat, you may want other features like:

- Exchanging text messages between multiple participants in a video chat (could be done using the WebRTC DataChannel but is simpler in a publish/subscribe model)
- A listing of all users available to chat with, and whether they are available for a call or not

These common features are better implemented using a publish/subscribe model, because it scales better than WebRTC. Since WebRTC is peer-to-peer, that means that a separate connection has to be established between every user in a call. This puts a practical limit on calls of perhaps 8 or 10 people at the most, probably less, depending on your bandwidth.

A publish/subscribe model scales indefinitely however. For someone publishing a piece of data like a chat message, it causes no extra burden on them if one or a thousand people are subscribed to that data channel. That is the messaging server’s problem to get the message pushed out to every subscriber, and this is much more manageable than direct P2P connections between all WebRTC users.

WebRTC and Publish/Subscribe messaging serve different purposes and different needs, but they are often complementary and you are likely to need both. That’s why we cover both in this book, but it’s important for you to understand the difference between the two concepts.

6.5. The differences between WebRTC and Publish/Subscribe messaging

WebRTC:

- Peer to Peer (after initial handshaking)

- Encrypted
- Can pass video, audio, or data
- Does not scale well due to Peer to Peer nature

Publish/Subscribe messaging:

- Uses an intermediate messaging server to relay all messages between publishers and subscribers
- Not encrypted by default, though you could encrypt your messages on your own if desired
- Only passes text data typically, no native support for video or audio
- Scales well, there is no extra load for the number of subscribers (or better said, there is no extra load on publishers. Your intermediate messaging server will still have a breaking point of how much load it can handle).

6.6. A final note on real-time nomenclature

Tip

From here forward, we will generally use the term “real-time messaging” to refer to anything using a publish/subscribe model. When speaking of real-time communications, video or audio chat, we will exclusively use the term “WebRTC.”

It’s worth pointing out that publish/subscribe models are not the only way to do real-time messaging, but that is what we will mean in the rest of this book.

7. Real Time Messaging Primer (Publish/ Subscribe pattern)

If you read the previous chapter, then you have a pretty good idea already how a publish/subscribe pattern for real-time messaging is going to work. Let’s take a quick look at a simple scenario.

7.1. Single vs multi-channel messaging

In the simplest form of messaging, you have multiple clients attached to a single channel. When one of them publishes a message to the messaging server, the messaging server then relays that message to anyone else subscribed to the same channel.

Let’s imagine a more complex, but very common scenario next. Our clients want to be able to subscribe to a variety of channels, and not everyone will subscribe to the same channels. The simplest example of this is chat rooms on a chat server, or different conversations in Skype. Each user is in different conversations, and only the users subscribed to that channel can see the conversations happening in it.

How do you control who can see the different conversations and subscribe to channels? That is up to your application. The messaging server by itself doesn’t really care, it’s basically just relaying messages based on the channel names you setup, and so it’s up to your application to handle any security around knowing what channels are available and choosing who can listen in on them.

7.2. Multi-channel messaging vs message types

Another way to handle different types of content in your messaging server is to define your own message types and send those as part of the content. It’s common message that you are not just going to send plain unstructured text, but that you will define your own message format to send to the server. A simple way to do that is with a JSON structure like the following example:

```
{
        “message_type” : “chat”,
        “chat_text” : “Hello world, this is my message”
}
```

In this case we are just sending a chat message to post. In our application, we might also send a command to advance to the next slide over the same channel using a different message type:

```
{
        “message_type” : “advance_slide”,
        “slide” : “2”
}
```

Notice that in both cases, we are just sending clear text to the messaging server. But we can add our own structure to that clear text using JSON (or XML if you prefer) so that we can encapsulate data.

In our application, that means we can’t just blindly do the same thing with each message that comes in, we need to first examine the message type as this pseudo code shows:

```
select case @message[“message_type”] {
        case “chat”:
                post_chat_message(@message[“chat_text”]);
                break;
        case “advance_slide”:
                advance_slide(@message[“slide”]);
                break;
        case else:
                //do nothing perhaps, or throw an error
}
```

This allows you to define your own message structure, and as long as every client to your application knows how to handle each message type that can come through, then this is a very flexible model to use. It works best though for messaging channels where all clients in that channel will want to see all the different message types. That is the case for our webinar application, and so we are going to use a single channel in our messaging server and send different message types over the wire.

In other cases, like private chat rooms, you don’t want to try and use a message type to delineate private messages from public, because you’re still sending those private messages to every client, even if your client side code chooses to ignore them. At a minimum, this is wasted bandwidth. It may also be a privacy or security concern. In those “private chat room” type of scenarios, it’s best to use entirely unique channels on the messaging server – that’s why they are there.

In many cases, you will use a combination of multiple channels and defining your own message types within that channel. It’s up to you. In our webinar application, we’re going to keep it simple and just use a single channel, with different message types.

One last thing to emphasize on the message types you define, remember that while I’ve used a simple “chat room” example to explain the channels vs message types concept, this has much wider applications than just text chat. We are going to use our channel for relaying comments, votes, and presentation slide commands between all the clients to our webinar. You might also use these messages in another application for exchanging custom data like stock prices or sensor measurements, or commands to control a robot or turn on the lights in your house. The possibilities are only limited by your imagination and what you can interface to the messaging server.

7.3. Choosing a real-time messaging server

Okay, so we’ve decided that for our webinar application we will just use a single channel, and we’ll define various message types as we go and structure them using JSON. Next we need to decide on an actual real-time messaging server to use. This is the intermediary server that can be used between our clients.

There are many different options for you to consider, and so the first constraint to consider is whether you want to manage the server yourself, or pay someone else to host and manage the server for you.

7.4. Standing up your own open source messaging server

If you like standing up your own servers and managing them, and cost is a concern, then you should consider using an open source messaging server like Faye or ejabberd. This is also a good idea if you have particular security constraints that prevent you from using a third party service.

These options can work great for you, but I strongly recommend that before committing to one, you read the documentation carefully and consider browser support. Most real-time messaging software relies on WebSockets for communication to the server. Older browsers, particularly Internet Explorer, may not support WebSocket connections. If you have to support older browsers, then you need to make sure that the server you choose supports that browser and will automatically fall back to a less efficient long polling implementation when WebSockets are not available.

Even if the documentation says that support for older browsers exists, we recommend that you stand up a test server and check it out yourself. We have run into problems in the past with a server that claimed to support IE, but in reality we could not get it to work reliably. Check that out before you commit to a particular platform.

This is also a good time to make sure that the test server you've setup will work in your specific hosting configuration. We ran into a problem once with a Faye messaging server hosted on AWS that wouldn't work reliably with our Ruby on Rails website hosted on Heroku. While I think this would no longer be the case, it's better to confirm before you code too far to a specific messaging server API.

7.5. Using a paid publish/subscribe server

If the prospect of hosting and maintaining the real-time messaging server is not worth it for you, then there are a number of paid providers you can look into. This is the route that we have gone with our projects.

We have had very good experiences with a service called PubNub.com. We've never received any compensation from them for saying this [5], but we are fanboys. They have very low latency (less than 0.25 seconds) and can process millions of messages per second [6]. Scalability is not an issue, and they also offer a wide variety of developer SDK's. Perhaps most importantly, when we've had questions, their customer service has been excellent.

We're going to use PubNub in our examples and sample code primarily in this book, and you can setup a free sandbox account that should be sufficient for what we need for the real-time messaging part of this project, although you will probably want to upgrade to a paid account later for the WebRTC part of this project.

7.6. How PubNub sends and receives messages

Once you have a PubNub account and have included the appropriate javascript in your project (more on this in a minute), then to subscribe to a channel is pretty simple. Basically you just tell PubNub what channel you're interested in, and then supply it with a method to call back to whenever data is available.

```
PUBNUB.subscribe({
        channel : “my_channel”,
        message : function(m){alert(m)}
})
```

In the simple example above, we are asking PubNub to send us all data on the channel named "my_channel". When there is data ready to send us, we have told PubNub what method to call. In this

[5]Ok, that's not entirely true. Once at the HTML5DevConf some PubNub representatives gave me a "real-time Frisbee" and a real-time bottle opener. I applaud them for the sense of humor, but I assure you, my dear reader, that this alone did not buy our loyalty.
[6]As I write this, their site claims 3 million messages are being sent to a 100 million devices.

case, we are just calling an anonymous function which will just put the message into a javascript alert popup in the browser.

To publish a message to this channel, the code is also very simple:

```
PUBNUB.publish({
        channel : “my_channel”,
        message : “Howdy doody!”
})
```

In the next section we’ll setup PubNub in our project and make a simple example. But if you would like to play with PubNub a little more before you setup an account, then you should check out their tutorials. These can be implemented in a browser using a demo key before you have your own account. A simple example to publish and subscribe to real-time message channels is available here:

http://www.pubnub.com/docs/javascript/tutorial/data-push.html

7.7. Setting up PubNub in our project

If you don’t already have one, you’ll first need to setup a free account on PubNub.com. Next up is to reference the pubnub javascript library in our project. You may want to refer to the javascript quickstart tutorial at PubNub, which you can currently find here:

http://www.pubnub.com/docs/javascript/javascript-sdk.html

The simplest way to include pubnub in your project is by referencing the latest copy of the code on their content servers. Just drop this script tag near the bottom of the views/index.ejs file, along with the other javascript includes that we already created.

```
<script src="http://cdn.pubnub.com/pubnub.min.js"></script>
```

It’s important that this be listed in views/index.ejs above our main application javascript file of webrtc.agility.js.

Now we need to take our pubnub keys and put them into the application’s main javascript file – public/javascripts/resources/domain/webrtc.agility.js

```
credentials : {
        publish_key     : 'your publish key from the pub nub admin',
        subscribe_key   : 'your subscribe key from the pub nub admin'
},
```

This should go above the init function definition, along with the other variables defined with a scope defined in the agility_webrtc structure.

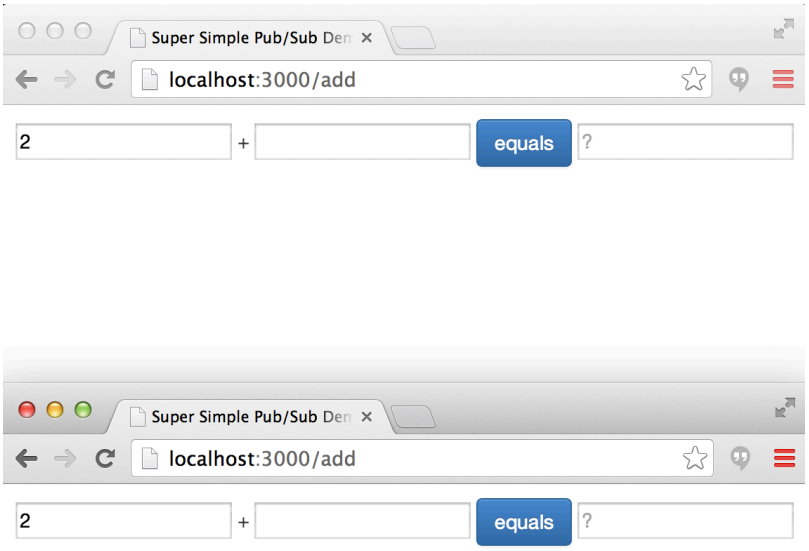
7.8. A simple PubNub example

Before we proceed with building in the real-time messaging into our webinar app, let’s look at a simple example. Our example will be a form for adding two numbers. As you change the two numbers and press the “Add” button, someone else looking at the same page will see the same numbers you are entering, and also see the result in their browser window.

The code for this chapter is in a branch called “pubsubsetup” and is accessible directly on github here:

https://github.com/agilityfeat/webinar-tool/tree/pubsubsetup

We’ll walk you through the code in a moment, but first let’s look at the application in action. If you run the app in your local environment and open it in two browsers you’ll see something like this:



We’ve set this up as an additional route for the “/add” URL to point to the “views/add.ejs” view file. To bring it up in your browser, visit:

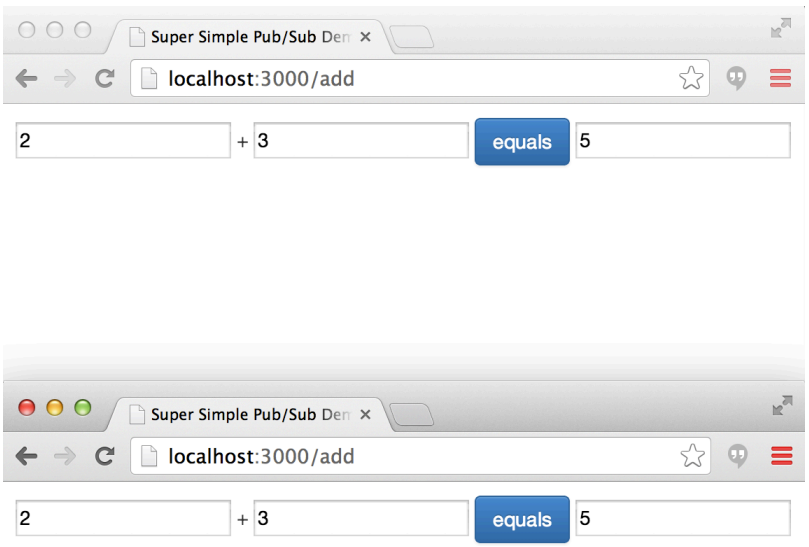
http://localhost:3000/add

Note that you’ll have to add in your own PubNub publish and subscribe keys to the add.ejs file before this will work in your local environment.

In the image above, we’ve opened two browsers side by side and pointed them both to the add page so that we can see the magic happening. If you enter a number “2” in the first box, then as soon as you step off that box, it will appear in the same box in the second window.

The same is true in the second part of our addition form. Enter a number “3” in the second box, and as soon as you tab off that box you’ll see the same number appear in the other window. It works in either direction; you don’t need to keep entering them in the same window.

Finally, press the “Equals” button to have our form calculate the answer and display it. Note that the answer also appears in the other browser at the same time. Go ahead, enter the super complex formula 2+3 and press Equals, and you’ll see an output like this in your browsers:



Okay, I admit that we're not about to revolutionize online education with this little real-time addition tool, but maybe you can if you build on top of it. Why not use the same real-time principles to build an online math tool that allows remote teachers to see students work out a problem as they do it. After reading the rest of this book, you could even integrate some video/audio chat into it so they can talk the student through tough problems. I just gave you a business idea that's worth much more than the price of this book!

7.9. Implementing the example

What we've done here is setup a single messaging channel called "add_example", and we are passing JSON messages to it with the numbers to display in a "message" field, and the box that they should be populated in goes in another field we are calling "message_type". Here's an example JSON message:

```
{
        "message_type":"operand1",
        "message":"2"
}
```

Simple, right? This message will tell all browsers subscribed to the "add_example" channel to update the first operand text box with the number 2.

Here is the code for actually sending that JSON message to the channel:

```
function publish(m) {
         console.log("sending message: " + m);
     pubnub.publish({
         channel : "add_example",
         message : m
     });
 };
```

And anyone who wants to subscribe to receive these updates will use a method like this:

```
pubnub.subscribe({
     channel : "add_example",
     message : function(m){
                m = JSON.parse(m);
                console.log("received message: " + m);
                document.getElementById(m.message_type).value = m.message;
        },
 });
```

Note that in the subscription code, we are going to parse the message we receive as JSON. We have implemented our layout to pass the id of the field we want to populate as the “message_type”, which makes this method very simple. The actual number to put in the field was stored in the “message” property.

It’s a pretty straightforward example, and we’ve kept all the code you need for it in the add.ejs file for you to play with. Here’s the full code in that file so you can also see how we trigger the javascript events.

```
<!doctype html>
<html lang="en">
 <head>
  <meta charset="utf-8">
  <title>Super Simple Pub/Sub Demo</title>
  <link rel="stylesheet" type="text/css"
    href="stylesheets/bootstrap.min.css" />
  <link rel="stylesheet" type="text/css"
    href="stylesheets/bootstrap-theme.css" />
 </head>
 <body>
  <script src="//cdn.pubnub.com/pubnub.js"></script>
  <script>
   var pubnub = PUBNUB.init({
    publish_key   : 'your pub key',
    subscribe_key : 'your sub key'
   });
   pubnub.subscribe({
    channel : "add_example",
    message : function(m){
     m = JSON.parse(m);
     console.log("received message: " + m);
     document.getElementById(m.message_type).value = m.message;
    },
   });
```

```
   //Continued from previous page
   function publish(m) {
    console.log("sending message: " + m);
    pubnub.publish({
     channel : "add_example",
     message : m
    });
   };
   function add() {
    return parseFloat(document.getElementById('operand1').value) +
         parseFloat(document.getElementById('operand2').value);
   };
   function create_message(msg, type) {
    return '{ "message_type":"' + type + '", "message":"' + msg + '" }';
   };
  </script>
  <div class="container" style="margin-top: 10px;">
   <div class="row">
    <div class="span6">
    <form>
     <div class="controls controls-row">
      <input id="operand1" type="text" class="span2"
                onchange="publish(create_message(
                document.getElementById('operand1').value,'operand1'))">
       +
      <input id="operand2" type="text" class="span2"
            onchange="publish(create_message(
                document.getElementById('operand2').value,'operand2'))">
      <button id="sendMessage" type="button" value="Add and Send!"
            onclick="publish(create_message(add(),'answer'))"
                class="btn btn-primary input-medium">equals </button>
      <input id="answer" type="text" class="span2" placeholder="?">
     </div>
     <div class="controls">
     </div>
    </form>
    </div>
   </div>
  </div>
</body>
</html>
```

As a reminder, all of the code for this chapter is located here:

https://github.com/agilityfeat/webinar-tool/tree/pubsubsetup

Now that you should be comfortable with real-time messaging, let’s start implementing it into our webinar tool!

8. Coding: Synchronized Slides

8.1. Code Samples

Prerequisite code for this chapter:

https://github.com/agilityfeat/webinar-tool/tree/designsetup

Completed code for this chapter:

https://github.com/agilityfeat/webinar-tool/tree/syncsetup

8.2. Introduction

In this chapter we are going to focus on synchronizing the slides between presenters and attendees. This means that when the presenter presses the Next or Previous buttons, or the number of a particular slide they want to display, then that slide should change on the presenter's screen and all attendees' screens using real-time messaging.

To see the full code for this chapter, check out this branch of the webinar-tool application:

https://github.com/agilityfeat/webinar-tool/tree/syncsetup

8.3. Setting up the Presenter "login"

The first thing that we need to do is determine if a webinar user is the presenter or an attendee. Only presenters will have the power to change slides. Only attendees have the power to post comments.

If you're going to take the code from this book and make a proper webinar tool, then you should put in some real security with actual authentication. But I've always been a big fan of not bothering with the mundane things like logins until you have to, because logins are not that valuable by themselves.

Since this book is not about user authentication models and I don't want to waste your time with that, we're going to do a cheap trick to determine if someone is a presenter or not.

After this chapter, if you are an attendee, you access the application locally this way:

http://localhost:3000/

If you are a presenter, you should access it this way:

http://localhost:3000/presenter

That's it! It's not exactly secure, but it's good enough for our learning purposes in this book. We're just setting this up in the routes for our application:

```
app.get('/', function(req, res){
                res.render('index.ejs');
        })
app.get('/presenter', function(req, res){
                res.render('index.ejs');
        })
```

In the routes/routes.ejs file, we've just added an additional route for '/presenter' underneath the root URL route. Both will render the index.ejs file.

Then in our webrtc.agility.js file, we've added a variable "isPresenter" to the agility_webrtc definition, and then in the init() method:

```
init : function(){
        var self = agility_webrtc;
        isPresenter = (document.URL.indexOf("presenter") > 0);
```

Now we can access the isPresenter variable anywhere in the layout to determine which view we should show the current user, and what functionality they can use.

8.4. Changing the responsive layout based on Presenter/Attendee view

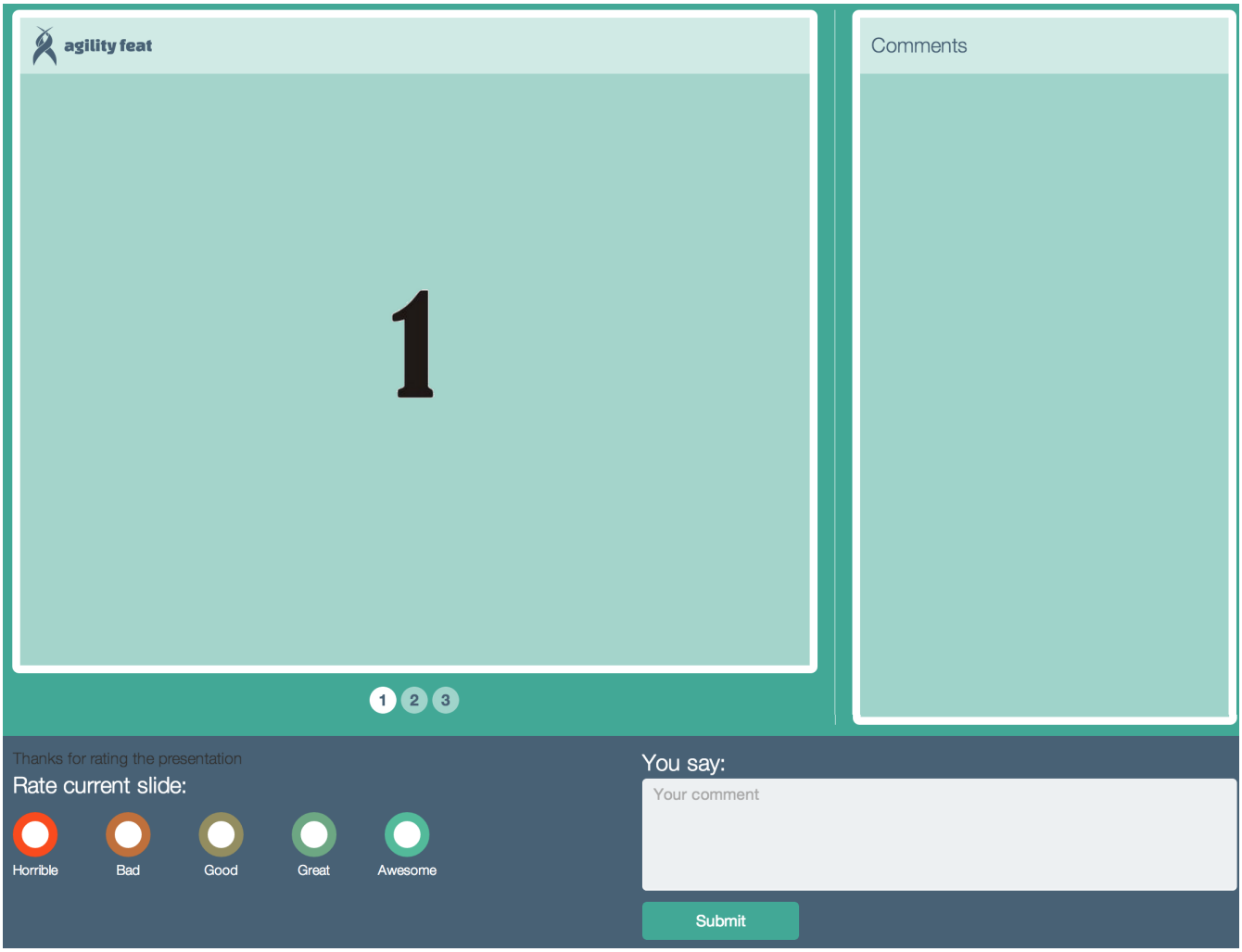
If you recall what the webinar tool looked like after the design chapter, you were seeing only the presenter's view. The next/previous buttons should not be visible to attendees, and the circle icons for each slide should only have a click event for the presenter. In the templates.html file, this is handled just by checking that isPresenter variable:

```
<div class="sliderControls">
        <% if(isPresenter){%>
                <div class="prevSlide control" data-slide="prev">
                        <span class="glyphicon glyphicon-chevron-left"></span>
                        <span class="textSpan">Prev</span></div>
                <div class="nextSlide control" data-slide="next">
                        <span class="textSpan">Next</span>
                        <span class="glyphicon glyphicon-chevron-right"></span>
                </div>
        <%}%>
        <ul class="slideCount">
                <% _.each(slides, function(slide, index){%>
                        <li
                        class = "<%= index == 0 ? 'active' : ''%>"
                        <% if(isPresenter){%>
                                data-target="#presentation-carousel"
                                  data-slide-to="<%= index %>"
                        <%}%>
                        ><%= index+ 1%></li>
                <%});%>
        </ul>
</div>
```

In addition to the slide controls, we should also go ahead and change the rest of the template for the upcoming chapters where we are dealing with attendee voting and comments. Those controls should only be visible to attendees, not to presenters. That is also handled in the templates.html file with our isPresenter variable:

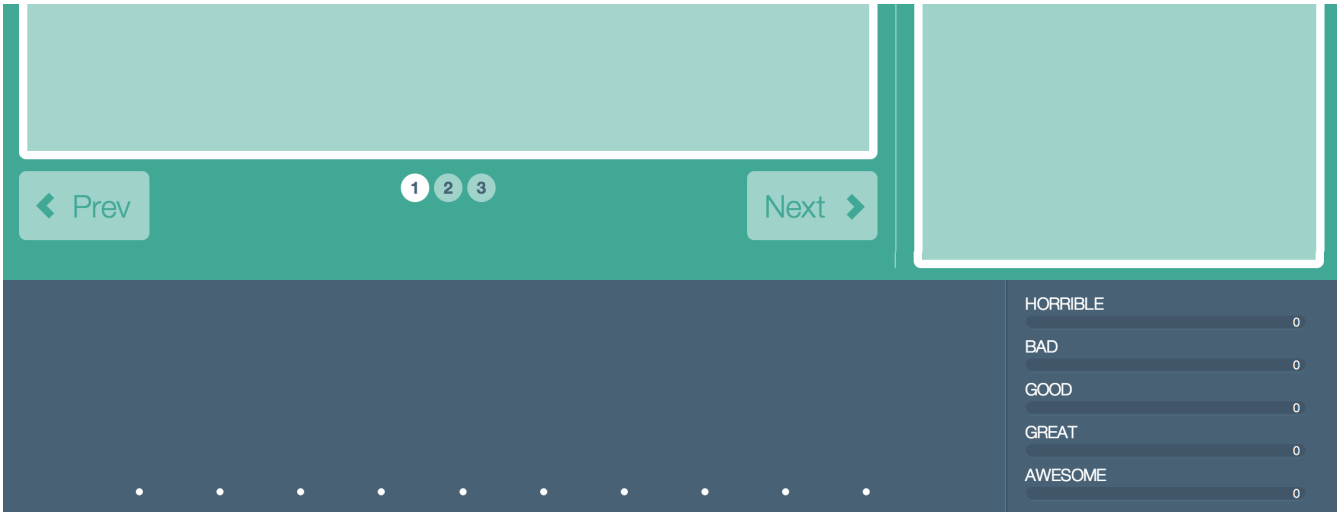
```
<footer>
  <div class="row">
    <% if( isPresenter ){%>
      <div class="col-sm-9 graphWrap">
        ... controls for slide numbers
      </div>
      <div class="col-sm-3 colGraphWrap bargraph">
          <% _.each(slide_moods, function(mood){%>
            <div class="graphLabel" data-mood-name="<%= mood.name %>">
              <span class="name"><%= mood.name.toUpperCase() %></span>
              <div class="barGraphBg">
                <div class="bar bar<%= mood.name %>"></div>
                <span class="mood_count"><%= mood.count %></span>
              </div>
            </div>
          <%});%>
      </div>
      <div class="col-sm-1 timeLapseWrap">
        <span>Time Lapse</span>
        <div class="timer" id="timerWrap">
          <span class="hour">00</span>:<span class="minute">00</span>:
                    <span class="second">00</span>
        </div>
      </div>
    <%} else {%>
      <div class="col-sm-6">
        <div class="rateSlideWrap">
          <div class="thanks_for_rating">Thanks for rating the presentation</div>
        <span class="title pull-left">Rate current slide:</span>
        <div class="rateOptionsWrap">
          <div class="floatClear"></div>
          <% _.each(slide_moods, function(mood){%>
            <span class="rateOption rate<%= mood.name %>"
              data-slide-mood="<%= mood.name %>">
            <span class="inneCircle"></span>
            <span class="label"><%= mood.name %></span>
            </span>
          <%});%>
        </div>
        </div>
      </div>
      <div class="col-sm-6">
        <span class="showCommentBox">
          Add Comments
        </span>
        <div class="commentSlideWrap">
        <span>
          You say:
        </span>
        <textarea class="commentsHere" placeholder="Your comment"></textarea>
        <button type="button" id="btn_send_message"
                  class="btn btn-presentation">Submit</button>
        <span class="hideCommentBox">Cancel</span>
        </div>
      </div>
    <%}%>
  </div>
</footer>
```

This is what the Attendee's view should look like now:



Note that the "Rate Current Slide" and comment box controls are visible to the attendees, but the Previous/Next buttons are now hidden.

For the presenter, the bottom of the screen should now look like:



Note that the Previous/Next buttons are back, and along the bottom we have some placeholder html for the eventual voting result displays.

8.5. Changing slides

Now that the controls are all placed on the page properly for our presenter, it's time to hook them up to some code to change the slides. In our webrtc.agility.js file, we have added two functions to a setBinds method.

The first method will bind a function to the "onclick" event of any control with the class ".control" attached to it. If you look closely at the layout from our templates.html file above, you'll see that only the Previous and Next buttons have the css class control attached to them:

```
<% if(isPresenter){%>
        <div class="prevSlide control" data-slide="prev">
                <span class="glyphicon glyphicon-chevron-left"></span>
                <span class="textSpan">Prev</span></div>
        <div class="nextSlide control" data-slide="next">
                <span class="textSpan">Next</span>
                <span class="glyphicon glyphicon-chevron-right"></span>
        </div>
<%}%>
```

Here is the first method for binding the Previous and Next buttons:

```
setBinds : function(){
  $(document).on("click", ".control", function(e){
    e.preventDefault();
    e.stopPropagation();
    var total_slides = $(".slideCount li").length;
    var is_next = $(this).is(".nextSlide");
    var slide_to = $(".slideCount li.active").data("slide-to");
    if(slide_to != null){
      slide_to = (is_next ? (slide_to + 1) : (slide_to - 1));
    } else {
      slide_to = 0;
    }
    slide_to = slide_to < 0 ? (total_slides) :
          (slide_to === total_slides ? 0 : slide_to);
    slide_to = slide_to ===
          total_slides ? total_slides - 1 : slide_to;
    slide_to = slide_to < 0 ? 0 : slide_to;
    $(".slideCount li").removeClass("active");
    $('.slideCount li[data-slide-to="' + slide_to
          + '"]').addClass("active");
    $(".slider").carousel(slide_to);
    agility_webrtc.currentUser.publish({
      channel: agility_webrtc.channelName,
      message: {
        type: "SLIDE",
        options: { slide: slide_to }
      }
    });
  });
  return this;
}
```

Notice that this function sets an "is_next" variable based on whether or not the button that generated the "onclick" event also has the "nextSlide class attached to it. In this way we can use one event handler for both buttons and just determine the command based on attach CSS classes. If you wanted

to add more buttons to the UI for some reason, this allows you to do that only in the templates file without additional code bindings needed (although we wouldn't recommend filling the screen with extra controls!).

The rest of the method just determines the right number for the next slide, and then sets the image carousel to use that number slide.

There is one more function we want to add to the setBinds method. In addition to the Previous and Next buttons, we have also put a set of numbered circles on the layout inbetween the two buttons. Each numbered circle represents a slide in the presentation deck (more on where those are stored later).

To make those circles bind to a function for navigating directly to that slide, we also add this function to the setBinds method:

```
$(document).on("click",".slideCount li:not(.active)",
  function(e){
        e.preventDefault();
        e.stopPropagation();
        if( isPresenter ){
                var el = $(e.target);
                var slide_to = Number($(el).data("slide-to"));
                $(".slideCount li").removeClass("active");
                $('.slideCount li[data-slide-to="' + slide_to +
                  '"]').addClass("active");
                agility_webrtc.currentUser.publish({
                        channel: agility_webrtc.channelName,
                                message: {
                                type: "SLIDE",
                                options: {slide: slide_to}
                        }
                });
        }
});
```

The code is simpler in this case because we know exactly what number slide we want to navigate to.

8.6. Synchronizing the slides across clients

There's one last thing we haven't explained yet about the code above, but you probably already noticed it. When you hit any of the slide navigation controls, the function ends with a method like this:

```
agility_webrtc.currentUser.publish({
        channel: agility_webrtc.channelName,
                message: {
                type: "SLIDE",
                options: {slide: slide_to}
        }
});
```

In either function, we are going to publish a message to our channel with a type of "SLIDE" to indicate a slide change. We have stored the slide number that we want everyone to switch to in a variable named "slide_to", and we'll pass that value on the channel as well.

Back up in the init() function of our webrtc.agility.js file, we've added a statement to subscribe to the pubnub channel:

```
agility_webrtc.currentUser.subscribe({
        channel         : agility_webrtc.channelName,
        callback        : agility_webrtc.onChannelListMessage
});
```

Now whenever a message is published to the channel, everyone subscribed will receive a callback to the onChannelListMessage method. This callback method is defined in our agility_webrtc variable as:

```
onChannelListMessage : function(message){
        var self = agility_webrtc;
        switch(message.type){
                case "SLIDE":
                        agility_webrtc.changeSlide(message.options);
                break;
        }
},
```

We've setup this method to use a switch statement, where each case will be one of the possible message.type's that we implement in our application. For now, we are only implementing "SLIDE" in order to change the slides. If a message with any other type is published, it will be ignored.

That SLIDE message will then call a method changeSlide(slide_number), which we have implemented right above the onChannelListMessage() in our code.

```
changeSlide             : function(options){
        if(options == null){
                options = {slide: 1}
        }
        $(".slider").carousel(options.slide);
        active_index = $(".carousel-inner .active").index();
        switch(options.slide){
                case "prev":
                        active_index--;
                break;
                case "next":
                        if(($(".slideCount li").length - 1) == active_index){
                                active_index = 1
                        } else {
                                active_index++;
                        }
                break;
                default:
                        if(typeof options.slide === 'number')
                        {
                                active_index = options.slide;
                        }
                break;
        }
        $(".slideCount li").removeClass("active");
        $($('.slideCount li')[active_index]).addClass("active");
        agility_webrtc.current_slide = active_index;
},
```

This final method for changing the slides is pretty straightforward. The options value that you pass in is the slide number to change to, and so the carousel holding the presentation images is set to switch to that slide number. We'll go to slide number one if no options are passed in.

We'll also change the value of an active_index variable, and use that at the bottom to add a CSS class called "active" to the appropriate circled number underneath the slides, so that we can show which one is the currently selected slide.

8.7. Testing the slide synchronization

At this point, your code to change the slides is complete. If you run "node server" and navigate back to the application at localhost:3000 in two different browsers. Use the http://localhost:3000/presenter/ URL in one browser, and in the other, just use the base URL http://localhost:3000/.

In the presenter view, you should see the navigation controls. Press any of the controls and you should see the slides change in the presenter's browser. We have put three simple placeholder slides in the application, each with the same number in the slide graphic so you can easily see the changes.

As the presenter is changing slides, you should also see the same slide changed in the regular attendee browser window, with only a momentary delay. Notice again that the attendee has no navigation controls, and that the circled numbers showing which slide you are on are not clickable for attendees.

If you're having any trouble getting this to work, you'll want to fix that before moving on to subsequent chapters. These are the problem areas to look out for:

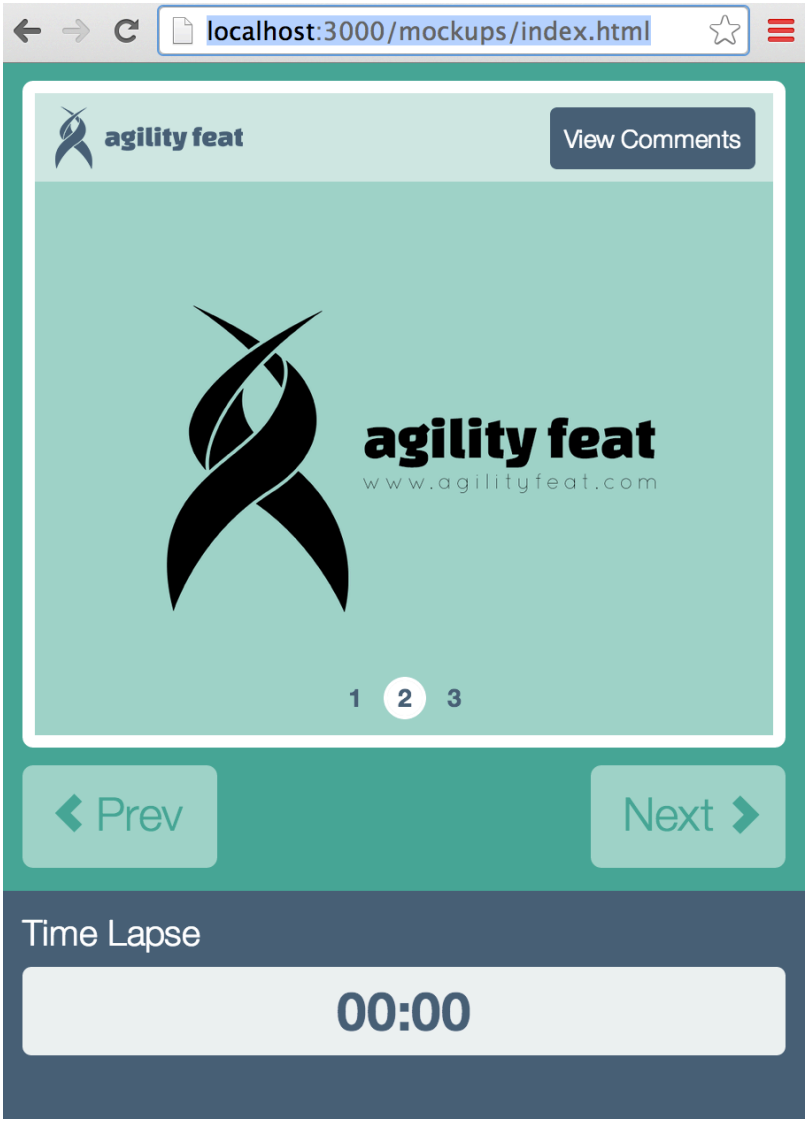
- Browser type – Although PubNub supports any recent modern browser, our code is optimized for Chrome and Firefox. Try one of those if you have problems with other browsers.

- PubNub keys – Remember that you need to update our sample code with your own publish and subscribe keys from PubNub. The variables containing those keys are located near the top of the webrtc.agility.js file. Just search on "your pub key" and you'll find the credentials declaration. Remember that anytime you refresh your code from one of our branches in this project, you'll need to put your keys back in so keep them handy.

- Javascript errors – If you're still having trouble, open the Javascript console in your browser (In Chrome, this is accessed by View#Developer#Javascript Console). You need to have a working internet connections, or you will see errors loading the pubnub.js file, and that the PUBNUB variable is not declared.

- You can get a copy of our code for this chapter on the following branch, it's always a good idea to compare your implementation to that or to just use our code as a base for future chapters: https://github.com/agilityfeat/webinar-tool/tree/syncsetup

There's one last fun thing I recommend you do now. Remember how the presenter on a mobile device has their own special layout with Previous and Next buttons? If you have your code deployed on a publicly accessible URL or IP Address, then you can bring up the presenter's view on your iPhone. Pressing the Previous and Next buttons from your phone will now send the same slide synchronization messages to all connected users.

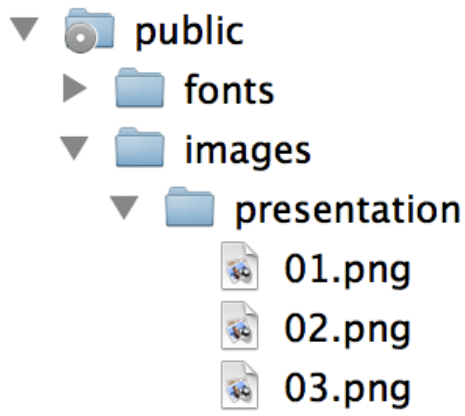


I like to do that when using this tool at conferences. From my laptop, I bring up the presenter view and project it on a big screen. Attendees at the talk are looking at the slides from their mobile devices on an attendee view. I then use my iPhone from the Presenter's URL and I can show them how pressing the Next button on my phone will advance the slide on the big screen and on their mobile devices. It's a cheap way to get ooh's and ahh's!

8.8. Adding your own slides to the tool

It's all working now, right! Excellent. But those three boring slides we put in our only good for testing the application. They don't make for a very good webinar.

We've kept this application intentionally simple, and the presentation is just a series of numbered images stored in the public/images/presentation folder. If you want to change the slides, just update or add to those images.



After putting in the appropriate image files, you also need to make the application aware of them. Look near the top of the webrtc.agility.js file for this declaration of the slide_pics:

```
slide_pics      : [
        { pic_url : "images/presentation/01.png" },
        { pic_url : "images/presentation/02.png" },
        { pic_url : "images/presentation/03.png" }
],
```

Just add/remove images to the slide_pics and you will be all set! We're using png's that resize well, so you if you're going to use this webinar tool for anything more than fun, make sure you pay attention to how the presentation images you use look on browsers, mobile devices, and maybe even projectors.

That's it for synchronizing slides! If you've got some time on your hands and want to enhance what we've done in this chapter, consider allowing presenters to upload their own images, change the order of them, or allow the display of powerpoint files natively.

Now that you know how to do the basics of synchronizing the slides, you'll find the next two chapters for adding in voting and commenting to be pretty straightforward. So make sure you get this working before moving on.

9. Coding: Attendee Votes

9.1. Code Samples

Prerequisite code for this chapter:

https://github.com/agilityfeat/webinar-tool/tree/syncsetup

Completed code for this chapter:

https://github.com/agilityfeat/webinar-tool/tree/voting

9.2. Introduction

The slide synchronization in the last chapter was our first messaging example, but only the presenter was able to send changes out to everyone. This is a democratic app though, we want the attendees to vote on slides and give us feedback. In this chapter, we'll create a simple example of how attendees can push messages out to the whole group too. Reviewing voting design aspects already setup

You may have already noticed that we had this array of vote types in the top of the webrtc.agility.js file:

```
slide_moods     : [
        { name : "Horrible" , count : 0, value : 0 },
        { name : "Bad"          , count : 0, value : 1 },
        { name : "Good"         , count : 0, value : 2 },
        { name : "Great"        , count : 0, value : 3 },
        { name : "Awesome"      , count : 0, value : 4 }
],
```

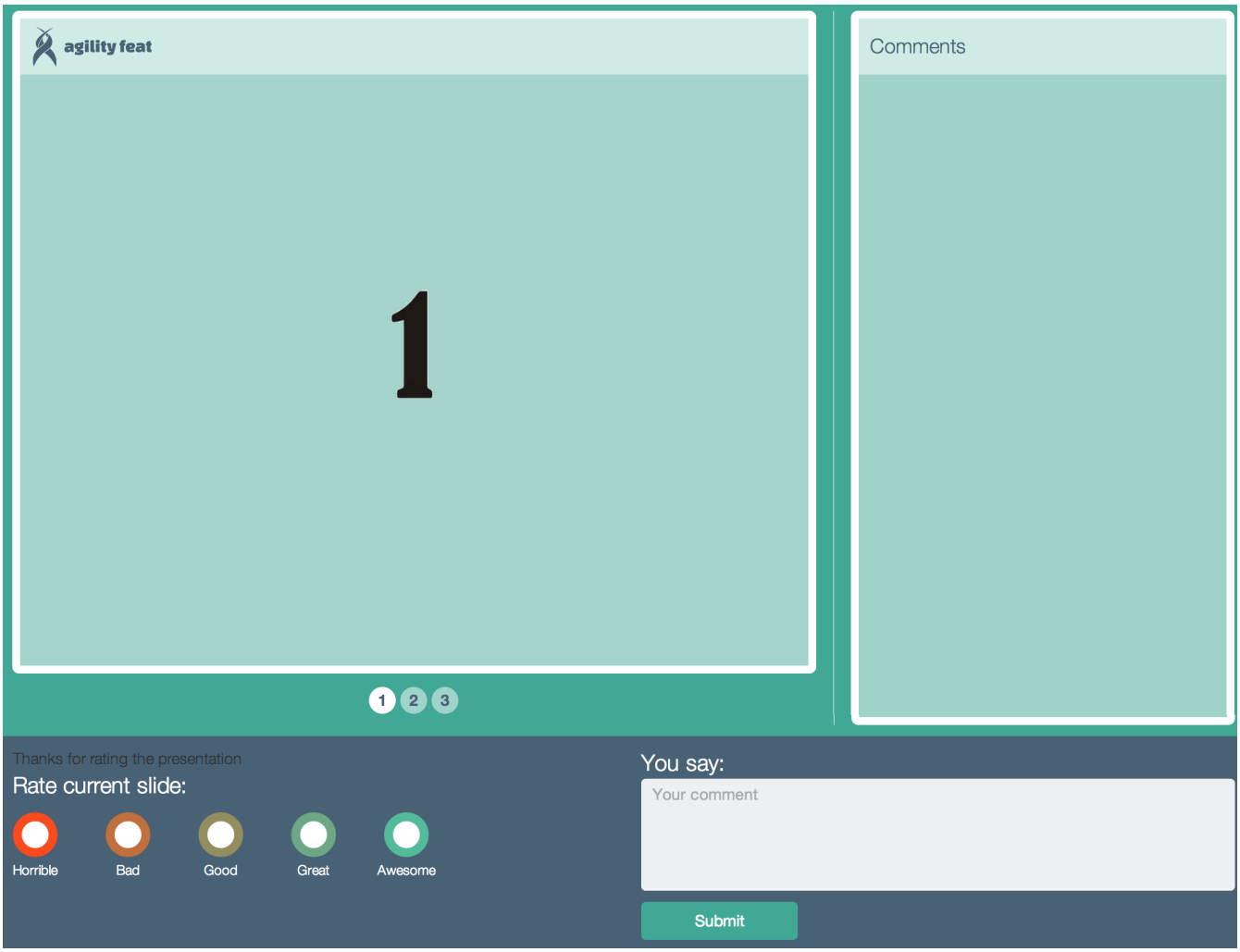
We're not exactly shooting for valuable feedback here, but you can imagine extending this same functionality to allow attendees to answer quizzes during a presentation in real time.

In the same file, we've also already declared a variable for storing the votes in later:

```
presentationVotes : [],
```

Our layouts were also already implemented for voting. In the presenter view remember that at the bottom of the screen is where votes are displayed. Currently, we are not putting any votes there so you just see an empty space with dots for the line graph.

And on the Attendee's view in the same place at the bottom, we see the controls for voting.



You'll notice that the controls for commenting are also visible in the Attendee's view on the bottom right, but we're not going to hook that up until the next chapter. For now, we need to start with attaching "onclick" events to the voting controls so that any attendee can cast a vote anytime they want to.

9.3. Adding in event bindings

Head back over to the setBinds method of the webrtc.agility.js file, and let's add in a function to bind to those voting circles that the attendee sees:

```
$(document).on("click", ".rateOption", function(e){
        var slide_mood = $(this).data("slide-mood");
        if($(this).is(".disabled")){
                return false;
        }
        $(".rateOption").addClass("disabled");
        var mood = _.find(agility_webrtc.slide_moods, function(mood){
                return mood.name === slide_mood;
        })
        $(this).animate({ opacity : 0.5 }, 400, function(){
                $(this).animate({ opacity : 1 }, 400);
                $(".rateOption").removeClass("disabled");
        })
        $(".title").animate({left : "-100%"}, 800);
        $(".thanks_for_rating").animate({left : "0"}, 800);
        _.delay(function(){
                $(".title").animate({left : "0"}, 800);
                $(".thanks_for_rating").animate({left : "-100%"}, 800);
        }, 2000);
        agility_webrtc.currentUser.publish({
                channel: agility_webrtc.channelName,
                message : {
                        type            : "VOTE",
                        mood_name       : slide_mood
                }
        });
})
```

Note that we are binding this function to any controls with the class of "rateOption" on them. If you look back at the templates.html file, you'll see that when we add the "slide moods" dynamically from that list we defined, then the classes of "rateOption" and "rate[mood name]" are added to each control.

```
<% _.each(slide_moods, function(mood){%>
  <span class="rateOption rate<%= mood.name %>" data-slide-mood="<%= mood.name %>">
        <span class="inneCircle"></span>
        <span class="label"><%= mood.name %></span>
  </span>
<%});%>
```

The second CSS class that is uniquely defined for each mood is used to determine dynamically which mood or vote the Attendee is casting.

Just to make things pretty, we've got some CSS animations going on there to give the user feedback that they have successfully cast their vote, and then we are ready to publish the vote out to the message channel.

9.4. Publishing vote messages to all subscribers

The code to actually cast the vote should look pretty familiar to you by now. Note that we are using a new message type of "VOTE" in this case, and we pass along the slide_mood that the user clicked on.

```
agility_webrtc.currentUser.publish({
        channel: agility_webrtc.channelName,
        message : {
                type            : "VOTE",
                mood_name       : slide_mood
        }
});
```

9.5. Subscribing to vote messages

In the last chapter, we already defined a callback method called “onChannelListMessage” for PubNub to pass along any new messages to. Now we just need to add a new switch statement in there for our VOTE message type.

```
onChannelListMessage : function(message){
        var self = agility_webrtc;
        switch(message.type){
                case "VOTE":
                        agility_webrtc.processVotes(message);
                break;
                case "SLIDE":
                        agility_webrtc.changeSlide(message.options);
                break;
        }
},
```

9.6. Displaying the votes

Our switch statement makes a call to method to processVotes, and passes in the vote from the message channel). Below is the code listing for processVotes and a few methods it is dependent on. Remember that because you define dependent javascript functions before they are called, you should start reading this code example from processVotes at the bottom, and then work your way back up the call stack.

```
displayAnalyticsGraphic : function(data){
        if(agility_webrtc.presentationVotes.length > 30){
                agility_webrtc.presentationVotes =
                        _.last(agility_webrtc.presentationVotes,2);
        }
        draw({
                data            : agility_webrtc.presentationVotes,
                container       : "#linesWarp",
                width           : $("#linesWarp").width(),
                height          : $("#linesWarp").height(),
                moods           : agility_webrtc.slide_moods
        });
        $("text, .guideWarp").hide();
        if($(".data-point").length > 0){
                setTimeout(function(){
                        $(".tipsy").hide();
                        $(".data-point:last").trigger("mouseover")
                        $(".area").addClass($(".data-point:last").
                          attr("class").split(" ")[1]);
                }, 1000);
        }
        agility_webrtc.last_time_votes_updated = Date.now();
},
```

```
displayBarsGraphic      : function(filtered_moods){
        $('.bargraph div.graphLabel[data-mood-name] div.bar').css({width:0});
        _.each(filtered_moods, function(mood){
                $('.bargraph div.graphLabel[data-mood-name="' +
 mood.name + '"] div.bar').animate({width: ((mood.percentage -1) +
 "%")}, 800, "swingFromTo")
                $('.bargraph div.graphLabel[data-mood-name="' +
 mood.name + '"] span.mood_count').html(mood.percentage + "%");
        })
},
```

```
filter_moods : function(){
        var filtered_moods = [];
        var mood_count, filtered_mood;
        _.each(agility_webrtc.slide_moods, function(mood){
                mood_count = _.countBy(agility_webrtc.presentationVotes,
                  function(vote){
                  return vote.mood_name === mood.name; }).true || 0;
                filtered_mood = {
                        name            : mood.name,
                        count           : mood_count,
                        percentage      :
 (mood_count * 100 / agility_webrtc.presentationVotes.length).toFixed(2)
                }
                filtered_moods.push(filtered_mood);
        })
        return filtered_moods;
},
```

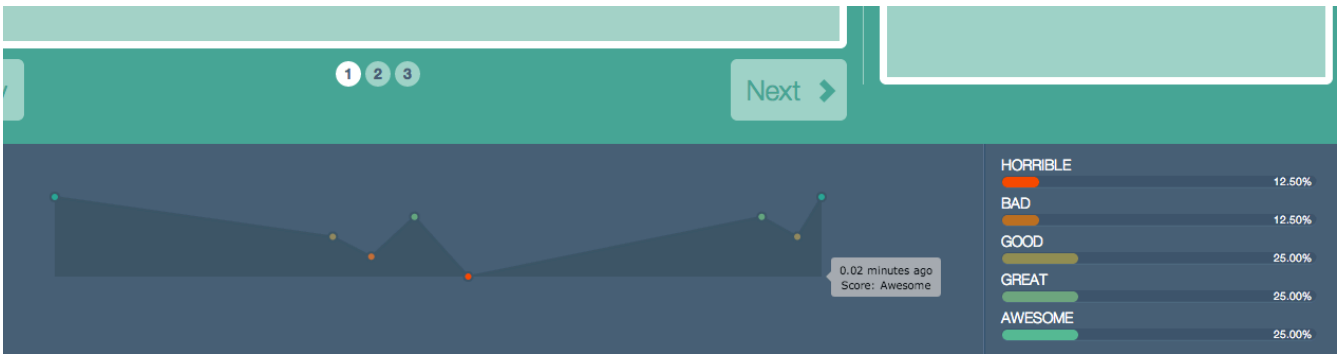
```
processVotes    : function(vote){
        var mood = _.find(agility_webrtc.slide_moods,
          function(mood){ return mood.name === vote.mood_name; });
        vote.date = vote.created_on ? new Date(vote.created_on) : new Date();
        console.log(vote.date);
        vote.value = mood.value;
        agility_webrtc.presentationVotes.push(vote);
        filtered_moods = agility_webrtc.filter_moods();
        agility_webrtc.displayBarsGraphic(filtered_moods);
        if((Date.now() - agility_webrtc.last_time_votes_updated) > 500){
                agility_webrtc.displayAnalyticsGraphic();
        }
},
```

In this example, we are just taking the specific vote and then updating that vote on the line graph. The line graph is updated every 500ms at the most just to keep from overwhelming the screen. For that to work, you'll need to initialize the last_time_votes_updated variable up in the top of the agility_webrtc definition, near the other variable definitions like isPresenter.

```
last_time_votes_updated : Date.now(),
```

9.7. Testing and extending the voting capabilities

Now you should be able to test your code. Open up two browsers again for a Presenter and an Attendee. Start casting votes in the Attendee view, and you'll see the line graph update in the Presenter screen along a horizontal time series.



You can also see a demonstration of this working at the following quick video:

http://www.screencast.com/t/Wgo9aCftoz

In this very simple implementation of voting, we've created a scrolling graph that shows you the most recent votes. It's giving the Presenter a real-time view of the mood of the audience and how they are enjoying the presentation.

If you were using this to actually gauge the mood of your audience, you might want to watch the line over time as you present. Are you starting to get bad votes or indications that the audience is bored? When you are presenting in person that is usually easy to gauge from watching the audience, but in a remote webinar you have no way to measure the pulse of your audience. You might use this as a way to measure that pulse and speed up, slow down, or pause for questions based on audience feedback.

If you want to extend this code to make it more useful, consider implementing any of these features as an additional exercise:

- Store the votes in a database and pull them from there (in our example code nothing is being saved, and so if you need to refresh your Presenter's view, you will lose the vote history and the line graph will reset)
- Save the votes on a per slide basis so you can get data about how much people liked individual slides in your presentation. In our example implementation, there's no correlation being made between the slide and the votes, it's just a running average on the line graph.
- Perhaps you want this vote data, but you don't want to share it with attendees on the big screen. They might make you look bad by casting a lot of negative votes! You could move the voting displays to be only on the Presenter's mobile view, or on a different page altogether, so that the Presenter can monitor the results as they come in without sharing them on the main screen with all Attendees.
- Instead of voting, change the functionality to be an online quiz that you administer to attendees in the presentation. This could be a complicated but interesting addition that allows the presenter to setup questions to ask in advance, specify the correct answer, and then display the quiz results in real-time (or after all entries are made). In this case, you probably want to hide the voting/quiz controls until the Presenter decides to administer the quiz.

At this point, you have fully working voting code. If you have any trouble getting this to work, remember that you can always look at our examples for ideas or to use as a starting point. The completed work for this chapter is located at:

https://github.com/agilityfeat/webinar-tool/tree/voting

Now you have a webinar tool that allows the Presenter to keep slides in synch across all attendees (Presenter pushes to Attendees), and for Attendees to push feedback up to the Presenter's screen.

In our final example of real-time messaging, the next chapter will show you how to incorporate comments into the application for a little more two-way communication between Presenter and Attendees. This commenting functionality is also going to be useful because we will use comments as the basis for how we make WebRTC video calls to users in later chapters.

10. Coding: Attendee Comments

10.1. Code Samples

Prerequisite code for this chapter:

https://github.com/agilityfeat/webinar-tool/tree/voting

Completed code for this chapter:

https://github.com/agilityfeat/webinar-tool/tree/comments

10.2. Introduction

In our final messaging example, we will show how Attendees can make comments on their mobile device, and publish those to the Presenter's view. We will also allow Presenters and Attendees to see the comments of all attendees in their mobile views. Only the Presenter will be able to delete offensive comments.

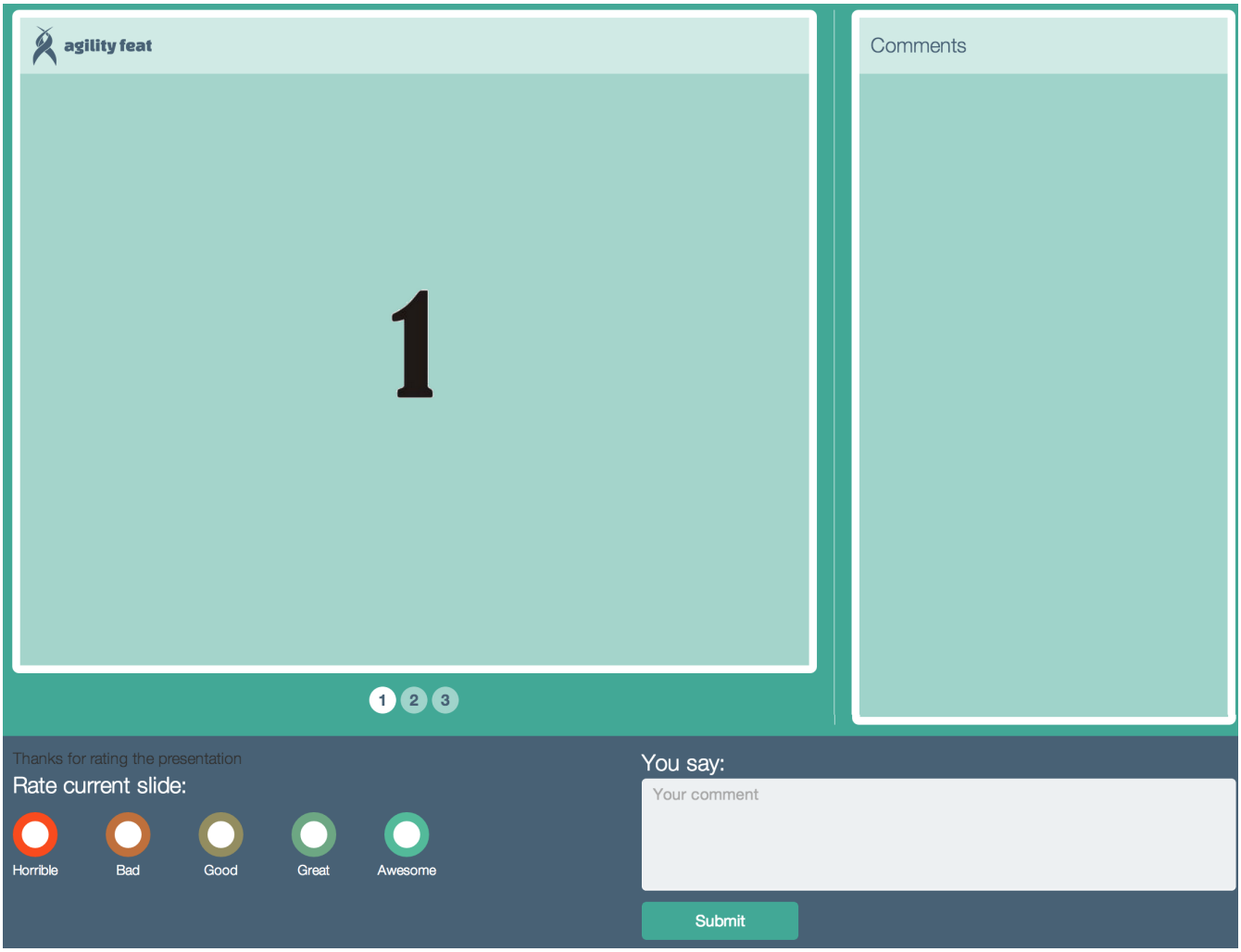
If you feel that you've got the concepts of real-time messaging down, you might want to skip this chapter in order to move on to the WebRTC calls. That's fine, but you will want to at least pull down the completed code for this chapter from this branch and keep it handy for future chapters.

https://github.com/agilityfeat/webinar-tool/tree/comments

In the WebRTC calls, we'll use comments from Attendees as the way that the Presenter can initiate a video chat. So you'll need our commenting code as a base to start from.

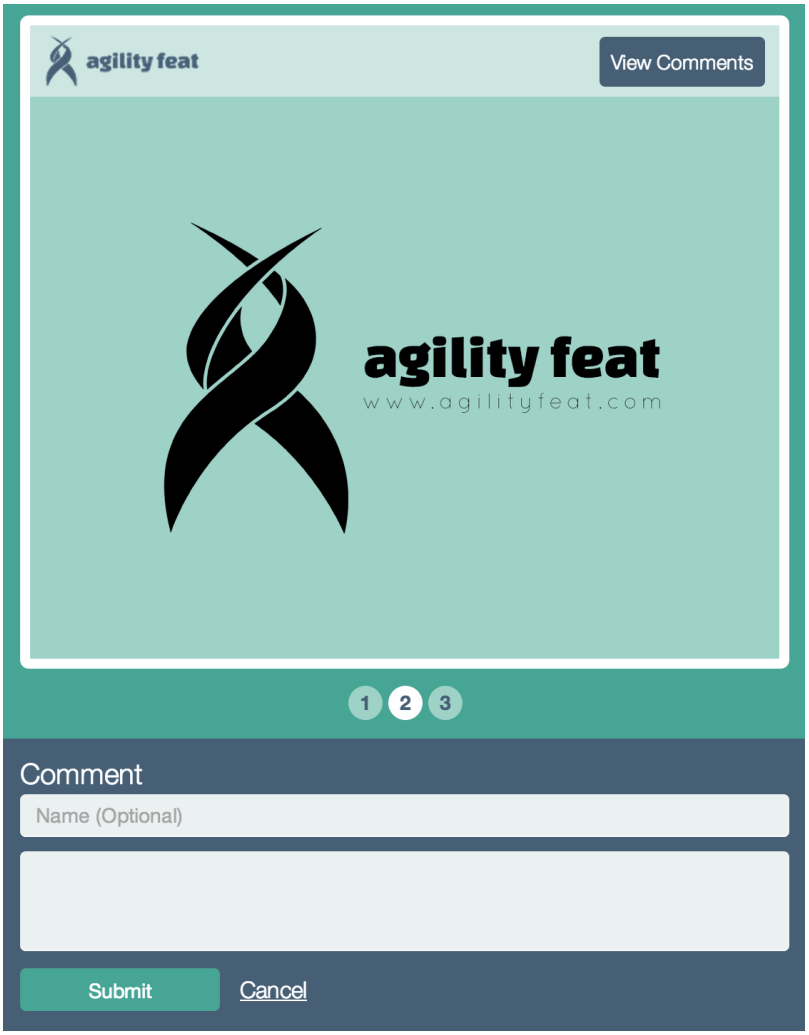
You might also want to implement your own commenting functionality before reading this chapter, as a way to challenge yourself. The concept for publishing and subscribing to messages are hopefully clear from the previous two chapters, and so you just need to add a new message type for comments and plug in some code to allow users to enter the comments, and then receive those messages and display them on the Presenter view. The design elements for entering comments from the Attendee view are already in place from previous chapters.

For the Attendee on their desktop, the commenting controls are already in the design on the lower right.



This is an area where responsive design is important though, because we expect that many people will attend our webinars on their mobile devices, or if they are watching us present in person at a conference, then they will want to follow along on their phone.

For tablets, we've left the design the same as on smartphones, but for smartphones we have a more compact view:



In this case, the main focus is kept on the slides and voting in the default attendee mobile view, but if they click on an “Add Comment” link, then the above view is visible where they can enter their name and a comment. Later, we’ll come back and make the “View Comments” control on the top right functional so they can also read comments on their smartphone.

10.3. Binding an event for Attendees to enter comments

Time to head back to our friendly (and growing) setBinds() method in webrtc.agility.js. We’re going to add an event binding for when the Attendee clicks on the Submit button.

```
$(document).on("click", "#btn_send_message", function(e){
        var message = $(".commentsHere").val().trim();
        if(message !== ""){
                var username = “attendee”;
                agility_webrtc.currentUser.publish({
                        channel: agility_webrtc.channelName,
                        message : {
                                type    : "MESSAGE",
                                text    : message,
                                user    : {
                                        name            : username,
                                        uuid            : agility_webrtc.uuid
                                },
                                id              : Date.now() + "-" +
                                        username.toLowerCase().replace(/ /g, '')
                        }
                });
                console.log(agility_webrtc.uuid +
                  " published comment " + message);
                $(".commentsHere").val("");
        } else {
                $(".commentsHere").val("");
        }
});
```

In this code, we use jquery to get the value of the comments text box, because it’s been marked with the “commentsHere” CSS class. That is now published to the message channel with a new message type of “MESSAGE”. We are also passing along an id value this time because we can use that to help with the deletion of messages later on by the Presenter.

Notice that we are sending a username and unique identified for the user (UUID) along with the message. The username is going to be used to help delete comments later. The UUID is going to be necessary for the WebRTC calls in subsequent chapters.

The username is a placeholder value for now. If you use this in production, you should add some authentication around the app and put an actual username in there.

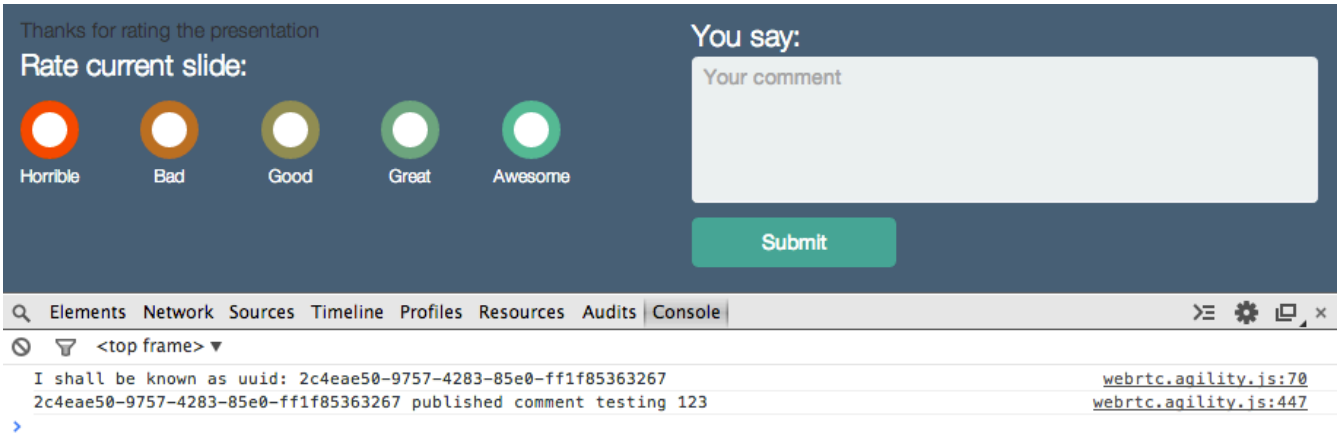
We need to add these two functions for storing and retrieving some values from the browser’s local store, which we’ve put just above the setBinds function:

```
setInStore      : function(item, key){
        if(item == null){
                return false;
        }
        item = _.isString(item) ? item : JSON.stringify(item);
        window.localStorage.setItem(key, item);
},
getFromStore : function(key){
        return JSON.parse(window.localStorage.getItem(key));
},
```

To populate the uuid, we will use the identifier that PubNub is using for our connection. In the init() function we’ve added this code to the bottom:

```
var UUID_from_storage = agility_webrtc.getFromStore("uuid");
if(UUID_from_storage){
        agility_webrtc.uuid = UUID_from_storage.uuid;
} else {
        agility_webrtc.uuid = PUBNUB.get_uuid();
        agility_webrtc.setInStore({ "uuid" : agility_webrtc.uuid }, "uuid");
}
console.log("I shall be known as uuid: " + agility_webrtc.uuid);
```

We now have everything we need to send comments, and you should see output in your javascript console similar to this from the Attendee's view:



Note that after a comment is sent, we are clearing the comment box out. Next we need to process the comment and display it on the screen in all clients.

10.4. Subscribing to new comments

Back in the onChannelListMessage method, we need to add a new switch case for the "MESSAGE" type:

```
case "MESSAGE":
        self.storeMessageAndDisplayMessages({
                from            : message.user.name,
                from_uuid       : message.user.uuid,
                message         : message.text.replace( /[<>]/g, '' ),
                id                      : message.id,
                can_webrtc      : false,
                type            : message.type,
                is_your_message : (message.user.uuid === agility_webrtc.uuid)
        });
break;
```

In this case, we will pass along all the message details to a method to display it on the UI. We pass in the username and UUID, a sanitized version of the message with all html or script tags removed (we don't want any injection attacks from attendees!).

The "can_webrtc" parameter is going to be an important indicator later of whether we can do a video chat with this user or not. For now, we just pass false.

10.5. Displaying comments on the Presenter view

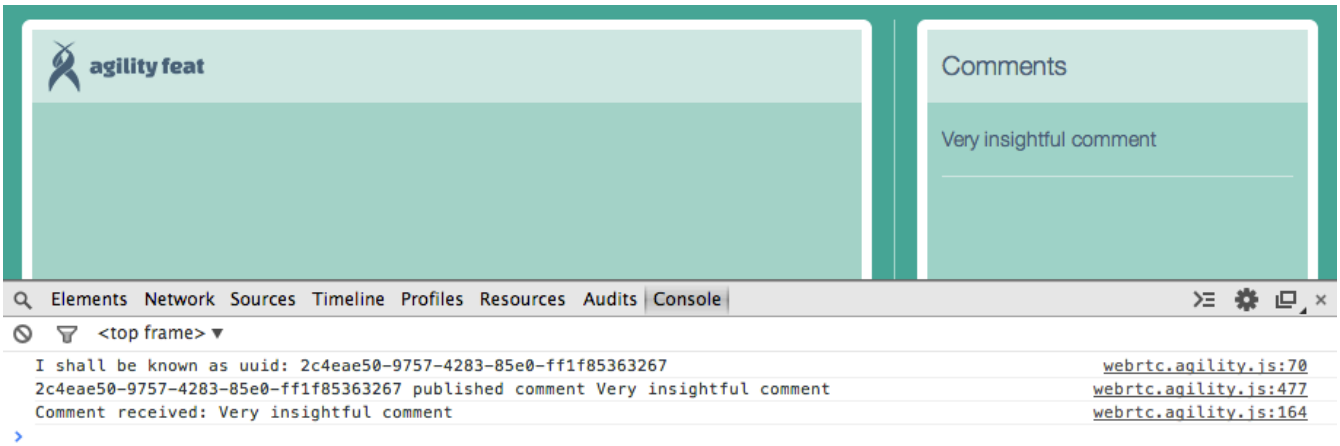
The method that we call for actually displaying these comments is rather simple, and goes in the top part of webrtc.agility.js with other local methods:

```
storeMessageAndDisplayMessages : function(message){
        var self = agility_webrtc;
        self.channelMessages.push(message);
        console.log("Comment received: " + message.message);
        self.render_prepend({
                container       : ".commentsList",
                template        : "#channel_chat",
                data            : {
                        messages                : self.channelMessages,
                        this_message    : message,
                        app                     : self
                }
        })
        if($(".doneBtn").is(":visible")){
                $(".deleteBtn").fadeIn();
        }
},
```

All we do here is push the new comment onto the stack of comments called “channelMessages”, and then re-rendering the commentsList from the templates.html file.

If you look in the templates.html file, you may notice that the div for commentsList has nothing in it. If you want to alter the style of the comment listing from our design, you’ll need to do that in the main.css file. Do a search for “commentsList” in the project and you’ll see the areas where we have specified the style and placement aspects of the comment listing.

It’s time to test it out a little bit! Open up separate Presenter and Attendee views in two browsers, and make a comment from the Attendee view. The comment will now be displayed on the Presenter and Attendee desktop views similar to the following. The comment box on the Attendee’s view is cleared after the comment is submitted.



Notice that when the attendee sends a comment in the above figure, it is also displayed in their browser. Display of the comments happens the same way regardless of whether you send the comment or not.

In the Presenter's window, they will also receive the same comment on the message channel, and then display it in the comments box.

10.6. Seeing the comments on the mobile views

There's just a couple more little event bindings that we need to add in order to get comments to work on mobile devices as well. On the mobile view, adding and viewing comments are slightly hidden behind a button and a link. Add the following four methods to your setBinds() function:

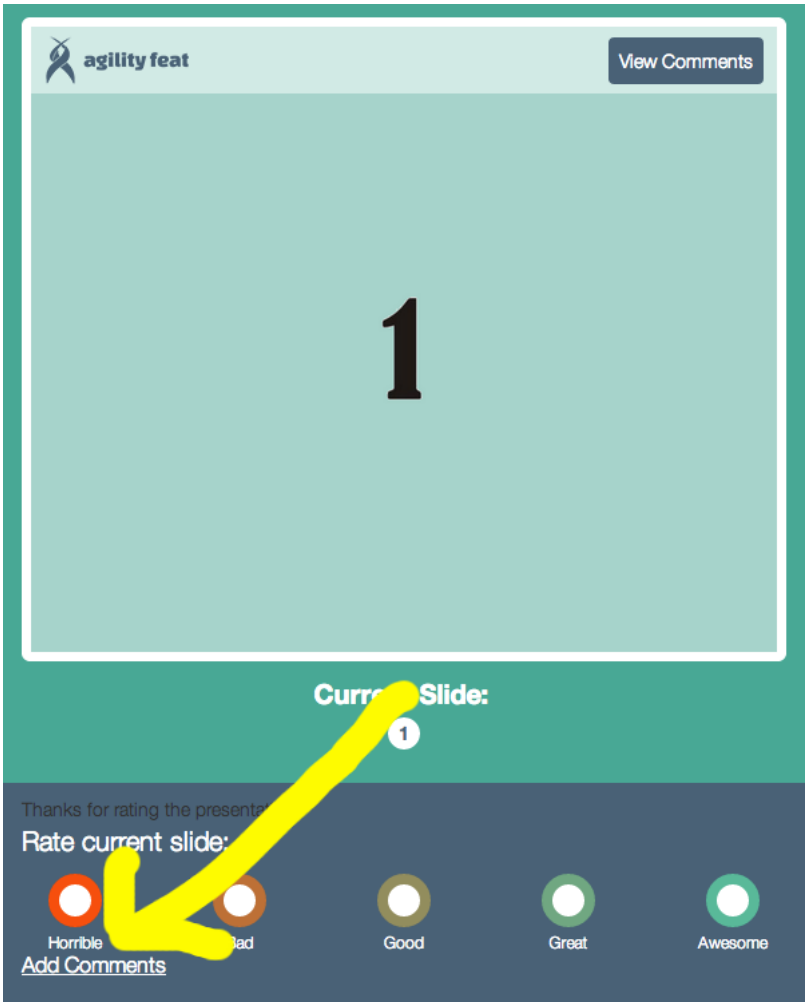
```
$(document).on("click", ".showCommentBox", function(e){
        $(this).fadeOut("fast");
        $(".commentSlideWrap").fadeIn("fast");
        $(".rateSlideWrap").fadeOut("fast");
});
```

```
$(document).on("click", ".hideCommentBox", function(e){
        $(this).fadeOut("fast");
        $(".showCommentBox").fadeIn("fast");
        $(".commentSlideWrap").fadeOut("fast");
        $(".rateSlideWrap").fadeIn("fast");
});
```

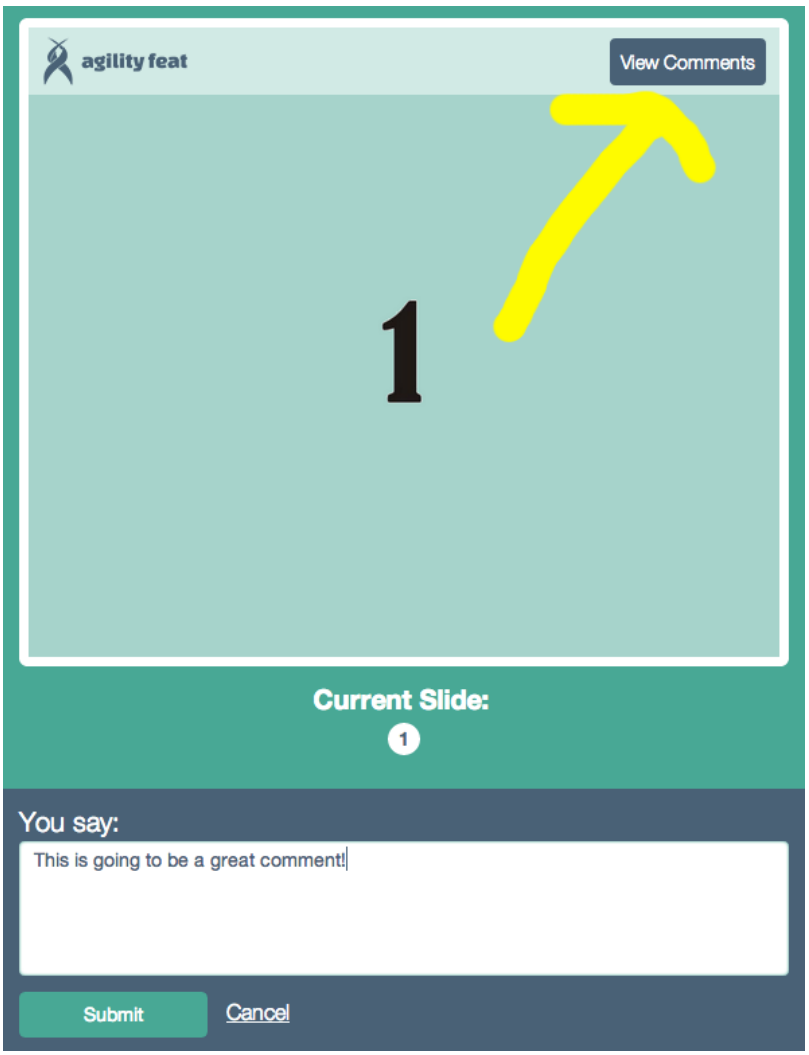
```
$(document).on("click", ".playerWindowWrap .btnShow", function(e){
        $(".playerWindowWrap").fadeOut("fast");
        $(".commentsWindowWrap").fadeIn("fast");
});
```

```
$(document).on("click", ".commentsWindowWrap .btnShow", function(e){
        $(".commentsWindowWrap").fadeOut("fast");
        $(".playerWindowWrap").fadeIn("fast");
});
```

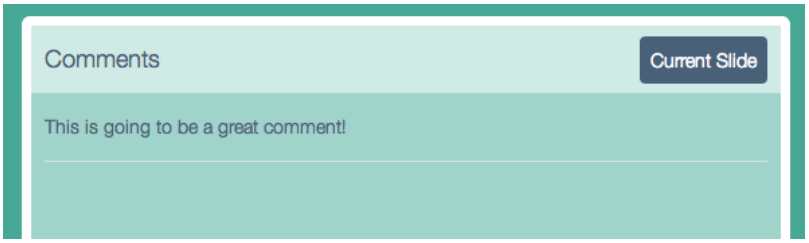
To see the functionality in action on mobile phones, either resize your attendee view down to a smaller size like a smartphone, or if you've deployed your app to a publicly addressable URL, go to that URL from your smartphone and enter a comment. The following figures show the functionality in order from an Attendee making a comment to a Presenter viewing it.



In the default Attendee view, the focus is kept on voting, but if the attendee wishes to make a comment, they press the “Add Comments” link on the bottom left.

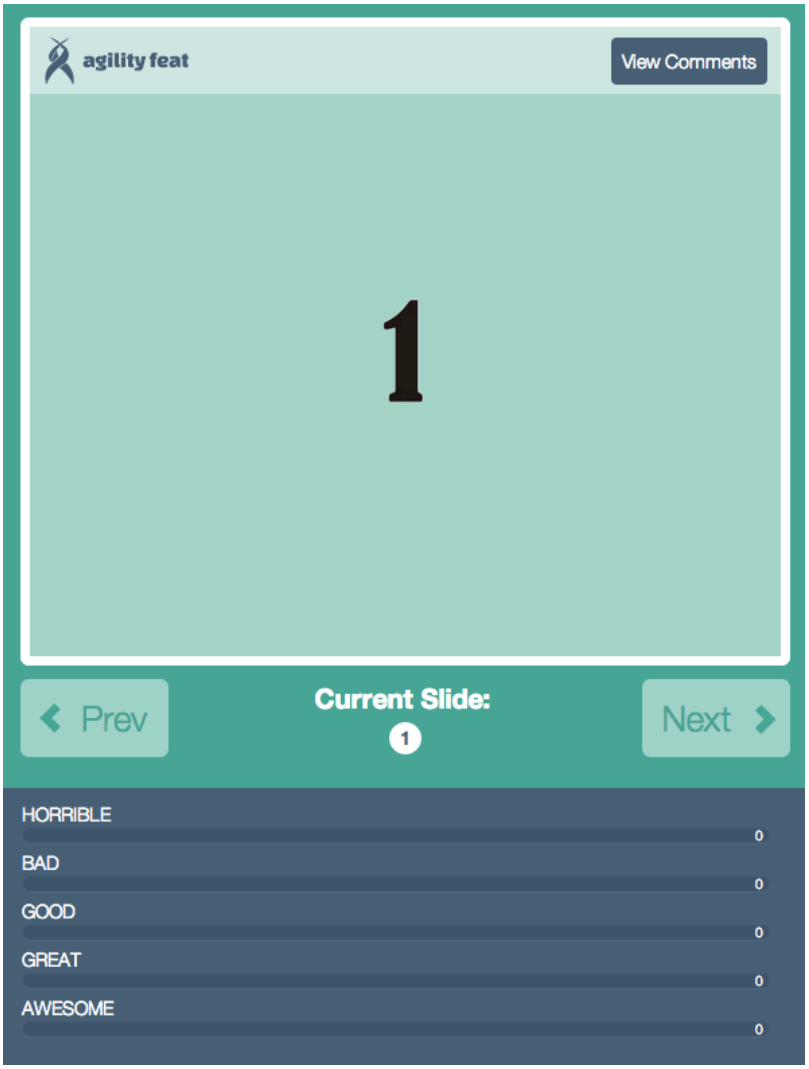


After pressing “Add Comments”, the attendee can enter a comment in the bottom of the mobile view. After submitting that comment, they can press the “View Comments” button in the top right to switch from viewing the current slide to a mobile view that shows their comment (as well as comments from other attendees).

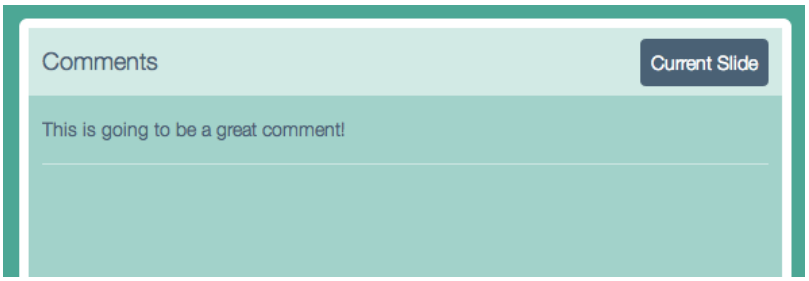


To go back to viewing the current slide, the attendee presses the “Current Slide” button in the top right.

For the Presenter, they also have a “View Comments” button on their mobile view. They are not able to make comments though because that’s going to be really awkward if the presenter is standing in front of an audience typing comments into their phone!



When the Presenter views the comments, it looks exactly like the Attendees view. All comments would be displayed here, although we are just showing one right now.



All comments from attendees are now accessible in all the different responsive views: In the Presenter desktop version, on Attendee's desktop versions, and in all mobile views through the View Comments button.

10.7. Allowing Presenters to delete comments

I was once at a conference where someone gave a demonstration of how to build a NodeJS text chat application. They invited attendees to enter text messages of their own and see them displayed on the projected screen.

In a room full of developers, the inevitable result was that people starting inserting html tags, images, and javascript “hello world” snippets in the text chat to see if they could hack the presenter’s example. At one point, someone inserted a huge “Pawned” image via the text chat that disrupted the presenter’s talk. Fortunately the presenter had a good sense of humor about it, but that’s why we put a little quick code in our example code here to sanitize the messages before posting them.

Even with a little tag sanitizing going on, attendees may still post comments that are offensive or otherwise disrupt your webinar. So we should give the Presenter a little extra power to delete messages.

Up to this point, we’re not storing anything in databases (that exercise is left up to the reader if you choose to build a production app on top of our examples). Because there is no way in our app to delete the comments from a database, we need to make sure that we delete the right comment from the Presenter’s view and also from the views of everyone subscribed to the messaging channel.

First, we need to add in a couple controls that only the presenter will see. In your templates.html file, search for the comment that “<!-- Comments call code will go here later -#”. Replace that comment with this code:

```
<% if( isPresenter ){%>
        <div class="actionsWrap">
                <span class="initialCall">
                        <span class="cameraCall"><span
                          class="glyphicon glyphicon-facetime-video"></span></span>
                        <span class="commentsCall">Edit</span>
                </span>
                <span class="doneBtn">Done</span>
        </div>
<%}%>
```

This adds a video call button (to be used in subsequent chapters) and an “Edit” button, only accessible to the Presenter.

Next we need to bind a couple methods to the Edit button (referred to below as the commentsCall button for reasons that become more clear later) in our setBinds function.

```
$(document).on("click", ".commentsCall", function(e){
        $(".commentsCall").parents(".initialCall").fadeOut();
        $(".deleteBtn").fadeIn();
        $(".cameraBtn,.screenShareBtn").fadeOut();
        $(".doneBtn").removeClass('blue').fadeIn();
        $(".commentItem").addClass('active');
});
```

```
$(document).on("click", ".doneBtn", function(e){
        e.preventDefault();
        $(".initialCall").fadeIn();
        $(".deleteBtn, .cameraBtn, .doneBtn, .screenShareBtn").fadeOut();
        $(".commentItem").removeClass('active');
})
```

Now if you refresh your Presenter and Attendee views, make a couple of new comments, and then press the Edit button in the Presenter view, you will see delete controls next to each comment.

Now we need to bind that Delete control to a function in setBinds().

```
$(document).on("click", ".deleteBtn", function(e){
        e.preventDefault();
        var message_id = $(this).data("message-id");
        $(this).parents('.commentItem').animate({right:"-100%"},
           200, function(){
                $(this).empty().remove();
                if($('.commentItem').length === 0 &&
                  $(".doneBtn").is(":visible")){
                        $(".doneBtn").trigger("click");
                }
                agility_webrtc.currentUser.publish({
                        channel: agility_webrtc.channelName,
                                message : {
                                type    : "DELETE_MESSAGE",
                                id              : message_id
                        }
                });
        });
});
```

The id that was passed previously when a comment was submitted is now accessible by jquery via the “message-id” tag. We can remove that specific comment from the Presenter’s view, and then publish a “DELETE_MESSAGE” out to the messaging channel to instruct other subscribers to also delete that message.

Now in the “onChannelListMessage” method we just need to add a simple switch case that will remove the comment for all subscribers when they receive the “DELETE_MESSAGE” type from the messaging channel.

```
case "DELETE_MESSAGE":
        $('.commentItem[data-message-id="' + message.id +
          '"]').animate({right:"-100%"}, 200, function(){
                $(this).empty().remove();
        })
break;
```

Now you have all the code you need to delete the functions. The code knows which message to delete based on the username that you passed and the date stamp that was added to it for the id parameter that we created earlier when we sent the message.

Refresh your browser views and try it out! Enter something really insulting, because now you can delete it!

10.8. Extending the commenting functionality

That does it! You should now be able to enter and view comments from any Attendee view. On the Presenter view you will see all comments and you have the option to delete offensive ones.

If you want to see a video of how the functionality works for us at this point, check this out:

http://www.screencast.com/t/0AWek6sl04

Would you like to add more functionality to the comments? Consider these ideas:

- Store the comments in a database. Currently they are not stored anywhere and so anyone late to the webinar will not see the comments entered before they joined
- Implement user authentication of login of some sort and use that username with the comments. How about using Facebook, Twitter, or oAuth login and then displaying the attendee's profile picture with the comment?

If you had any trouble getting the code in this chapter to work, remember that you can see our complete example here:

https://github.com/agilityfeat/webinar-tool/tree/comments

11. WebRTC Video/Audio Primer

11.1. WebRTC Overview

WebRTC stands for Web Real-Time Communications, and is an emerging HTML5 standard for peer-to-peer video, audio, and data communications in the browser.

WebRTC is about more than just video chat applications, although that is the most obvious use for it. As we mentioned in an earlier chapter, this book focuses on two major technology areas:

- Real Time Messaging using Publish/Subscribe patterns
- Real Time Communications using WebRTC

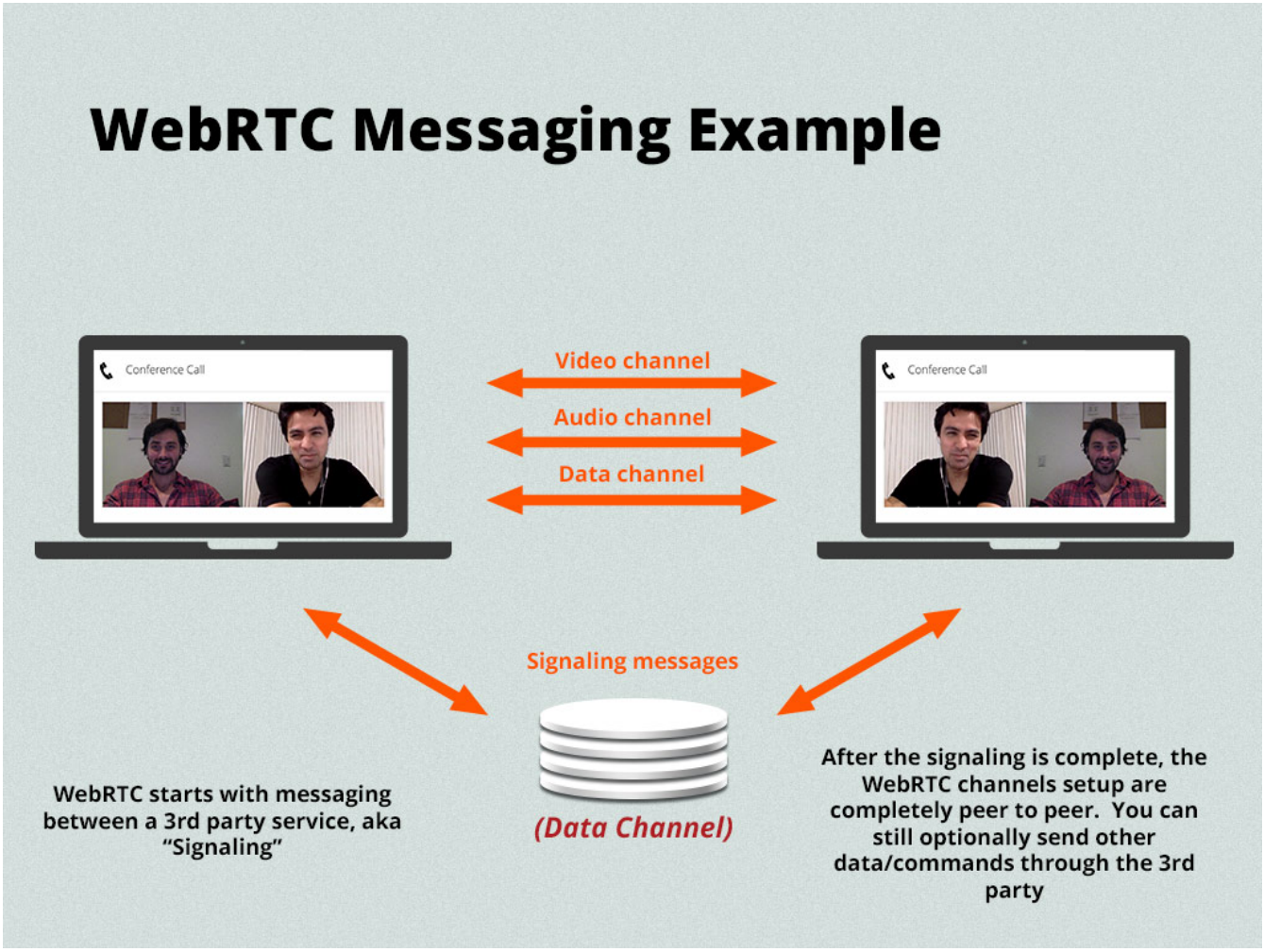
Up to this point, everything we have done utilizes real time messaging to send what are essentially text messages back and forth. We are using those plain text messages to case votes, make comments, and to keep our presentation slides in synch across all browsers.

The applications you can build that communicate in real-time using Publish/Subscribe patterns are interesting, and you can build cool applications based only on that. But now we're going to get into some even more interesting technology for Real Time Communications.

Now we are moving into the WebRTC part of the book. This will enable us in subsequent chapters to let the Presenter call an Attendee and have a live video chat with them as part of our webinar tool.

Let's revisit our WebRTC overview from earlier in the book. As we described previously, the simplest application of WebRTC is to build your own personal version of a Skype client, directly in your browser application.

To implement a WebRTC call you only need HTML5 and Javascript. The users of your application do not need to download any plugins (like was necessary to build similar applications using Flash).



In the figure above, we see Daniel and Allan having a video chat using a web browser.

For Allan to call Daniel, there must be a connection setup between their two browsers. This generally means they are going to both visit the same website and agree to talk to each other. The technical term for this agreement to talk is “signaling” in WebRTC. The signaling is typically handled by the website that they are both visiting or a 3rd party server that the website coordinates their connections with. This 3rd party server could be your web server, or you could use a paid-service to handle the signaling for you.

Once Allan and Daniel have established a WebRTC connection, the rest of their conversation and data exchange is all done in a Peer-to-Peer encrypted connection. The signaling server never receives any of the video, audio, or data traffic between the two browsers. This is an interesting architecture that presents both opportunities and challenges, as we’ll explore shortly.

11.2. WebRTC is not supported in all browsers

This is the biggest disclaimer that needs to be given about WebRTC, and unfortunately this disclaimer is likely to apply for a while longer.

WebRTC is fully supported in the latest versions of Chrome, Firefox, and Opera. These browsers update themselves automatically on your computer without you even realizing it, so if you’re using one of them then you support WebRTC. Hooray!

Internet Explorer and Safari do not support WebRTC as of this writing. There is no set timeline when they will support it, and lots of speculation about if they ever will or not.

If you are only interested in building applications to run on Internet Explorer, then you might as well stop reading here and cool your heels for a while. I'll hopefully speculate that interoperability in IE and Safari will happen in 2015, but I'm saying that from the comfort of early 2014 and future updates to this book may remove this sentence altogether.

But if you have an application where you can dictate which browsers your users must use for your app, or you are just speculatively building an app for a future market (which is not a bad idea in my opinion), then keep reading.

I personally believe that IE and Safari will support WebRTC in the future. I have no sources to back that up, it's just my personal belief that the momentum behind WebRTC is pretty strong and Microsoft and Apple will have to get on board sooner or later.

Why aren't Microsoft or Apple on the WebRTC bandwagon now? There are a few reasons people throw around:

1. **They see WebRTC as a threat to their own products.** Microsoft owns Skype, and Apple has Facetime. Perhaps they are hoping that WebRTC will go away because it poses a competitive threat to those product lines. While there may be some of that fear going on, I think both companies are smart enough to realize they can't hold out forever.

2. **WebRTC is not stable enough.** Firefox and Chrome have a distinct advantage when dealing with new technologies like WebRTC. Their browsers update automatically on your computer which means that they can deploy frequent improvements to their WebRTC support. IE and Safari do not deploy on such a frequent release cycle, so they have an incentive to wait for the final standard rather than deploy a half-baked solution that cannot be updated for months at a time.

3. **The big companies always innovate last.** Perhaps they are intentionally taking a "wait and see" approach. Google is the biggest force behind WebRTC, and so maybe Microsoft and Apple prefer to just wait and see what comes of this before they support a competitor's pet project. Microsoft and Apple can probably afford to be late adopters on this and still become important players further down the line.

11.3. WebRTC on mobile devices

The current state of WebRTC on mobile devices is underwhelming. Chrome, Firefox, and Opera all support it on their mobile browsers, but this only applies to Android devices.

There is no WebRTC support in any iOS browser at this time. That's a big hurdle to future adoption of WebRTC. There are commercial platforms that will allow you to seamlessly integrate WebRTC applications with support for all major browsers and mobile devices, but WebRTC itself cannot work on iOS at this point.

That means if you need support on iOS, Safari desktop, or Internet Explorer, you need to explore commercial platforms that will blend WebRTC support with those other platforms. This means users on a non-WebRTC device will have to install an app or download a plugin. That takes some of the fun out of the WebRTC marketing pitch, but at least you can still build your app.

Our examples in this book focus solely on WebRTC, which means that this code only supports Firefox, Chrome, and Opera in the browser, or on Android devices.

11.4. WebRTC is Peer to Peer

After the signaling is complete, and a WebRTC connection has been established between you and another user, your connection is now completely Peer to Peer (P2P).

This is a wonderful thing for many applications. It's especially nice for data intensive applications (like video chat or file sharing), because it means the communication is happening directly between the two users who care about the data or video being exchanged. No intermediary server needs to host the data or stream it to other parties. This makes WebRTC a very efficient solution for data or video exchange between two parties.

This also means that WebRTC can scale extremely well for small conversations. The load put on your service's servers by signaling is very small. Once the P2P connection has been established between two parties, your server is out of the loop and so you can theoretically build a WebRTC based service that allows for thousands of simultaneous video chats between small groups of people.

11.5. WebRTC does not scale alone

The downside of WebRTC is also it's P2P nature. There is not a burden on your server necessarily for additional parties in a video chat, but there is a burden on each individual device in the video chat.

WebRTC only supports peer connections, which means that if you want to have a group video chat, every party to the conversation must have a separate WebRTC P2P connection to every other party in the conversation.

This is referred to as a mesh network, and it is not scalable. You simply won't be able to use WebRTC in its native form for a video chat between dozens of people. There is no set limit built into WebRTC, but a practical limit is probably between 4-8 connections in a single conversation.

WebRTC scales well to lots of simultaneous small conversations, but it does not scale well to large conversations.

11.6. WebRTC is Encrypted

I have a little bit of a "privacy nut" streak in me. That's unfortunate because I have a very unique name and so if you find anything on the internet from "Arin Sime", it's hard to deny that is me. Fortunately I'm also a pretty boring person, so this doesn't cause any practical concerns.

Given my privacy streak, and all the news these days about government and corporate snooping, it's good to know that WebRTC has us covered.

Once the P2P connection is established between two WebRTC clients, all traffic between those two clients is completely encrypted. I'm not going to guarantee you that the NSA or your neighbor can't break that decryption, but at least the wall is there to slow them down.

Beyond the more nefarious parts of the internet, this also opens up interesting applications for WebRTC in much more conventional industries. For example, WebRTC is likely to show up in healthcare and remote telemedicine applications because the natively encrypted communications channels that WebRTC provides is an important step to ensuring regulatory compliance.

In the USA, using WebRTC doesn't automatically mean your application will be HIPAA compliant, but it will be a nice feather in your cap when you try to meet the regulations.

Regardless of any regulations, it's just plain good manners to encrypt sensitive patient information and medical data, so WebRTC is a great choice for those type of real-time communications. This also makes WebRTC a great choice for corporate applications where enterprise security concerns control the use of sensitive company data outside internal company networks.

One important thing to note is that the signaling or handshaking required to start a WebRTC connection is not encrypted (unless you separately implement encryption on top of your own signaling solution).

Be careful about including uniquely identifying information about users in the signaling code or you may cause a security leak. The WebRTC encryption will still protect the content of the conversation, but including sensitive information in the signaling process could allow hackers to determine who is in conversations at what times, and that alone may be information you don't want to share.

11.7. WebRTC Signaling

The signaling, or handshaking, process is very important to the establishment of a WebRTC call. However, it's not specified in the WebRTC standard itself.

Some people point to this as a problem because it leaves a lot of decisions up to each development team tackling a WebRTC app. It's also an advantage though, because the context of how to establish a WebRTC call may vary based on your specific situation. That's the WebRTC standards' groups left it undefined.

We prefer to use a WebRTC API that handles the more complicated parts of the signaling for us. That way we don't need to deal with the intricacies of establishing or setting up STUN and TURN servers.

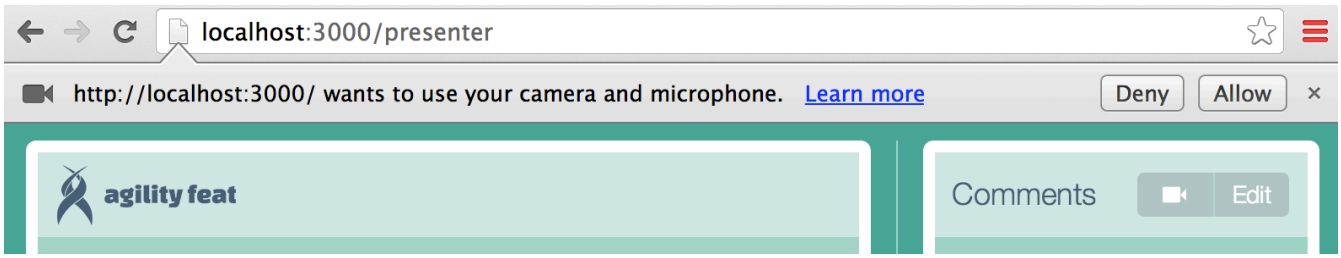
Even with the use of an API though, your app probably still needs to know some information about the users engaging in the WebRTC connection. Things like user id, browser type, and other application specific details may be important for you to pass along too.

11.8. Displaying WebRTC Video/Audio

Once you have established the connection between two WebRTC peers, the code to display the other user's video stream is (deceptively) simple. A call like this will return a video and audio stream from the user's browser:

```
navigator.webkitGetUserMedia({ audio: true, video: true}, gotStream);
```

In this case, we are telling getUserMedia that we want both audio and video access. Making this call will prompt the browser to ask the user for permission to access the camera and microphone on their computer, as shown in the following screen shot from the webinar tool example in this book.



If the user presses "Allow", then the combined stream is returned to the callback method ("gotStream" in the above code example).

That returned stream will then be assigned to a video tag in order to be displayed in the browser, something like this:

```
video.src = window.URL.createObjectURL(stream);
```

Those two lines of code are at the essence of WebRTC. As you'll see in the upcoming chapters, it's not really that simple. There's a whole lot of code we need to write in our webinar tool to enable those two magical lines above.

12. Coding: Signaling and Calling Attendees

12.1. Code Samples

Prerequisite code for this chapter:

https://github.com/agilityfeat/webinar-tool/tree/comments

Completed code for this chapter:

https://github.com/agilityfeat/webinar-tool/tree/calling

12.2. Introduction

It's time to get back to the code! In this chapter we'll get a basic WebRTC call working in our webinar tool. WebRTC is not really like a phone call … you can't just dial someone up [7]. You both need to be in the same application in order to initiate a call.

In our case we'll do that by allowing the presenter to call users who make a comment in the webinar tool. If an attendee doesn't make a comment, the presenter can't call them. In our implementation, we won't allow the attendees to call each other (that would be disruptive to our webinar!), but you can implement it so attendees can make calls too if you like. WebRTC doesn't care about our distinction of presenters/attendees, that is an abstraction we use only in the context of our webinar tool.

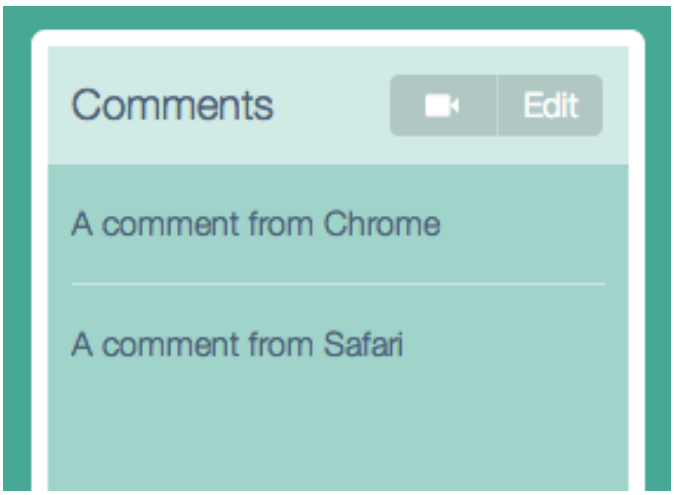
However, we can't just call any attendee unfortunately. Remember that in the last chapter we discussed browser support for WebRTC. As of this writing, it's going to work best in Chrome, Firefox, and Opera. Attendees of our webinars using Safari or Internet Explorer will still be able to see the webinar and make comments, but they can't participate in a WebRTC call.

The first thing we are going to do in the code is put an icon next to all comments from users in browsers that support WebRTC.

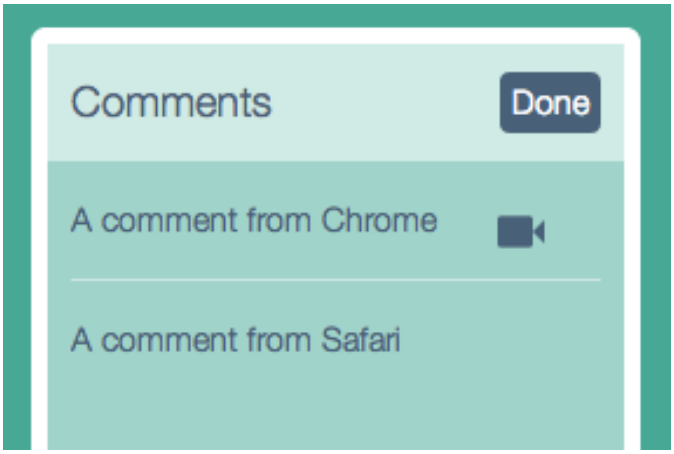
12.3. UI elements for starting a WebRTC call

Before we show you the code to do this, let's walk through how it will look from the presenter's perspective.

[7] Unless you get fancy and integrate your WebRTC application with a PSTN Gateway (PSTN is the Public Switched Telephone Network). This is possible and would allow you to create functionality similar to Google Voice where you can connect a web based call to a traditional phone number.



The presenter can click on the camera icon in the top right of the comments box (as shown above), and then a camera icon will appear next to each comment from a user in a WebRTC enabled browser.



In the figure above, you see that we made two comments, one from Chrome and one from Safari. Only the Chrome comment has a camera icon next to it, because Safari doesn't support WebRTC yet.

When the presenter wants to start a WebRTC call, they will do it by clicking on that camera icon. We'll get to that part later, but now let's take a look at the code we've added to determine if a user's browser supports WebRTC.

12.4. Enabling the camera button based on browser

In the commenting chapter, we already bound an event to the "Edit" button so we could delete offensive comments. Now we need to add a method to setBinds in webrtc_agility.js in order to bind the first camera button.

```
$(document).on("click", ".cameraCall", function(e){
        e.preventDefault();
        $(".cameraCall").parents(".initialCall").fadeOut();
        $(".deleteBtn").fadeOut();
        $(".doneBtn").addClass('blue').fadeIn();
        $(".commentItem").addClass('active');
})
```

This will make the camera button appear next to each comment, if it exists there to be seen. In the templates.html file, there was already code that checked if the cameraBtn should be displayed or not:

```
<% if(this_message.can_webrtc){%>
        <span class="cameraBtn"
                data-call-button="<%= this_message.from_uuid %>"
                data-user-username="<%= this_message.from %>"
                data-user="<%= this_message.from_uuid %>">
                <span class="glyphicon glyphicon-facetime-video"></span>
        </span>
<%}%>
```

This is checking a variable passed with each message called “can_webrtc.” Remember the “onChannelListMessage” method we have that handles receiving comments? In the case statement for the “MESSAGE” type, we were previously hardcoding the can_webrtc parameter to be false. Now we need to go change that to use the can_webrtc value passed with the message.

```
case "MESSAGE":
        console.log('Comment received - user can_webrtc: ' +
          message.user.can_webrtc);
        self.storeMessageAndDisplayMessages({
                from            : message.user.name,
                from_uuid       : message.user.uuid,
                message         : message.text.replace( /[<>]/g, '' ),
                id                      : message.id,
                can_webrtc      : message.user.can_webrtc,
                type            : message.type,
                is_your_message : (message.user.uuid === agility_webrtc.uuid)
        });
```

If we go back to the event handler for the “on click” method of the “#btn_send_message” control, we see that the can_webrtc parameter is populated from variable agility_webrtc.can_webrtc. Previously this was just being passed as undefined, because we never initialized it anywhere.

Now it’s time to initialize that variable and get ready to allow WebRTC calls. At the bottom of our init() function, we now want to add a line to “checkUserMedia()”.

```
init : function(){
        ...
        self.checkUserMedia();
},
```

This method checkUserMedia doesn’t exist yet, so define it just below the init function:

```
checkUserMedia : function(callback){
        agility_webrtc.can_webrtc = !!(navigator.getUserMedia ||
          navigator.webkitGetUserMedia || navigator.mozGetUserMedia
          || navigator.msGetUserMedia);
        console.log("This browser can use WebRTC: " + agility_webrtc.can_webrtc);
        if(agility_webrtc.can_webrtc === true){
                $.getScript("javascripts/resources/vendor/webrtc-beta-pubnub.js")
                        .done(function( script, textStatus ) {
                                if(typeof callback === 'function'){
                                        callback();
                                }
                        })
                        .fail(function( jqxhr, settings, exception ) {
                                console.log("there was an error");
                        }
                );
        } else {
                if(typeof callback === 'function'){
                        callback();
                }
        }
        return this;
},
```

In this method, the key line to notice is the following:

```
agility_webrtc.can_webrtc = !!(navigator.getUserMedia ||
  navigator.webkitGetUserMedia || navigator.mozGetUserMedia
  || navigator.msGetUserMedia);
```

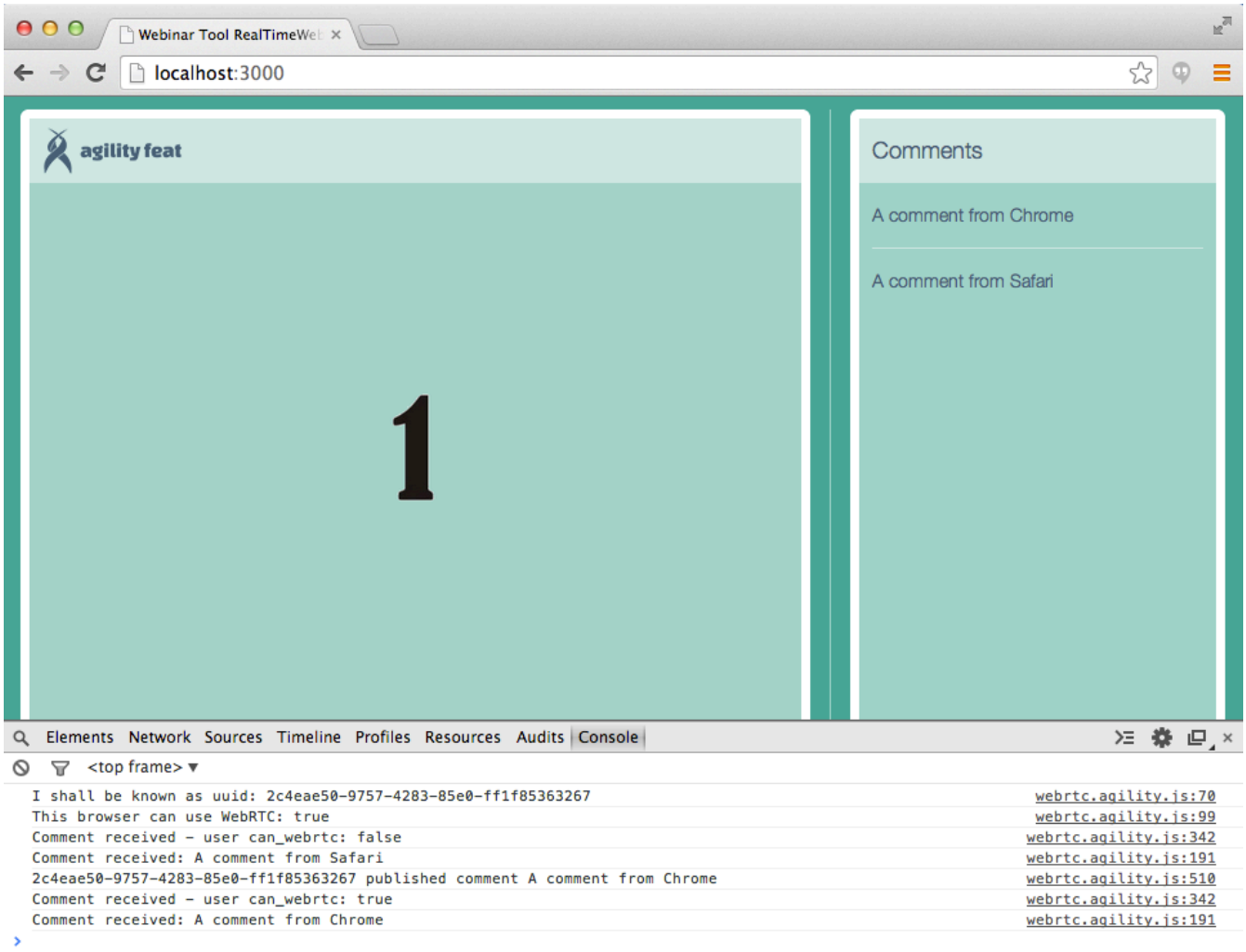
This line will populate can_webrtc with true if any one of the different options for getUserMedia is available in the current browser. While the WebRTC part of the HTML5 standard calls for a getUserMedia method, it's exact implementation may vary in each browser. This line of code, as of this writing, will let you know if any of the possible getUserMedia variants exist in the current browser.

If the browser does not support WebRTC, it's going to fall down to the bottom of the function where we can optionally go to a callback method if one was supplied.

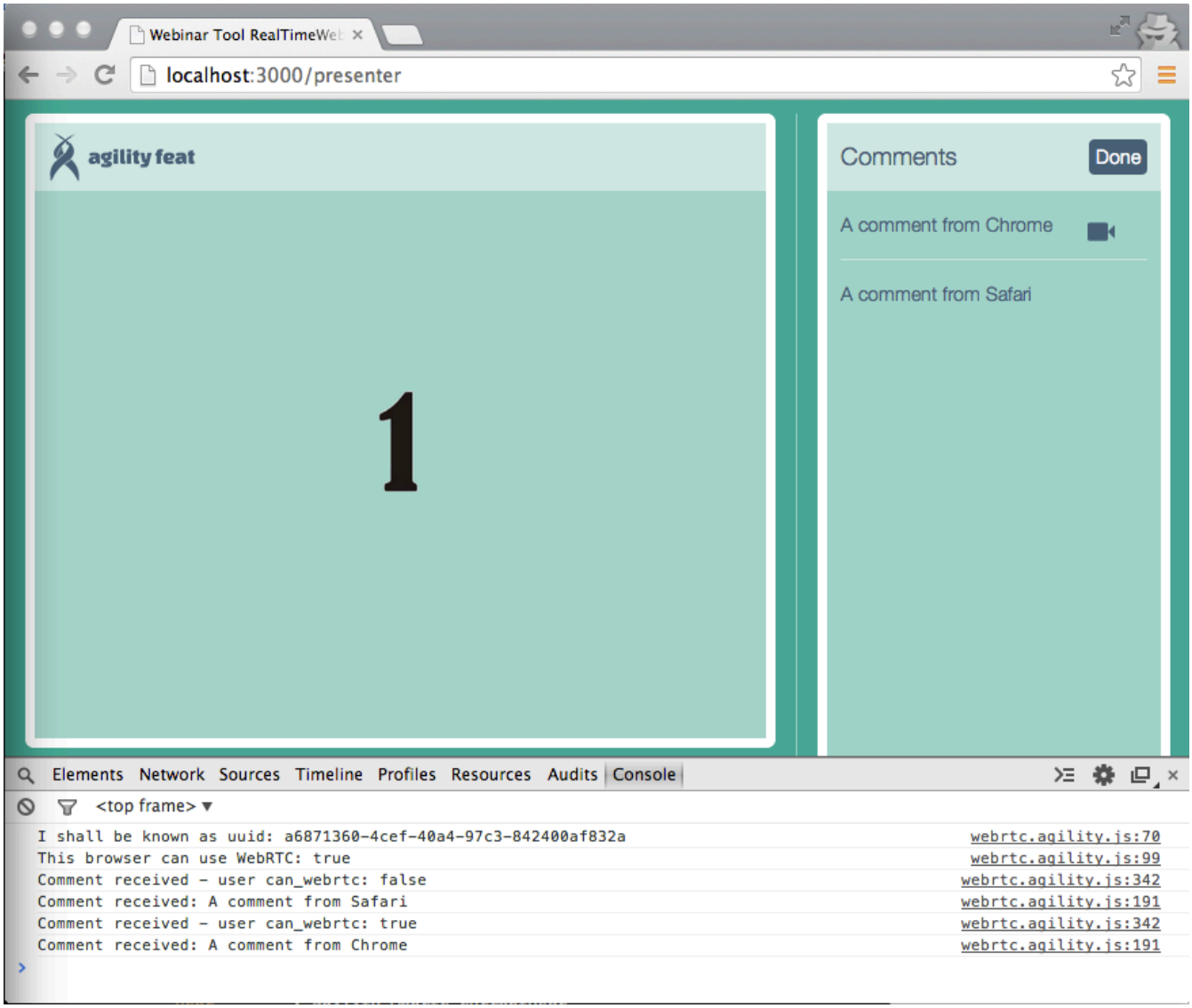
If the browser does support WebRTC, then we are going to load in the PubNub beta WebRTC javascript. We're using the PubNub API in our examples so that we don't need to configure our own STUN and TURN servers and those more complicated details of WebRTC. For most users of WebRTC in the future, we see developers using API's like PubNub rather than implementing the finer details of WebRTC that frankly the average web developer doesn't care about or really need to know.

At this point, you should be able to run the webinar tool in two different windows, and try making comments from different browsers. Try making a comment from Chrome and one from Safari, and you'll see that the video camera icon is only available next to the Chrome commenter. The following two screen shots show what your javascript console may look like with two Chrome windows open.

Note that we are using the Presenter's view in Chrome's "Incognito" mode so that it will be treated as an entirely different session.



Shown above: Attendee view with Chrome's javascript console showing value of can_webrtc variables



Shown above: Presenter view with Chrome's javascript console showing that only the Chrome comment is WebRTC enabled

12.5. Final design elements for the video call

That blue camera button is taunting us, but we've got some work to do before you can click it!

First we need to add in few things to the main view in views\index.ejs. We'll place the following code underneath the div tag where content is rendered from (That's the div tag with id "content").

We need to add an audio tag for playing a ringtone (the actual ringtone file should already be in your project).

```
<div class="top-container" id="content">
        <!--RENDERED CONTENT HERE -->
</div>
<audio loop id="ringer">
        <source src="medias/ring.mp3" type="audio/mpeg">
</audio>
```

Under that, we need to create several div tags which will serve as modal popups for when we are making a call, for the person receiving the call, and then a “conference” modal for when we are in the call together. Here’s the code we’re adding:

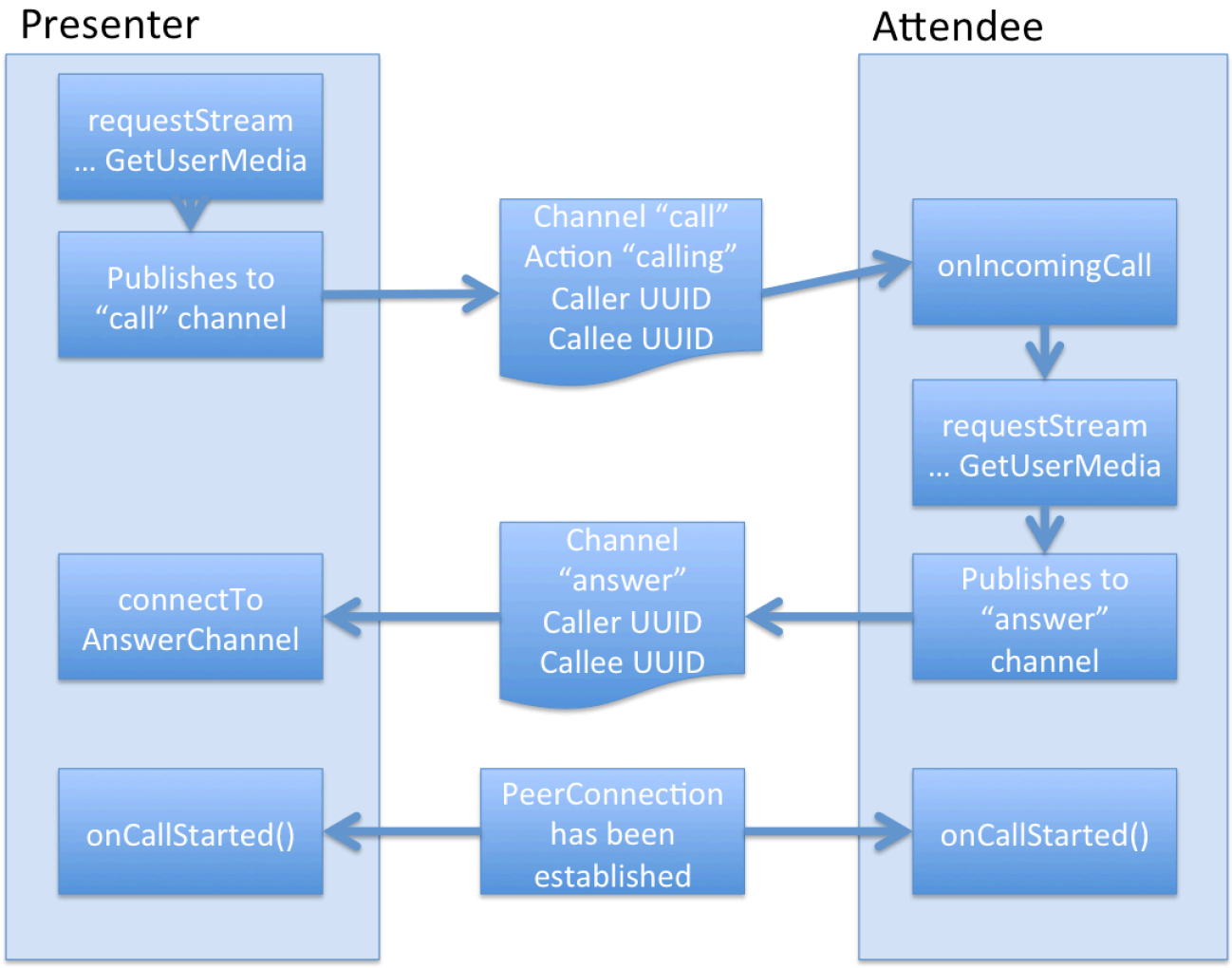
```
<div id="answer-modal" class="modal hide">
        <div class="modal-header">
                <button type="button" class="close" data-dismiss="modal"
                  aria-hidden="true">×</button>
                <h3 id="myModalLabel"><img src="images/phone.svg"
                  class="shake"/> Incoming Call</h3>
        </div>
        <div class="modal-body">
                <p class="caller">(User) is calling...</p>
        </div>
        <div class="modal-footer">
                <a href="#" class="btn btn-danger" id="ignoreCall">Hang Up</a>
                <button class="btn btn-success" id="answer">Answer</button>
        </div>
</div>
```

```
<div id="calling-modal" class="modal hide">
        <div class="modal-header">
                <button type="button" class="close" data-dismiss="modal"
                  aria-hidden="true">×</button>
                <h3 id="myModalLabel"><img src="images/phone.svg"
                  class="shake"/> Outgoing Call</h3>
        </div>
        <div class="modal-body">
                <p class="calling">Calling (User)...</p>
        </div>
        <div class="modal-footer">
                <button class="btn btn-danger" data-dismiss="modal"
                  aria-hidden="true">Cancel</button>
        </div>
</div>
```

```
<div id="conference-modal" class="modal hide">
        <div class="modal-header">
                <h3 id="myModalLabel"><img class="phone_icon"
                  src="images/phone.svg"/> Conference Call</h3>
        </div>
        <div class="modal-body">
        <div class="streaming_container pull-left">
                <img class="audio_controls" id="your_audio_mute"
                  src="/images/mute.svg"/>
                <img class="audio_controls" id="your_audio_unmute"
                  src="/images/unmute.svg"/>
                <video autoplay id="you"></video>
                <video autoplay id="me" muted></video>
        </div>
        </div>
        <div class="modal-footer">
                <div class="span12" id="video-controls" style="display: none;">
                        <a href="#" class="btn btn-danger" id="hangup">Hang Up</a>
                        <span id="time">00:00</span>
                </div>
        </div>
</div>
```

12.6. Overview of the calling process

With those design elements in place, we are almost ready to start a WebRTC video chat. Before we get into the code, let's examine this overview of the process we're going to follow.



The code may start to get confusing here, because we are using the same code base for two different roles: the Presenter and the Attendee. Walking through the diagram above should help clear this up.

When the Presenter clicks on the camera next to an Attendee's comment in order to initiate a call, the first thing that happens is the javascript will request the Presenter's video and audio stream. This is going to be done with a call to getUserMedia().

After that, we'll use the PubNub WebRTC API to publish a "call" message. This will signify to all users of the webinar tool that the Presenter is calling someone, but only the callee who matches the given Callee UUID will answer the call.

After the Attendee's code answers that call, their javascript will make a call to getUserMedia(), in this case retrieving the video and audio streams of the Attendee.

The Attendee's javascript will publish an "answer" via the PubNub WebRTC API, as well as publish their own stream. At this point, the Presenter is also publishing a reference to their video stream when they receive the answer.

With streams exchanged and displayed in the browser, the PubNub WebRTC API will trigger an onCallStarted event in both browsers, and our call is now a Peer to Peer WebRTC video chat!

Our diagram is going to get a little more complicated than this as the code comes together, but just keep the above exchange in mind as we start to flesh this out.

12.7. Starting a call

Let's get this call working! Head back to our friendly (and growing!) webrtc.agility.js file.

In there, we need to define a "streams" variable that we will use for storing a reference to the actual WebRTC streams. In addition, we'll put in some JSON for configuring the video for several different options. These video_constraints store three options for the size (and therefore quality) of the video we want to use. We're going to default to the qvga, or lowest definition of our three options.

```
streams         : [],
video_constraint_default : "qvga",
video_constraints       : [{
        name : "qvga",//Low def
        constraints : {
                mandatory : {
                        maxWidth        : 320,
                        maxHeight       : 180
                }
        }
},{
        name : "vga",//Regular
        constraints : {
                mandatory : {
                        maxWidth        : 640,
                        maxHeight       : 360
                }
        }
},{
        name : "hd",//Regular
        constraints : {
                mandatory : {
                        maxWidth        : 1280,
                        maxHeight       : 720
                }
        }
}],
```

Now let's jump down to the bottom of the file and revisit our setBinds() method. We need to add in a binding here for when the Presenter clicks on the blue camera next to an attendee.

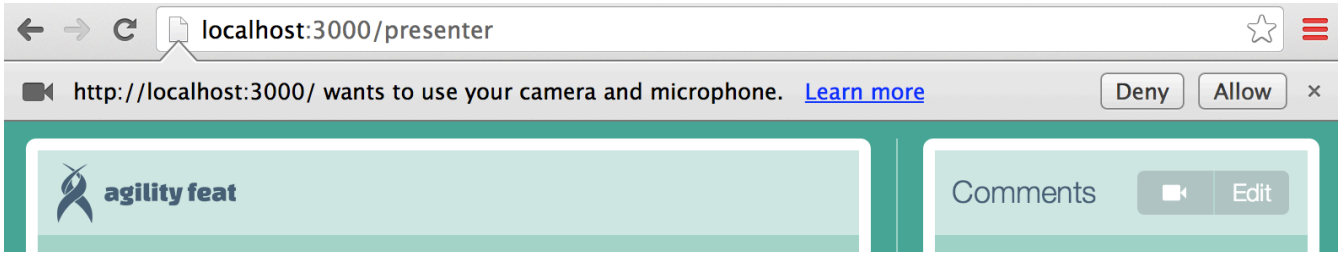
```
$(document).on("click", "[data-user]", function(e){
        e.preventDefault();
        e.stopPropagation();
        $(".doneBtn").trigger("click");
        $(this).parents(".commentItem")
                .find(".glyphicon-hand-up")
                .removeClass("bouncing").hide();
        var name;
        var callingTo = {
                uuid            : $(this).data('user'),
                username        : $(this).data('user-username')
        }
        if(agility_webrtc.currentUser
                .db.get('is_presenter') === "true"){
                console.log('Going to call ' +
                        callingTo.username + ' ' + callingTo.uuid);
                agility_webrtc.callPerson(callingTo);
        } else {
                var video_constraints =
                        _.find(agility_webrtc.video_constraints,
                                function(video_constraint){
                                        return video_constraint.name ===
                                        agility_webrtc.video_constraint_default;
                }).constraints;
                agility_webrtc.requestStream({
                        video : video_constraints,
                        audio : true
                }, function(stream){
                        var my_stream = _.find(agility_webrtc.streams,
                                function(stream){
                                        return stream.who === "mine";
                        })
                        if(my_stream){
                                //Stream exists in the streams array,
                                //let's update the reference...
                                my_stream.stream = stream;
                        } else {
                                console.log("pushing stream 'mine'");
                                agility_webrtc.streams.push({ who : "mine",
                                        stream : stream });
                        }
                        agility_webrtc.callPerson(callingTo);
                }, function(){
alert("To call someone please allow access to audio and video...");
                })
        }
});
```

The first thing to notice here is the requestStream call. We pass in the video options (defaulted to that lowest size video to save on bandwidth), and we are also requesting access to the microphone by indicating “audio: true”.

The requestStream method needs to be added into the webrtc.agility.js file, we’ll put it underneath the checkUserMedia method.

```
requestStream : function(options,callback, errorCallback){
        var is_presenter = agility_webrtc.currentUser ?
          agility_webrtc.currentUser.db.get('is_presenter') ===
          "true" : false;
        var stream = _.find(agility_webrtc.streams, function(stream){
          return stream.who === (is_presenter ? "presenter" : "mine"); });
        if(stream != null){
                callback(stream.stream);
        } else {
                navigator.getUserMedia = ( navigator.getUserMedia ||
                  navigator.webkitGetUserMedia || navigator.mozGetUserMedia ||
                  navigator.msGetUserMedia);
                if(navigator.getUserMedia != null){
                        navigator.getUserMedia(options, function(stream) {
                                if(typeof callback === 'function'){
                                        callback(stream);
                                }
                        }, function(e) {
                                console.log('No access to getUserMedia!', e);
                                if(e.name === "PermissionDeniedError" &&
                                  window.location.protocol !== "https:"){
                                        alert("Must be behind a SSL...");
                                }
                                if(typeof errorCallback === 'function'){
                                        errorCallback(e);
                                }
                        });
                }
        }
},
```

Assuming this is the first time we have tried to start a call, and our video stream does not already exist in our streams array, then we are going to do some getUserMedia checks similar to the checkUserMedia function. This matters here because the Presenter is going to be asked now if they are willing to share their video and audio, like shown in the following figure.



If the user presses Deny, then the getUserMedia is not going to return a valid stream and we will present pop ups to the Presenter letting them know they have to Allow the camera and microphone before proceeding. Once they've denied access to this, the Presenter will probably have to close and reopen their browser in order to be presented with the permissions bar again.

Normally of course, the Presenter will press the Allow button, which means that requestStream will now pass the approved stream object back to the callback function.

Back in our onclick event for the camera, notice these lines within the callback of the requestStream call:

```
agility_webrtc.requestStream({
        video : video_constraints,
        audio : true
}, function(stream){
        // ... surrounding code omitted here for clarity
                console.log("pushing stream 'mine'");
                agility_webrtc.streams.push({ who : "mine", stream : stream });
        // ... surrounding code omitted here for clarity
        agility_webrtc.callPerson(callingTo);
}
```

The callback method stores the video/audio stream in our streams array along with an indication that the stream belongs to the current user (in this case, *mine* indicates the Presenter).

In the final part of that callback, we invoke a method “callPerson”, and pass in who we are calling. The unique identifier for who we are calling was stored in the div tag when the comment was displayed originally in the “storeMessageAndDisplayMessages” function.

Next we need to create the callPerson method, which is where we will use the PubNub API to publish the "call" method referred to in our overview diagram from above.

The callPerson method lives in webrtc.agility.js, we’ve put it just below the requestStream method.

```
callPerson      : function(options){
        console.log('Going to call ' + options.username + ' ' + options.uuid);
        agility_webrtc.currentCallUUID = options.uuid;
        var modalCalling = $("#calling-modal");
        var message = "Calling " + options.username + "...";
        modalCalling.find('.calling').text(message);
        modalCalling.find(".btn-danger").data("calling-user", options.uuid);
        modalCalling.modal('show');
        $("#ringer")[0].play();
        modalCalling.removeClass("hide");
        console.log('Publishing call channel message. Caller: '
                + agility_webrtc.uuid + ' Callee: ' + options.uuid);
        agility_webrtc.currentUser.publish({
                channel: 'call',
                message: {
                        caller  : {
                                uuid : agility_webrtc.uuid,
                                username : agility_webrtc
                                        .currentUser.db.get("username")
                        },
                        callee  : {
                                uuid : options.uuid,
                                username : options.username
                        },
                        action  : "calling"
                }
        });
},
```

In the beginning of callPerson, we’re doing some visual house cleaning. We make a modal visible to show that we are calling someone, and we play the ringtone audio.

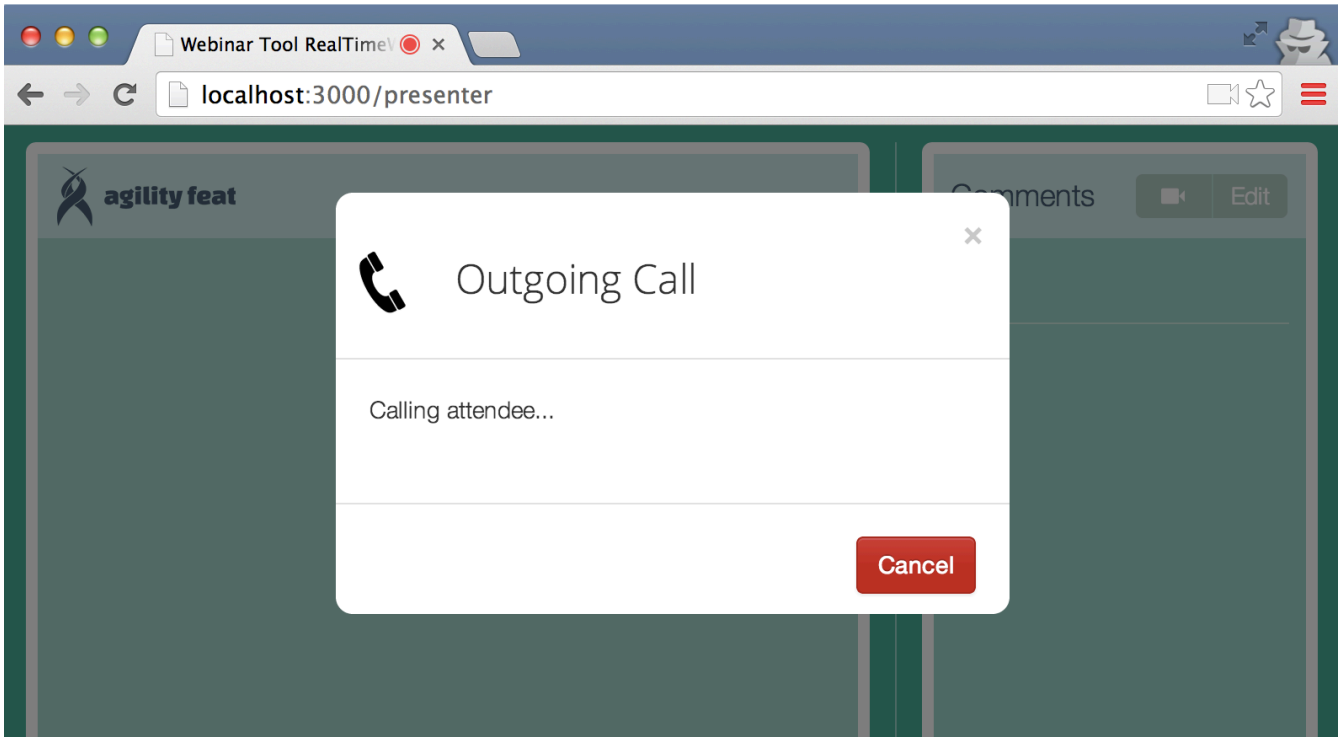
After that comes our implementation of signaling in this application. Remember from the previous chapter that the WebRTC intentionally does not tell you how to handle the signaling, or handshaking,

process that is necessary for two parties to exchange WebRTC streams. Since we're already using PubNub for messaging between the browsers, it makes sense for us to also use PubNub for our WebRTC signaling.

We just define a JSON formatted message for the "call" channel on our PubNub account which says who is calling whom, and the unique identifiers for both parties.

That message is published to the PubNub channel, and we have now done everything we need to on the Presenter's side to start a call! No one is going to be able to answer the call yet, but this is still a good point to check your code so far.

If you run "node server" and bring up an Attendee and Presenter view in two different windows of Chrome, then have the Attendee make a comment to the webinar. The Presenter clicks on the video camera next to the Attendee's comment, and now they should have the "calling" modal appear, complete with ringtone sound!



When you get tired of basking in the glory that is our ringtone file, just hit Cancel. No one's going to answer this call yet, because we still need to write the receiving code.

12.8. Receiving a call

Remember that our signaling is done through PubNub messaging, and so we need to put a statement in the init() function to subscribe to a channel for making calls, and a channel for answering calls. We'll start with the channel for making calls.

In the init() function, we're going to add in a checkUserMedia() call to initialize the UUID's that we need for the WebRTC calls and to wire up the events we'll need later and publish our streams:

```
self.checkUserMedia(function(){
        agility_webrtc.credentials.uuid = PUBNUB.uuid();
        agility_webrtc.currentUser = PUBNUB.init(agility_webrtc.credentials);
        agility_webrtc.uuid = agility_webrtc.currentUser.UUID;
        console.log("I shall be known as uuid: " + agility_webrtc.uuid);
        agility_webrtc.currentUser.subscribe({
                channel         : agility_webrtc.channelName,
                callback        : agility_webrtc.onChannelListMessage
        });
        agility_webrtc.connectToCallChannel();
        if(agility_webrtc.currentUser.onNewConnection){
                agility_webrtc.currentUser.onNewConnection(function(uuid){
                        console.info("onNewConnection triggered...");
                        agility_webrtc.publishStream({ uuid : uuid });
                });
        }
});
```

The connectToCallChannel() method doesn't exist yet, so let's create that underneath our callPerson() method:

```
connectToCallChannel : function(){
  agility_webrtc.currentUser.subscribe({
    channel: 'call',
    callback: function(call) {
      switch(call.action){
        case "calling":
          if(call.caller.uuid !==
            agility_webrtc.uuid && call.callee.uuid ===
            agility_webrtc.uuid){
            agility_webrtc.incomingCallFrom =
              call.caller.uuid;
            agility_webrtc.onIncomingCall({
              caller     : call.caller.username,
              call_type   : call.action
            });
          }
        break;
      }
    }
  });
},
```

And here's the onIncomingCall method that is referenced by connectToCallChannel:

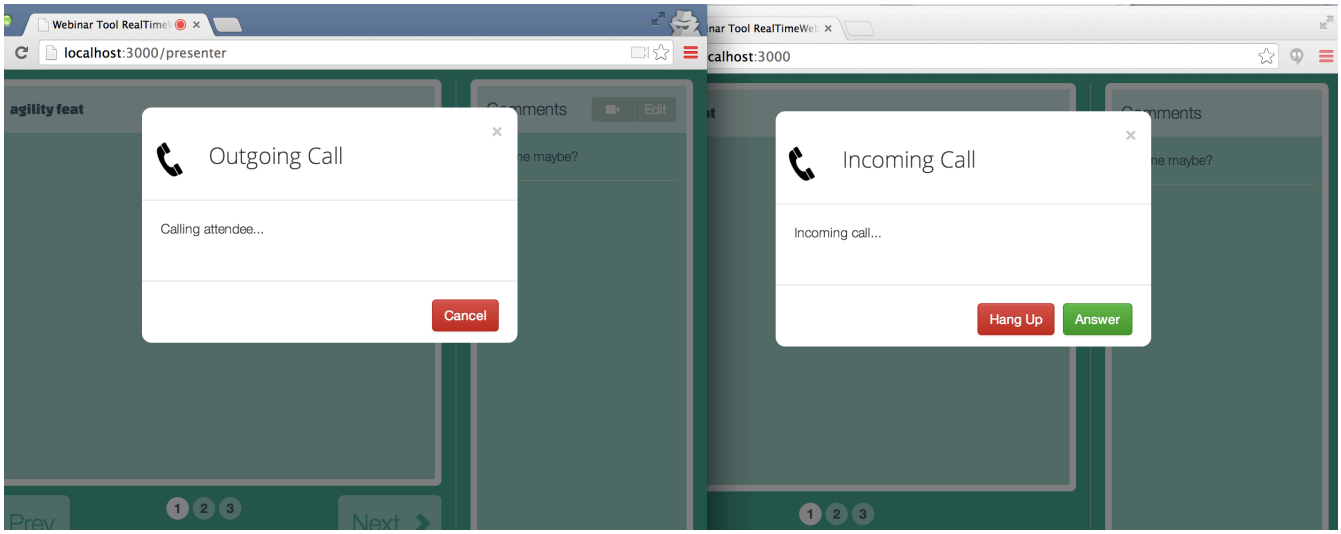
```
onIncomingCall : function(options){
        var modalAnswer = $("#answer-modal");
        modalAnswer.removeClass("hide");
        var message = "Incoming call...";
        modalAnswer
                .removeClass("hide").find('.caller')
                .text(message)
                .end().find('.modal-footer').show().end().modal('show');
        $("#ringer")[0].play();
},
```

The connectToCallChannel method starts out by subscribing to a channel called "call". When a call message is sent to that channel, the callback method calls the onIncomingCall method, which will display the answer modal on the Attendee's side.

Note that anyone subscribed to that call channel would get the message, but this statement makes sure that only the Attendee will answer it:

```
call.callee.uuid === agility_webrtc.uuid
```

At this point, you still can't answer the call, but you can at least see that a call is coming in. Here's what it should look like.



Refresh your pages to get rid of the ringer noise, and let's go hook up that Answer button!

12.9. Answering a call

In addition to subscribing to the Call channel in the checkUserMedia() method, we also need to subscribe to an Answer channel. Add in a statement to call agility_webrtc.connectToAnswerChannel():

```
self.checkUserMedia(function(){
        agility_webrtc.credentials.uuid = PUBNUB.uuid();
        agility_webrtc.currentUser = PUBNUB.init(agility_webrtc.credentials);
        agility_webrtc.uuid = agility_webrtc.currentUser.UUID;
        console.log("I shall be known as uuid: " + agility_webrtc.uuid);
        agility_webrtc.currentUser.subscribe({
                channel         : agility_webrtc.channelName,
                callback        : agility_webrtc.onChannelListMessage
        });
        agility_webrtc.connectToCallChannel();
        //Here's the new call to add:
        agility_webrtc.connectToAnswerChannel();
        if(agility_webrtc.currentUser.onNewConnection){
                agility_webrtc.currentUser.onNewConnection(function(uuid){
                        console.info("onNewConnection triggered...");
                        agility_webrtc.publishStream({ uuid : uuid });
                });
        }
});
```

And here is the implementation of connectToAnswerChannel, which is a pretty straightforward subscription to a PubNub channel, with a callback that will end up publishing the Presenter's video when an answer is received from the Attendee.

```
connectToAnswerChannel : function(){
        agility_webrtc.currentUser.subscribe({
                channel: 'answer',
                callback: function(data) {
                        if (data.caller.uuid === agility_webrtc.uuid) {
                                console.log('Received an answer message from '
                                        + data.callee.uuid);
                                agility_webrtc.publishStream({ uuid :
                                        data.callee.uuid });
                                $("#calling-modal").modal('hide');
                                $("#ringer")[0].pause();
                                //The user answered the call
                                agility_webrtc.onCallStarted();
                        }
                }
        });
},
```

Before the Attendee can actually answer the call though, we need to wire up that Answer button! At the bottom of setBinds, add in an event binding for answering the call:

```
$(document).on("click", "#answer", function(e){
        e.preventDefault();
        e.stopPropagation();
        console.log('I have clicked to answer the call');
        agility_webrtc.answerCall(agility_webrtc.incomingCallFrom);
})
```

We need to add in the answerCall() method next in order for the Attendee to give permission for their Video/Audio streams to be used, and then to publish an “answer” back to the Presenter saying they are ready to take the call.

```
answerCall : function(from){
        var video_constraints = _.find(agility_webrtc.video_constraints,
          function(video_constraint){
                return video_constraint.name ===
                  agility_webrtc.video_constraint_default;
        }).constraints;
        agility_webrtc.requestStream({
                video : video_constraints,
                audio : true
        }, function(stream){
                agility_webrtc.currentCallUUID = agility_webrtc.incomingCallFrom;
                var my_stream = _.find(agility_webrtc.streams, function(stream){
                        return stream.who === "mine";
                })
                if(my_stream){
                        my_stream.stream = stream;
                } else {
                        agility_webrtc.streams.push({ who : "mine",
                          stream : stream });
                }
                agility_webrtc.publishStream({ uuid : from  });
                var modalAnswer = $("#answer-modal");
                modalAnswer.modal("hide");
                $("#ringer")[0].pause()
                //Continued on next page
```

```
                //Continued from previous page
                agility_webrtc.currentUser.publish({
                        channel: 'answer',
                        message: {
                                caller: {
                                        uuid : agility_webrtc.incomingCallFrom
                                },
                                callee: {
                                  uuid : agility_webrtc.uuid,
                                  username : agility_webrtc.currentUser
                                    .db.get("username")
                                }
                        }
                });
        }, function(){
                alert("Unable to access stream");
        })
},
```

As part of this method, the Attendee (in our use case, this is always the person receiving the call) needs to publish their own stream too, which happens in the publishStream method:

```
publishStream : function(options){
        var stream = _.find(agility_webrtc.streams, function(stream){
          return stream.who === "mine"; });
        var resumeStreaming = function(stream){
                agility_webrtc.incomingCallFrom = options.uuid;
                agility_webrtc.showStream({ who : "mine" , container : '#me'});
                agility_webrtc.currentUser.publish({
                        user: options.uuid,
                        stream: stream
                });
                agility_webrtc.currentUser.subscribe({
                        user: options.uuid,
                        stream: function(bad, event) {
                                var remote_stream =
                                  _.find(agility_webrtc.streams, function(stream){
                                        return stream.who === "you";
                                })
                                if(remote_stream){
                                        remote_stream.stream = event.stream;
                                } else {
                                        agility_webrtc.streams.push(
                                          { who : "you", stream : event.stream });
                                }
                                agility_webrtc.showStream(
                                  { who : "you" , container : '#you'});
                                $("#conference-modal").removeClass("hide")
                                  .modal("show");
                                agility_webrtc.onCallStarted();
                        },
                        disconnect: function(uuid, pc) {
                                //The caller disconnected the call...
                                //Let's just hide the conference
                                agility_webrtc.onEndCall();
                        }
                });
        }
        //Continued on next page
```

```
        //Continued from previous page
        if(stream == null){
                var video_constraints = _.find(agility_webrtc.video_constraints,
                  function(video_constraint){
                        return video_constraint.name ===
                          agility_webrtc.video_constraint_default;
                }).constraints;
                agility_webrtc.requestStream({
                        video : video_constraints,
                        audio : true
                }, function(stream){
                        var my_stream = _.find(agility_webrtc.streams,
                          function(stream){
                                return stream.who === "mine";
                        })
                        if(my_stream){
                                my_stream.stream = stream;
                        } else {
                                agility_webrtc.streams.push({ who : "mine",
                                  stream : stream });
                        }
                        resumeStreaming(stream);
                }, function(){
                        alert("Unable to get stream");
                })
        } else {
                resumeStreaming(stream.stream);
        }
},
```

The method showStream is also necessary to include here, and this is where the real magic of sending the stream to a video element happens.

```
showStream  : function(options){
        var stream = _.find(this.streams, function(stream){
                return stream.who === options.who;
        }).stream;
        var video = $(options.container)[0];
        if(video){
                video.src = window.URL.createObjectURL(stream);
                $(video).fadeIn(300);
        }
},
```

This method is going to be used for both the Presenter and Attendee streams to actually put that video stream into the src of a video tag for display in the browser. That is happening in this line:

```
video.src = window.URL.createObjectURL(stream);
```

In presentations about WebRTC, I am guilty sometimes of over-selling all this by showing that one line of code and saying something to the effect of “that’s basically it – that’s WebRTC!”

As you well know by now, there is a lot more to WebRTC than that video tag. But I can’t let this moment in our coding go by without a passing remark that this is really the point we have been building to with most of this book. Yes, there is a lot of coding to build any real application, and there is a lot of handshaking that needs to go on around WebRTC.

In the end however, that line of code represents the technical beauty of what we are building. It all comes down to getting someone else's video stream, and displaying that in your browser, with a simple javascript statement.

12.10. Final details when the call starts

After the Attendee has published that "Answer" message, the connectToAnswerChannel method that we already put in the code gets called on the Presenter's side. Just as a reminder, that method looks like:

```
connectToAnswerChannel : function(){
        agility_webrtc.currentUser.subscribe({
                channel: 'answer',
                callback: function(data) {
                        if (data.caller.uuid === agility_webrtc.uuid) {
                                console.log('Received an answer message from '
                                        + data.callee.uuid);
                                agility_webrtc.publishStream({ uuid :
                                        data.callee.uuid });
                                $("#calling-modal").modal('hide');
                                $("#ringer")[0].pause();
                                //The user answered the call
                                agility_webrtc.onCallStarted();
                        }
                }
        });
},
```

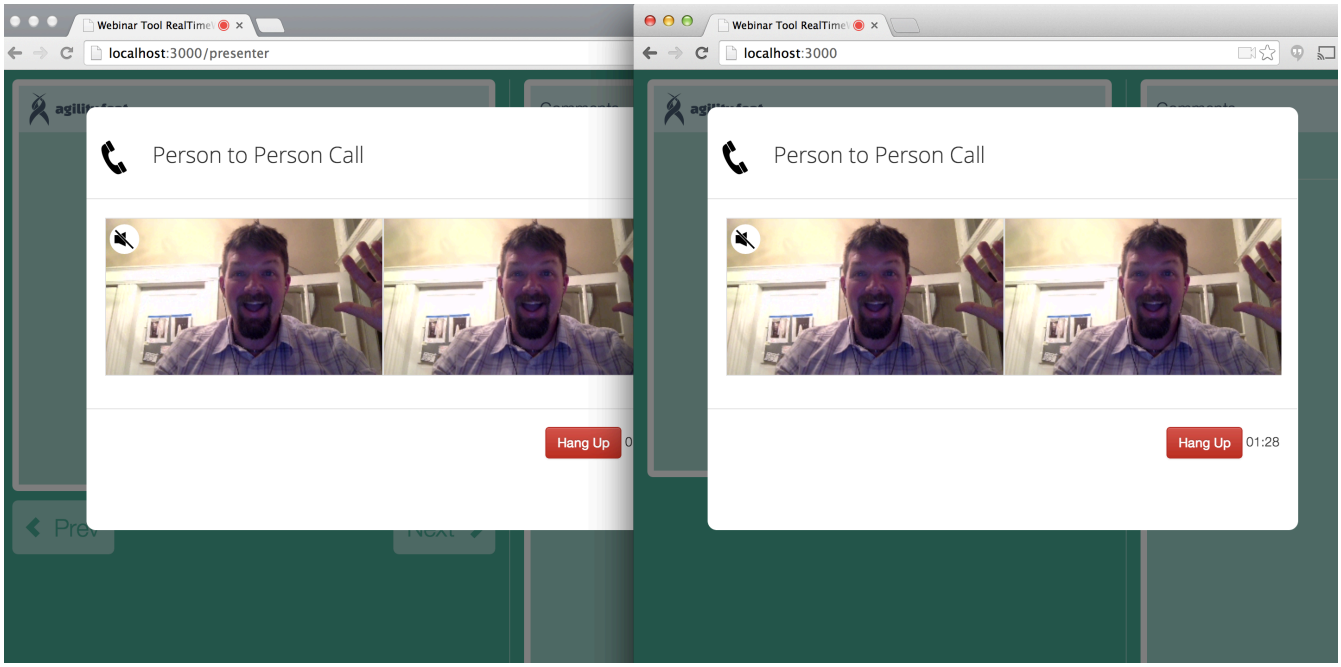
Now that we see where this method fits, we'll briefly explain it. The code verifies that you are in fact the Presenter (data.caller.uuid === agility_webrtc.uuid), and then publishes back the Presenter's stream via the PubNub WebRTC API.

Finally, the "calling" modal is hidden and the ringer silenced. The onCallStarted() method is triggered, which just displays some video controls and starts a timer for how long we have been talking:

```
onCallStarted           : function(){
        console.log('onCallStarted!');
        $('#video-controls').show();
        $('#video-controls #time').html("00:00");
        agility_webrtc.currentCallTime = 0;
        clearInterval(agility_webrtc.timeInterval);
        agility_webrtc.timeInterval =
                setInterval(agility_webrtc.incrementTimer, 1000);
},
```

12.11. Completing the WebRTC call

At this point Presenters should be able to call Attendees who make a comment from WebRTC enabled browsers. Your first real time communications application is complete!



If you had any trouble completing the code in this chapter, you can look at the Master of the GitHub repository, or the "calling" branch:

https://github.com/agilityfeat/webinar-tool/tree/calling

13. Conclusion

You've made it! Now you are armed with some hands on experience building a real time web application, and ready to enter the next era of software development.

Remember if for any reason you don't have the code working yet, you can download the latest copy of our code at:

https://github.com/agilityfeat/webinar-tool

If by some small chance your goal in reading this book was specifically to build a webinar application and commercialize it, then you should be off to a good start!

For the vast majority of you though, this webinar tool was never really the point. The point was to get comfortable with some real-time messaging concepts, learn some NodeJS, and start to work with WebRTC for real time communications in the browser. We hope you have met your goals.

What's next? We humbly recommend that you join our RealTimeWeekly.com mailing list. This is our weekly attempt to let you know about the latest blog posts related to the technologies this book is based on, as well as additional real-time software technologies.

We'd love to hear more about what you're building, how this book may have helped you, and your suggestions for improvement. Please contact us at Arin@AgilityFeat.com [mailto:Arin@AgilityFeat.com].