# Development testing

TODO: should I have a sort of “concluding” paragraph under each level 3 subsection?

This chapter will focus on the implementation of the test application described in chapter (TODO: crossref testapp methodology). Each framework has its own subsection where I describe every aspect of the development and thoughts I made during the process. Each section has its own summary, and finally, there is a conclusion giving a nuanced look at what framework solves the different tasks of a real-time application best.

## Socket.IO

Socket.io is a module for Node.js (TODO: nodejs) that provides real-time through pure JavaScript on both server and client. It has been around since 2011 (TODO: last commitpage)[[1]](#footnote-1), and it aims to provide clean and simple real-time across all platforms:

*“Socket.IOaims to make realtime apps possible in every browser and mobile device, blurring the difference between the different transport mechanisms”.*

While it hasn’t reached 1.0 yet (TODO: check), it is used in production code by several companies, and it is considered stable. Perhaps one of the most “famous” applications that use Socket.IO is Trello – the online “Scrum Board”[[2]](#footnote-2).

TODO: made by + licence (open source, free)

As mentioned, Socket.IO is a module for Node.js. But what exactly is Node.js? As the name implies, it has to do with JavaScript. More precisely, Node.js is a framework for building network applications using JavaScript. In other words, it is the framework that has made JavaScript enter the realm of backend languages like C# and Java.

Node builds on Google Chrome’s JavaScript runtime (TODO: nodejs) and the project has been around for some time. Still, it is not finished and is currently (TODO: write date?) in version 0.10.15. However, its Documentation has indicators on each aspect, showing how stable it is, hence making it rather easy to use in a safe way.

### Why I chose it

Node.js is increasingly popular, and the idea of using JavaScript on the server is very exciting! Over the past couple of years, there has been a dramatic change in the way developers think of JavaScript (TODO: sources). Therefore, it was only natural that I chose at least one framework that uses Node.js as server.

Though there are several modules for Node that provides real-time (TODO: link to node modules), Socket.IO stands out from the crowd. It seems to have the largest community, as it is frequently featured at conferences and generally mentioned many times in traditional forums like Stack Overflow[[3]](#footnote-3).

Furthermore, Socket.IO feels like more than just a Node module. It has its own homepage (TODO: homepage) with some examples and demos–all presented in a good looking and easy to understand fashion. I feel this gives Socket.IO a more professional impression, which makes it stand out even more from some of the other modules that exist that seem more like something someone threw together in a hurry.

Socket.IO doesn’t have a lot of documentation, but what it has gives users a quick overview of the module and how to use it. The API documentation (TODO: docs) uses code samples, which I find more useful than a so-called “wall of text”. There is also a wiki page (TODO: wiki) to give information beyond the API documentation.

As stated in the quote from Socket.IO’s homepage in the introduction to this section, Socket.IO strives to blur the difference between the different transport mechanisms. WebSockets is the preferred transport, but if the client doesn’t support it, Socket.IO will fall back gracefully[[4]](#footnote-4) to one of the following transports:

* Adobe Flash Socket (TODO: source), which uses, surprise, Flash to establish a TCP socket connection between the client and the server, thus “mimicking” a WebSocket connection.
* Ajax multipart streaming (TODO: source): An alternative streaming technique to the forever frame technique described in section (TODO: crossref AND should I write this in the essaypart?).
* Forever Frame
* JSONP Polling, which is polling with data type set to JSONP. This allows cross domain requests; something that is not allowed in normal HTTP Polling (TODO: same source as multipart).

TODO: table, summarize why I chose it?

### How it works

TODO!

### Getting started

Having Node.js installed on your computer, installing Socket.IO is done via a simple command to the Node Package Manager (*NPM*)[[5]](#footnote-5). After that you can require it in any JavaScript file in your project (TODO: codelisting).

If you are new to Node.js, the learning curve is somewhat steep. However, this is almost always the case for other frameworks (not just real time frameworks) as well–it is expected that you know how to use the underlying technology.

Still, Socket.IO provides only simple examples, that demonstrates quite simple behavior, on their homepage (TODO: source) and on GitHub (TODO: source). All of these uses just a single HTMLfile, and a single JavaScript file on the server, a case which is quite uncommon in normal web applications. I missed some more information about how to build more complex apps, or at least some more reference to the other frameworks that are used in Socket.IO’s examples (like Express (TODO: source)).

### Coding environment

As JavaScript code is traditionally just the client part of a web application, it is often written in the same editor as the server code[[6]](#footnote-6). That may be why most of the examples I could find in videos throughout the web either use a Linux based text editor like Vim or Emacs or the excellent Sublime Text[[7]](#footnote-7), a cross platform editor that has become increasingly popular.

There is also an IDE provided by JetBrains[[8]](#footnote-8) under the name of “WebStorm IDE”, which is designed specifically for JavaScript, HTML and CSS. It also has a plugin that allows for Node.js development. This is the environment I chose to use, as it gives good IntelliSense (TODO: explain + crossref appendix), has good syntax highlighting and lets you debug Node.js applications (TODO: figure).

When it comes to debugging, there are a number of options with Node.js applications in addition to WebStorm. Since Node.js is built on Chrome’s JavaScript Runtime (TODO: ref Node’s homepage), that exposes an extensive debugger over TCP, you can build your own debugger. This is exactly what has been done with the Node Inspector (TODO: ref Git). Using this, you can use Chrome’s familiar in-browser debugger to debug your Node.js code (TODO: figure). I actually found this to work better than the WebStorm debugger for certain cases, especially with functions that were used to get back data from a database.

Another option is to use the debugger that comes bundled with Node.js. This is a command line tool that was surprisingly easy and intuitive to use. However, it requires you to write the keyword “debugger”, in your code instead of setting breakpoints (TODO: codelist), so it really only works for simple cases, where you don’t have to clean up a lot of lines with “debugger” afterwards.

Running a Node.js application is normally done via the command line. You just simply “node” (TODO: cmd list) the main file of your application and navigate to the port it is listening to in the browser. However, when using WebStorm you get the more traditional option of pressing a play button. All output that would normally show in the console, then appears within the IDE’s output instead (TODO: figure)–something I found helpful since it meant one less window to toggle between while testing the application manually.

### Code structuring

Perhaps the most unfamiliar aspect of JavaScript compared to other languages, is the fact that it is asynchronous (TODO: examplecode)[[9]](#footnote-9). A common pitfall for JavaScript frameworks is to not fully disclose, either via documentation or function names, whether a function is asynchronous or not. With Socket.IO this is, thankfully, not the case. Socket.IO follows the WebSockets protocol tightly as it provides and event based architecture, and events are always asynchronous. While the WebSockets API only provides a few, standard events (TODO: crossref), Socket.IO lets you used self named events in addition to the standard WebSocket API events.

With traditional, object oriented, languages like Java and C#, code is structured into separate classes, which are normally given their own files. In Node.js, code is structured through the use of modules. Modules is a natural way of separating code within different domains[[10]](#footnote-10), and it is also a very nice way of separating logic that can be used in other applications (TODO: require code list). Furthermore, modules can depend on other modules, which makes you able to build your application using modules as small building blocks. This is how Socket.IO is built up–it is a module, but it depends on a lot of other, smaller modules.

This modularization has both good and not so good aspects. The best aspect of the modularity is diversity: If you need some functionality, for instance a module for communication with a MySQL database, you will almost always not just find it, but find many different alternatives. Using modules also allows for an easy way to contribute to Node.js by making your own modules. This allows for a rather rapid growth of the Node.js project.

However, with so many modules, and seemingly little quality control, developers might end up using quite some time just to find a module that fits their requirements. Furthermore, some modules that are displayed on Node.js’s module page (TODO: source), are no longer maintained, which easily can cause problems if you have used it in production code and a bug arises.

Another problem is that modules often tend to favor the use of other modules in their examples and other documentation. This was also the case for Socket.IO, which mostly uses Express. That meant I had to learn more than what I set out to do.

### Serialization

When working with JavaScript on the, client the preferred serialization format is, in my opinion, *JSON* (JavaScript Object Notation). With Socket.IO, which uses JavaScript also on the server, I expected serialization to be automatic and abstracted away from me as a developer. Luckily, I was not disappointed, as this is exactly how Socket.IO has solved serialization of objects. When sending data back and forth, one simply sends values, objects or arrays without concern of how they are serialized or deserialized (TODO: codelist). The only problem I had, was when sending Date objects back and forth. This was not Socket.IO’s, nor Node.js’s fault, though, as it has to do with how JavaScript handles deserialization of Date objects[[11]](#footnote-11).

### Maturity

Socket.IO is not supported by any large company or otherwise backed by near endless funds. It was started by mainly one person, who is still the largest contributor to the project. It hasn’t reached version 1.0 yet, and development seems to be slowing to a crawling pace (TODO: source - git).

Nonetheless, Socket.IO seems very stable and, as I have said, is used in production code (TODO: crossref 1.1). The fact that development has slowed down, may just be a sign that the framework is near complete and very bug free (still, there are quite a lot of issues reported on github(TODO: source)).

However, I must say that I am a little concerned regarding the state of the project. As I said, development seemed to have reach a crawling pace with no new releases recently. The GitHub page states that version 1.0 is just around the corner (the documentation there is even updated to fit this version), but as this is being written, version 1.0 has been “upcoming” for nearly a year.

When it comes to Node.js in general, it is still not a proven platform for large scale applications. Nor is JavaScript itself for that matter. However, building larger applications client side with JavaScript has become more common, but it remains to be seen if the language can make its way onto the server.

Today, I don’t think anyone is willing to gamble on choosing Node.js as their primary server solution. However, if some real time functionality is needed, there is no problem using Socket.IO for this, even if the primary server is something else than Node.js. Then, if Node.js doesn’t catch on, one can replace it with something else without having to change the entire application.

Delivering real time is something I believe Socket.IO does well, and I must say that I am really excited to see how Node.js will pave the way for JavaScript on the server. If the road ends up full of pot holes or not, remains to be seen.

### Documentation

My first impression was that Socket.IO didn’t have a lot of documentation, but that what it had was enough. Mostly this is actually true. However, the problem occurs when dealing with the modularity of Node.js. The documentation should do a better job of pointing you in the right direction when it comes to functionality provided by other modules. Because of how it is now, I had to search around a lot and do some trial and error to achieve something as simple as serving the client with multiple files and not just a single “index.html” file as in every single example provided by Socket.IO’s documentation (TODO: codelist multiple files and socket.io example).

Node.js itself has a lot of documentation that is well authored. It is also mostly example based, which makes it easier to understand certain aspects. I actually didn’t think of the importance of Node.js’s documentation before I started working with Socket.IO. But just as it is common to need guidance about core functions of the .NET framework when working with C#, you cannot get by with a Node.js module without getting reference about core features of Node.js itself.

### Implementation of test application

As Socket.IO doesn’t offer any kind of client side template language, the test application was built using the common UI using Knockout (TODO: as described in.. + ref to git?). The application was build using Node.js version 0.10.8 and Socket.IO version 0.9.11.

To structure my application and harness the asynchronous control flow, I chose a technique that is very common in the asynchronous world, namely promises (TODO: codelist)[[12]](#footnote-12). The use of promises in JavaScript hasn’t been standardized, but most strive to follow one specific proposal (TODO: source). I chose a module for promises called promised-io by Kris Zyp (TODO: source), which closely resembles the client side implementation that JQuery provides (TODO: jquery deferred).

The application’s requirements specification (TODO: crossref), states that the application should utilize MySql as database engine. As with everything else in Node.js, I had several different modules to choose from. Felix Geisendörfer’s node-mysql was chosen because it had the most comprehensive documentation and it also seemed simple and intuitive to use. No problems arose during development regarding the use of this module, so I still believe that it was the right choice.

There aren’t many examples at Socket.IO’s homepage regarding how to provide the client with the needed files. That is not within the scope of Socket.IO’s functionality either, so it isn’t really a big drawback. Nonetheless, I still had to find out how to provide the client side of my application with several files, not just a single HTML file, which is the case in all the examples. Since Socket.IO seemingly recommends Express (TODO: move footnotes?), I chose to use it as well. Express is a web application framework, but for my use case, it was simply used to create the web server of my application (TODO: codelist).

When it comes to browser support, Socket.IO seems to hold what it promises (TODO: crossref). It works fluently in all browsers (TODO: crossref to methodology fallbacks), and it is all handled by Socket.IO behind the scenes.

### Testing

I was inexperienced in writing JavaScript code in general, but when it comes to testing in JavaScript, I was completely new. Still, it proved to be rather familiar once I got the gist of it. As with every other aspect of Node.js, there are a lot of choices when it comes to testing frameworks. My choice fell on the test runner Mohca (TODO: mocha) as it seemed like a good and stable choice[[13]](#footnote-13). For assertions, there was also a lot of choices, but I landed on Should.js (TODO: shouldjs or should.js + source) which was one linked from Mocha’s homepage..

The modularity of Node.js makes it easy to separate small, testable units for unit testing. With Mocha, writing tests for asynchronous code is also very simple. Actually, I was rather surprised with how easy it was to write JavaScript tests, as I have heard a lot of murmuring from different people in the business, that really don’t like writing tests for JavaScript.

Integration testing Socket.IO is actually very little painful. Many frameworks cannot be thoroughly integration tested without involving a browser. Socket.IO provides a client module for Node.js which can be used in test code. That means that you can establish a connection to the server from your test code (TODO: codelisting), without initiating a browser first, something that makes the tests a lot more effective and easy to write and understand.

### Summary

Socket.IO is a very solid framework that delivers what it promises (TODO: crossref intro quote). It has seamless fallbacks that enables it to function across all major browsers. Furthermore, it is simple to understand and use, and it is seemingly very stable. The documentation could be a little more comprehensive, and I have some small concerns regarding the pace new releases has come out over the past months. However, this is more likely caused by the fact that Socket.IO is near completion.

TODO: table? Good at, Bad at?

## Lightstreamer

Lightstreamer is a framework made in Java that provides a server specialized for real time applications. The server can communicate with a number of other server technologies via a clever adapter based architecture (TODO: figure). Due to this structuring, you can integrate Lightstreamer with virtually any existing system, as long as the language is supported by it. Currently, Lightstreamer provides adapters for Java, C# and JavaScript (Node.js). You can even integrate with other languages via the adapter “Remoting Infrastructure” (TODO: doc), though this is actually raw TCP sockets, so it is pretty low level code, and probably a little more work than integrating with a more traditional language adapter.

Client side, Lightstreamer supports numerous different platforms, including a JavaScript API for web applications, several desktop APIs in Java and C#, numerous mobile device APIs and even a generic client that is more low level (TODO: doc).

Behind Lightstreamer is an Italian software company named Weswit, which is actually one of the most experienced companies in the world when it comes to delivering software for making real time applications. Lightstreamer was released as early as 2000 (TODO: ppt about real time), and was one of the first attempts at delivering real time push in web applications without utilizing Java Applets (TODO: wikipedia).

Lightstreamer, unlike the other frameworks in this thesis, is a commercial product. However, they do provide a free license, which is the one I have utilized in my work.

### Why I chose it

First of all, Lightstreamer is completely different from all the other frameworks I have found. It is a commercial product from a rather large, European company that has a lot of customers worldwide. Furthermore, no other framework has been on the marked for as long as Lightstreamer, which gives me an unique opportunity to find out how much experience can influence the performance and usability of a real time framework.

As I mentioned, Weswit has a lot of customers worldwide, and if I have interpreted their homepage correctly, Lightstreamer is their only product (TODO: what we do). That means that Lightstreamer is used by a lot of companies in numerous solutions on the web, which is also indicated on the Lightstreamer homepage (TODO: homepage). It is even used by NASA to provide real time telemetry data for the International Space Station (TODO: article from NASA)!

Being a professional product and not “just a GitHub project” give rise to certain demands regarding the documentation of the framework. A well written and explanatory documentation is of course always preferable, but it is even more so for a product people actually pay for. With Lightstreamer, Weswit have obviously had this in mind–there are tons of documentation covering both the server side adapters and the client side APIs.

Another aspect you don’t necessarily get with a non-commercial product is support from the developers or forums. Lightstreamer hosts their own forum on their homepage[[14]](#footnote-14), in which representatives from Weswit frequently answers questions from users. There are also some activity on more general forums like Stack Overflow and Google Groups.

Finally, Lightstreamer, of course (TODO: as of section crossref methodology) supports every major browser. This means that it supports fallbacks to WebSockets, but I have not been able to find out exactly which. Their data sheet is the closest I have found mentioning Comet and HTTP streaming. As I wrote in section (TODO: crossref comet essay), HTTP streaming is covered by the Comet umbrella term, so I’m guessing that they mean some other form of HTTP based real time technology than streaming when they write Comet. A Norwegian blog post suggests that one of the fallbacks is long polling (TODO:allekonsulentene), which makes sense given the separation of HTTP streaming from Comet in Lightstreamer’s docs. Nonetheless, there is fallback support which is what really matters. How it works in practice remains to be seen.

But before I delve into the testing of the framework itself, I need to make some disclaimers. First, I have used the free license of Lightstreamer that does not come with all features included[[15]](#footnote-15). Furthermore, I used the Lightstreamer server itself as application server, which is not the recommended course of action when using Lightstreamer. I chose to do it this way, mostly to keep all the implementations of the test application as similar in architecture as possible. Furthermore, it is generally the case that you would separate the real time aspects of any application to its own server. In this thesis, the purpose is not to implement the whole architecture of a real time enabled web application–just the real time aspects are relevant.

### How it works

Lightstreamer incorporates a publish and subscribe model where clients subscribe to items. In this context and item is generally a collection of items (like a database table), for instance can an item be the products in an online auctionhouse.

Using an adapter based architecture makes Lightstreamer easy to incorporate with almost any system. To achieve this, an application using Lightstreamer has to have one metadata adapter and one or more data adapters (TODO: figure or crossref to previous figure).

The metadata adapter’s responsibilities include registering subscriptions to users, authenticating and filtering client communication to the appropriate data adapter. In my use case, the metadata adapter mostly routes messages to the data adapter(TODO: codelist).

While the metadata adapter handles incoming communication with users, the data adapter(s) keeps track of the individual items that users can subscribe to. When an item is updated, the data adapter is normally notified via an event listener that handles sending the updates to the clients (TODO: codelist).

On the client, Lightstreamer provides its own grid based template syntax in HTML (TODO: codelist). Furthermore, the framework ships with a JavaScript library for handling these grids and subscriptions. The template engine of Lightstreamer is made to handle received updates on the items the clients subscribes to–there are no mechanisms equivalent to those offered by for instance Knockout, which means that you have to do a lot of UI updates manually via for instance JQuery.

### Getting started

Obtaining a free license for Lightstreamer wasn’t particularly hard; just fill in a form and you are good to go. Downloading and installing too, was just as you would expect from a well functioning piece of software.

Learning about it and how to use it started out very gently, but it soon proved to be more difficult than what I had hoped. There is only one tutorial for a “Hello World” application that, while it is very informative, doesn’t really offer too much help when it comes to making something a bit more complex. To help developers move on from the “Hello World” stage, the framework comes bundled with quite a lot of demos. However, these are nearly not documented at all, which can be very frustrating as a lot of the code in the demos handle presentation of data rather than actually sending data back and forth. This was far from clear, and I must say that I spent a bit more time learning how to set up subscriptions (TODO: Cross ref) than I had hoped for.

Another problem is the way the demos are organized. A normal Lightstreamer application would have one metadata adapter and one or more data adapters. Therefore, I expected each demo app to have its own metadata- and data adapter, but when debugging, methods in these were never called. It turned out that all the demo applications share a common metadata adapter class in which all the separate meta data adapters are gathered. A small text on the server start page is the only place where the common meta data adapter is mentioned (TODO: figure), and it was first after I discovered this that the learning process accelerated.

Finally, I missed some indications as to how a database normally would fit into a Lightstreamer application. There is no mention on how to do this in the documentation. In fact, the only mention I found of this was a question in the forum that had been answered, somewhat unclear in my opinion, by a Weswit employee (TODO: source). This may be due to the fact that the Lightstreamer server is supposed to be decoupled from everything except the real time aspects. Therefore, I do not regard this as a drawback with Lightstreamer, but there should perhaps have been some suggestion in the documentation on how a database would fit into the picture.

### Coding environment

Giving developers multiple choices regarding programming language, both on the server and client, makes choice of environment very dependent on choice of technologies. Since Lightstreamer is written in Java, the Java adapters are more tightly connected to the framework itself. The .NET version for instance, requires you to build a small proxy server that registers the .NET adapter and handles communication with the Lightstreamer server. In the final paragraph of section (TODO: crossref why I chose it), I argued why I used the Lightstreamer server as an application server, even though this is not the normal use case. By using the .NET adapter, not only would it contradict my intentions, but it could also possibly introduce a bottleneck in my application, since it introduces two more TCP connections to the communication flow (TODO: figure). Therefore, the choice of adapter language fell on Java.

With Java as adapter language, you can choose from all the traditional Java IDEs. I used Eclipse for the server side logic[[16]](#footnote-16), and since the client is pure JavaScript, HTML and CSS, I decided to use WebStorm for this purpose.

Debugging a Lightstreamer application was a little tricky to begin with as there are no documentation on how to attach the debugger to the server. However, a forum post concerning this issue (TODO: forum post) helped me setting it up in a rather simple and straightforward manner. After that, debugging is just like you expect with any Java application.

However, there were some issues. First of all it seems like the source code is either written without any regard to normal naming conventions, or it has undergone some kind of minifying process before it was shipped. A lot of class- and variable-names show up as “t”, “arg0”, “arg1” etc., leaving me as a developer scratching my head from time to time–“What exactly is this parameter really?”. In my opinion, describing names on variables and classes are alpha omega when it comes to writing clean and understandable code.

Additionally, there seems to be some exception handling within the core of the framework that doesn’t get logged in any way. Several times, I would step through my code hunting for where an error had occurred, only to find out that it suddenly stopped working without showing any output at all. After this, the server sometimes shut down, while it sometimes didn’t. Further investigation normally revealed that I had passed a value that was null to somewhere it wasn’t supposed to, which in any other application would result in a red error stack trace in the output window (TODO: figure).

Running the server can be done in two different ways: You can use the provided “Start\_LS\_as\_Application.bat” file, or you can configure Eclipse to launch the server for you. The last option is the only way you can debug the application, so it was the obvious choice. The first option also requires you to manually put related compiled sources into the Lightstreamer folder structure (TODO: figure), which I found rather cumbersome to do while developing the test application.

There is no escape from this when it comes to the client side files though, since the Lightstreamer application server has no other way of serving the client with files that putting them manually into the file system. This problem would, however, go away if you use a separate application server. Therefore, I will not count this aspect towards the final evaluation of Lightstreamer.

### Code structuring

Using adapters provides for a very simple separation of concerns. Still, I find a bit odd that you can have several data adapters, but only one meta data adapter. While the meta data adapter handles a lot more than a data adapter, like authentication and such, it could have been useful with more than one of these in a larger application. However, this would require some kind out routing, as it is the meta data adapter that receives messages from the clients, so the way it is now, is really the simplest solution – you can still delegate responsibility to other classes that the meta data adapter administers.

Handling concurrency is always an issue with server applications. Socket.IO is fully asynchronous using events (TODO: crossref), while SignalR’s concurrency issues is handled behind the scenes by the .NET framework (TODO crossref and ref github?). With Lightstreamer you are more in charge of handling concurrency, but you are still urged to use a asynchronous approach using an executor service and event listeners to control the flow in which code gets executed(TODO: codelist – large?). This may be a good thing, as it gives you more freedom in configuring and prioritizing how messages are handled, but it can also be a bit confusing. In my opinion, there should be some default behavior bundled with the framework that you could chose to either hook into or completely override as you see fit. Then you would at least ensure that you can get a basic, well functioning concurrency handling within the application.

Another aspect that has to be handled manually in Lightstreamer is subscriptions. The other “pure” real time frameworks (and also Meteor(TODO: crossref))[[17]](#footnote-17) abstracts this away from the developer, something I find to be very useful as long as there are good mechanisms in place that allows for some form of message routing. With Lightstreamer, you have to keep track of subscriptions via an unique object (a handle), that each subscription receives. If the subscription is “general”, in other words something that all users subscribe to, you relate the handle with a listener for that item (TODO: codelist). If you have subscriptions that are user specific, you have to keep track of the subscriptions in a map structure using some unique key to store the handle objects (TODO: codelist).

Client side the code feels somewhat forced. First of all, you must utilize RequireJS[[18]](#footnote-18) to get access to the Lightstreamer client API. Though RequireJS is a very good framework for structuring JavaScript code, it is not the only way, and for some projects it might be a little to comprehensive. In my case, I mixed it with the structure I already had in the common UI, sacrificing a consistent codebase instead of rewriting the whole thing.

As mentioned in section (TODO: how it works), Lightstreamer provides a grid based template syntax for its HTML. For pure adding, removing and updating this works very well, as it allows the server to send a prepared DOM-element that automatically gets added to the grid (TODO: codelist), but if you need more it becomes insufficient rather fast. You are only allowed to data-bind DIV- or SPAN-elements, which effectively restricts any possibility of setting visual traits automatically like you would with Knockout. In order to solve this, I wrote quite a lot of JQuery code that felt rather unnecessary since it did the same as what Knockout handles behind the scenes (TODO: codelist one of the util functions as example).

You can use Knockout if you like, but then you sort of need to “hack” the client side of your application. By using a hidden field and a subscription to a single item, you could send JSON messages back and forth and then process them on the client (TODO: codelist).

However, I believe that Lightstreamer’s client is meant to use the template syntax it provides because Lightstreamer is meant to be used for more push based applications. One clear indication of this is the fact that the server has no method, other than the update of an item, to send a message to the client. Simple messaging from server to client is common in more bi directional frameworks like for instance Socket.IO. But, if you keep in mind that Lightstreamer is supposed to coexist with a separate application server, the messaging from server to client could easily be handled by this server instead of Lightstreamer. This would then be through a HTTP response, though, and not via WebSockets.

### Serialization

Lightstreamer serializes data that is tightly coupled with the DOM. Server side you send a map containing the fields that are represented in the DOM to a listener. When this arrives at the client, a new DOM-node has been created. When it comes to sending data from the client however, you have to handle serialization your self – both when sending and on the server upon reception. This isn’t necessarily a bad thing, but it means that, as with other aspects of Lightstreamer (TODO: crossref), you have to write some more logic.

### Maturity

Released in 2000, Lightstreamer is, by far, the oldest framework in this thesis. In many ways, you can tell that Lightstreamer is somewhat aging. It is based on the publish/subscribe model, which is quite different from the full bi directional nature of other real time frameworks today. Still, when it comes to pure push, I believe that Lightstreamer is a very stable and mature alternative that delivers high quality to its users.

Nonetheless, there are some flaws. It may actually have become too mature over the years. Compared to for instance Socket.IO and SignalR, Lightstreamer is a huge framework that offers a lot more than just real time technologies with authentication, throttling of bandwidth and other aspects (TODO: datasheet) that normally is handled by an application server. If your application server already does all of this, these extra features of Lightstreamer are redundant.

New versions come out with a little over 2 years in between. The version I used, 5.1.1 came out March 2013 (TODO: downloads page), which means that there are still some time to the 6.0 release. To get up to speed with today’s trends, a complete restructuring of the framework might be a good idea. Switching to a more module based framework, they could let developers have more choice in exactly what they want. This would also allow for a better foundation for a more bi directional communication focus rather than the publish/subscribe model they provide today.

But why would they do such a restructuring? Weswit has a large customer base and only good reviews. WebSocekts is a new technology though, and with it comes a new way of solving real time. And while Lightstreamer uses WebSockets, I do believe they would benefit from offering something in addition to the publish/subscribe solution.

### Documentation

The documentation is massive–actually it felt a little overwhelming. Still, the API documentation is separated in several documents, one for each API. Something that is almost non-existent though, are examples. With a lot of text and no examples, you get an overview of the concepts, but not how to actually implement them.

This is where Lightstreamer refers you to the demos that comes with the framework. Most of these are rather simple applications, but with a lot of unnecessary code regarding UI formatting. In addition, the code is poorly commented, which is quite strange given the abundant amount of comments in the various configuration files you can use to tweak the Lightstreamer server.

Another rather strange thing with the documentation is that it is very comprehensive, but it does not offer a lot of explanation regarding one of the central concepts of Lightstreamer, namely the different subscription modes. I still don’t feel that I have a good understanding of what MERGE, DISTINCT and COMMAND mode really means. The best explanation I found on some of them was actually not in the documentation, but in the forums (TODO: forum post).

### Implementation of test application

Lightstreamer allows you to have multiple data adapters for handling several different items. In my case, there is really just one item; the products in the auction house. In addition comes the ability to register users, log in and view a users current bids. All of these additional features are not something that I believe Lightstreamer would normally handle – they would probably be handled by the application server beforehand.

To circumvent this, I made three different items a client can subscribe to: items (products), logged in users (to keep track over users who are logged in), and bids (per user). The item subscription is initiated the first time the page loads, while the others are initiated on demand.

As described in section (TODO: crossref), Lightstreamer is probably not meant to interact with a database directly. Consequently, there was little help offered, both in the documentation and in the forum, on how to integrate a database directly with Lightstreamer. Hence, solving this was a matter of “best guess”, implementing database logic in a logical place within my application logic (TODO: make some kind of figure).

To do the actual interaction, Java applications rely on a MySql connector (TODO: mysql connector). This connector introduces some logic that quickly can become a little messy. In an effort to prevent this, I created a database utility package (TODO: link to git) that I used in the Lightstreamer application, which, in turn, made the database logic very short and readable (TODO: codelist).

### Testing

TODO: write about after refactoring tests!

### Summary

Lightstreamer is a solid framework when it comes to publish/subscribe applications. No other framework in this thesis has been available nearly as long as Lightstreamer, something that shows regarding bi directional communication. However, with the solid platform the framework already has, it has a very good potential when it comes to a restructuring to a module based solution. While Lightstreamer has a large customer base, I believe that it will be unable to compete with the more compact solutions offered by Socket.IO and SignalR in terms of real time applications if they do not do a restructuring similar to what I have described. The reason for this is that, even though Lightstreamer does its job very well, it offers a lot more than what you might generally need in a real time application.

## Play Framework

Play Framework (or just Play) is a *MVC* (Model View Controller) web application framework written in Scala and Java. The project was started by a software developer from the company, Zenexity in 2007, but it was two years later, when it went open source, that it really became popular (TODO: philosophy).

The framework utilizes Akka (TODO: akka) to provide an asynchronous programming model for handling concurrency. Akka is a *“toolkit and runtime for building highly concurrent, distributed, and fault tolerant event-driven applications on the JVM” –* (TODO: akka front page). It is also built using Scala.

Developing a Play application lets you use a lot of different tools and language dialects on both server and client. Server side, you can choose between Java and Scala, whilst on the client, you have a lot of choices. Play uses a template engine to make views, and it comes bundled with one using Scala. However you can easily use any JavaScript based template engine like Knockout, although this requires some use of the Scala template engine as well (TODO: crossref implement or code structure). Furthermore, Play comes bundled with some asset compilers that allow you to use LESS instead of CSS (TODO: source) and CofeeScript instead of JavaScript (TODO: source), right out of the box.

### Why I chose it

As you might have realized, Play is not a real time framework, so why is it in this thesis? One of the questions I seek to answer is whether or not you need a framework at all to implement scalable real time applications across multiple platforms (browsers) (TODO: crossref problem statement). While Play doesn’t leave you on completely bare ground (TODO: crossref how it works) when it comes to implementing real time, it is quite a lot of steps closer to a manual implementation than what is provided by the other frameworks in this thesis.

Play is one of the few web application frameworks I have seen that promotes an asynchronous application model and real time features. Hence, it stands out from the crowd in this matter. Moreover, it is an increasingly popular framework that is used in production code by some serious actors, like for instance LinkedIn[[19]](#footnote-19).

The asynchronous model that Play offers, means that it should be perfect for implementing real time features. In addition, the framework gives you some helper classes for WebSockets (TODO: ws api doc) and HTTP-streaming (Comet)(TODO: comet api doc) via the forever frame technique (TODO: crossref essay chapter). An interesting byproduct of having these helpers when making real time features, is that it may offer quite a lot of insight in how it would be without the help: If it’s really cumbersome in Play, it is most definitely hard without the help that Play offers. On the other hand, if it’s not too difficult with Play, it means that you can probably write your own helpers by following the Play code (it is open source).

### How it works

In the previous section, I mentioned that Play offers some helper classes for WebSockets and Comet. This does not limit you to use just these means of implementing real time functionality, since Play is, a web application framework with asynchronous capabilities. However, since I used only WebSockets and Comet, I will focus solemnly on those.

Exposing a WebSockets connection with Play is nearly as simple as writing a normal action[[20]](#footnote-20) in a controller. The aforementioned helper class is generic and lets you operate with either JSON or simple strings as message exchange format. To me, the obvious choice was JSON since the client is JavaScript. Initiation a connection is done by simply returning an instance of the helper class (TODO: codelist), rather than returning a result as you would normally do (TODO: or codelist with both?). The handshake is then handled by Play. After the handshake is done, you can use an event based server side architecture to handle incoming messages and disconnections (TODO: codelist).

Using Comet is a little more work, which was what I expected since this requires you to maintain one open connection for pushing data as well as handling incoming POST requests. Nevertheless, it is quite similar to implementing WebSockets, only that the Comet object only provides an outgoing channel. The incoming channel is just normal POST requests. This means that two actions are required in the controller (TODO: codelist). The Comet class also, just like the one for WebSockets, provide an event for handling disconnections.

As mentioned in section (TODO: play intro), Play uses Akka to provide an asynchronous way of handling concurrency. This is done by using constructs known as “actors” (TODO: wiki). An actor can receive messages, make necessary computations (and even make new actors), and then reply with a proper response. Each actor then, sort of, has its own event loop, forcing all code within the actor itself to be executed with mutual exclusion (TODO: codelist). This all may seem a little complicated, but it is actually quite easy to learn, especially through the excellent documentation of the Akka framework (TODO: source).

### Getting started

Getting started with Play was a dream. Almost every aspect of using the framework, from installing to advanced usage, is well documented with accompanying examples. Additionally, there are some samples and tutorials, covering both simple cases as well as more in-depth studies.

However, Play builds on, and uses, a lot of other libraries and frameworks. Although Play does a good job at referencing those external parts’ individual documentations, it still means that the framework itself gets a quite steep learning curve.

While you are mostly helped gently towards the top of that curve, there are some small, but quite unnecessary issues. Some package names in import statements in the tutorials. There were also some links that returned 404 errors in the more advanced tutorial (TODO: zentask + figure?), and in the end, there was some issues that had to be corrected to make it compile.

Relying on quite a lot of different third party libraries and framework, Play can become somewhat vulnerable when some of this third party software becomes outdated. Luckily, Play has a active and alert community that catches such things quickly and solves them. I had some issues when following a tutorial on writing integration tests, which I got help to solve. It turned out that the issue was due to an outdated version of the Selenium Webdriver.

### Coding environment

The framework comes bundled with its own console application. This is used to run applications or tests and debugging as well as making new projects. It’s a very functional tool that you don’t really have to worry too much about while developing, since Play uses auto-reload when the application runs in development mode. Due to this feature, your code is recompiled and reloaded in the browser each time you save changes to your project.

Play allows you to use whatever Java or Scala IDE you want (TODO: IDE), and it is even described how to set up projects using the most common IDEs in the documentation. Furthermore, if you don’t like to use an IDE, you can simply use a text editor like Emacs, SublimeText or any other. With the application running from the console, you still get auto-reloading while developing with a text editor.

Debugging a Play application requires you to start the application in debug mode. Then you can attach the debugger from your IDE of choice. While this is a little different than what the other frameworks in this thesis has required, it is actually this method that is the norm. An usual development process has you running the server as a separate process (not started from the IDE). When you do this, you have to attach a debugger to that process in order to debug. For my simple test cases though, it has not been necessary to set up servers in this way, but with Play, this is done automatically through the console, and deploying new changes are handled behind the scenes.

Even though debugging is simple enough most of the time, there were some issues. First of all, while Play comes bundled with Ebean, it does not understand this if you run tests for your models via the IDE–you get an exception. To fix it, you actually have to get Ebean manually (TODO: blogpost) and then reference it through a VM argument in the IDEs run configuration (TODO: figure).

Secondly, debugging the asynchronous nature of Akka is a bit “narrow minded”. When debugging code that uses the “await” feature (TODO: codelist), you either have to be very fast, or increase the timeout time beforehand. Neither are preferred ways, but there are no other work around. In my case, this wasn’t a big issue, but with a full scale web application built on Play, using actors, it will definitely cause problems. Java obviously has a lot to learn in the terms of asynchronous programming.

### Code structuring

Developers who has used ASP.NET MVC will feel very much at home with Plays MVC layout. You may write as many controllers, models or views as you like to keep concerns separated and your code structured–just as one expects from a MVC framework.

There are some slight differences though, especially when it comes to organizing routes. In .NET, routing is, by default, dependent on the names of your controller classes and action names (TODO: MVC blog thing). Play uses a static file, and while you get compilation errors in the browser (TODO: figure), it is not strongly typed in the IDE[[21]](#footnote-21). In a small application, maintaining the routes file isn’t a cause for problems, but in a large application with many routes, I can imagine that it quickly gets out of hand and becomes hard to refactor at a later time.

However, there are probably other means of organizing routes that comes with experience in using Play. When it comes to the models and views part of MVC in Play, you stand very free. Ebean is shipped with Play and used in every example, but you may choose not to use it and write your models however you see fit. The same goes for views: Scala is the default template language, but you may choose not to use it. For .NET developers though, it should feel very familiar as it resembles the Razor view engine that ASP.NET MVC views use (TODO: codelist comparison).

When it comes to concurrency handling, you are, once again, free to use whatever you prefer. Akka, which is most common, gives you a simple, hierarchical way of handling concurrency while maintaining a clean separation of concerns. As I used Play, I learnt more and more about how to use Akka, and looking back at the application structure I ended up with, I see that I probably should have used several actors to handle messaging. Still, one actor does the job, but the code can easily become a little messy with a lot of instance of checks (TODO: codelist).

Any framework needs a simple way of handling dependencies. Maven users are probably accustomed to using a .pom file to list dependencies. With Express for Node.js you have a JSON file (TODO: figure that shows both or two separate..?). Play gives you the Build.scala file, which allows you to list dependencies in a code file in a way that closely resembles what developers already know.

### Serialization

As I described in section (TODO: crossref how it works), Play lets you operate with JSON as exchange format. The serialization (TODO: codelist) and deserialization (TODO: codelist) though, you have to handle manually. Luckily, the JSON helper class that Play offers by default is the Jackson JsonNode[[22]](#footnote-22), which makes this quite simple. Furthermore, some core features of Play is integrated with Jackson, allowing you to for instance get request bodies as JSON using a simple utility method (TODO: codelist).

### Maturity

### Documentation

### Implementation of test application

### Testing

### Summary

## SignalR

### Why I chose it

### How it works

### Getting started

### Coding environment

### Code structuring

### Serialization

### Maturity

### Documentation

### Implementation of test application

### Testing

### Summary

## Meteor

### Why I chose it

### How it works

### Getting started

### Coding environment

### Code structuring

### Serialization

### Maturity

### Documentation

### Implementation of test application

### Testing

### Summary

## Conclusion

1. 0.7 preview was released May 5th 2011 [↑](#footnote-ref-1)
2. <https://trello.com/> [↑](#footnote-ref-2)
3. [www.stackoverflow.com](http://www.stackoverflow.com) [↑](#footnote-ref-3)
4. The fallback happens ”behind the scenes” so that developers do not need to worry about it. [↑](#footnote-ref-4)
5. The command ”npm install socket.io” installes socket.io and all dependencies right into your project.TODO: codelist instead of footnote? [↑](#footnote-ref-5)
6. If the server code is C# for instance, it is common to use Visual Studio also for the client side JavaScript code. [↑](#footnote-ref-6)
7. <http://www.sublimetext.com/> [↑](#footnote-ref-7)
8. [www.jetbrains.com](http://www.jetbrains.com) [↑](#footnote-ref-8)
9. This means that, by default, JavaScript code is non-blocking. You can, however, write synchronous code. [↑](#footnote-ref-9)
10. One might for instance put all database logic in one module. If this is a lot of logic, one can split the “main” database module into several, smaller modules, each with a more narrow responsibility. [↑](#footnote-ref-10)
11. When serializing a Date object to JSON, you actually just get the string representation of the object (toString). To deserialize this, one has to initialize a new Date object instance (TODO: js date node soruce). [↑](#footnote-ref-11)
12. Also known as deferred or futures. [↑](#footnote-ref-12)
13. Jasmine (<http://pivotal.github.io/jasmine/>) was another testing framework I considered, but it doesn’t seem to provide as good asynchronous testing as Mocha. [↑](#footnote-ref-13)
14. <http://forums.lightstreamer.com/forum.php> [↑](#footnote-ref-14)
15. See full list of features here: <http://www.lightstreamer.com/products> [↑](#footnote-ref-15)
16. With server side logic, I mean the adapters. [↑](#footnote-ref-16)
17. A ”pure” real time framework is a framework whose main task is to provide real time messaging. Socket.IO, Lightstreamer and SignalR are examples of such frameworks. [↑](#footnote-ref-17)
18. <http://requirejs.org/> [↑](#footnote-ref-18)
19. See Play’s homepage at the bottom: <http://www.playframework.com/> [↑](#footnote-ref-19)
20. Controller-methods in a MVC framework are referred to as actions. [↑](#footnote-ref-20)
21. Meaning you don’t get IntelliSense while writing new routes. [↑](#footnote-ref-21)
22. The version used can be downloaded or viewed here: <http://jackson.codehaus.org/> [↑](#footnote-ref-22)