# 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).

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 Stackoverflow[[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?

### 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?

## Lightstreamer

## Play Framework

## SignalR

## Meteor

## 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)