Socket-based game

Network Programming with Socket.io

Philip Ekblom – phiekb@kth.se

6 January 2017

Contents

1. Introduction

This project is based on a library called Socket.io. Socket.io is a JavaScript framework that's ported over to many different platforms. It can therefore be used to communicate through sockets between a lot of different devices and programming languages.

The idea for this project came from a game concept I was sketching on. The idea was that events in the game could be triggered from another platform, such as a website. I then realized that this is possible with sockets and started digging more into it.

The system currently contains a NodeJS server, Java clients, Web clients (using AngularJS) and Unreal Engine game clients. All clients communicate with the NodeJS server through Socket.io.

Socket.io is event-based. This means that each "message" you send is sent as an event. Each request has the following structure [Event, JSON Object]. You'll see more examples of this later.

The current state of the system is a basic prototype of the game. The system is right now focused around network features such as a chat and simple events.

I am planning on continuing the project after this rapport, this means that the files on the Github will change later and therefore not link correctly to some parts in this rapport.

The game itself is supposed to be a hacking game, where the players needs to find different clues and vulnerabilities hidden in different systems. These systems are both random generated devices in the game but also real programs such as the java client. I also plan on having an IT team you can play as that is going to try securing these devices. Basically, Hackers vs IT. If you're interested more in the actual game you can check some of my ideas in my concept document.

I'll exclude a lot of details about the actual game in this rapport since the game itself is quite irrelevant to this course. I will however still explain all the network features related to the game.

All code can be found at my Github:
https://github.com/phek/clue_seeker

2. Literature Study

The main goal was to find a way to implements sockets in the game engine I am using, Unreal Engine, I figured it would be quite hard to implement this myself since you must think about a lot of things related to the engine. After some research I found Socket.io which is a JavaScript framework for sockets. This framework is however also ported to Unreal Engines own system called Blueprints. Another great thing about it is that it supports JWT authentication.

Since Socket.io is a JavaScript framework from scratch I chose to use the NodeJS framework for the server, both because it was better tested and documented than the ported versions, but also because NodeJS works very well with AngularJS and MongoDB which I planned on using for the frontend and database. The reason why I think NodeJS works very well with AngularJS and MongoDB is because they're all using JavaScript. Developers then only need to know one language, no matter if they work with the database, frontend, backend or all of them. They also support test-driven development and package management through NPM.

One way to implement sessions in AngularJS is through JWT. This is why I was so happy about the JWT support in Socket.io. I had however never used JWT nor created authentication in AngularJS before and therefore had to make some research in this too.

I've always thought it is really hard to implement JWT but turns out that you can achieve it quite easily. There's a NPM package called Jsonwebtoken that encodes a payload into a token for you. You can also decode the tokens with this package. Another option is to use Auth0 that will handle all this for you, this service cost money though but could be an option if you're running a company and don't want to care about authentication. I do not mind this though and therefore chose the Jsonwebtoken package.

Another thing to consider was that I haven't worked that much in Unreal Engine and I am very new to their Blueprint system. Unreal Engine do support C++ too though, therefore I made some research about if I should use Blueprints or C++ for my game.

Blueprints was used by most tutorials and documentation I could find. The Blueprint version of Socket.io was also very well documented and had good reviews, however, there was a ported version for C++ too, but I wasn't sure how well it would work with Unreal Engine.

C++ have a lot better performance than their Blueprint system, but it seems like they've improved their Blueprint system a lot over the years. When looking through some performance tests between the two it seemed like C++ were about 4 times faster than Blueprints, this could vary a lot though and you could convert Blueprints to C++ code to lower the gap later when packaging the game.

C++ clearly seemed to be a lot more complex in every single way, but it is a language I am familiar with and it does have a lot better performance, therefore I tested to create a small dummy project in C++ and tried to use the C++ Socket.io version with it, I very soon noticed that things became way too complex and how hard it was to implement the sockets through C++ in the engine.

I therefore changed to Blueprints since the performance difference shouldn't be a problem in my game. It's well documented and there's a lot of guides for it. I also plan to bring in some Game Designers later,

Blueprints is a graph-based programming language and was mainly created to make it easier for designers to program different features and shaders without knowing programming.

3. Method

Three editors were used during the project, Netbeans for the Java-client, Unreal Engine for the game client and Webstorm for the NodeJS server and AngularJS web-client. The programming languages used is Java, Unreals blueprint system and JavaScript.

The first task for the project was to make sure everything was compatible with each other. I therefore made very basic clients for each of the platforms I planned to use. I then followed Socket.io's documentation to create a connection from each of the clients to the server.

When each of the clients could connect to the server I started creating the authentication system since I had to make sure JWT worked for both AngularJS and Socket.io. I did this by creating an RESTful API on the NodeJS server that was called each time the AngularJS client wanted to login. Instead of using a database I had a simple IF-statement to test a single user. If the correct info was entered I generated and returned a JWT token, otherwise an error was thrown.

When the authentication worked in Angular I had to figure out a way to authenticate users through Socket.io too, this was solved by using a middleware. You could then include either a token or login information with your connection request before the server accepted the socket connection. If the data was valid the middleware accepted the connection, otherwise it blocked it. I could then call the same API on the server for the socket authentication with some minor changes.

When the authentication worked I created a client-handler on the server to keep track of all the active clients. I thought it would be handy to have more information about the socket connections, I therefore also implemented that the clients had to send what platform they were using when authenticating to the socket. I then stored all this data in a list of connected sockets, because of this I then could lookup socket connections by username and platform. So for example if I would like to send a message to all Java-client this would be possible.

I then created some basic game functionality for my game and made it possible to play online. I first wanted to use the NodeJS server for syncing all the game data, but this didn't seem to be possible or simply just too complex to achieve. I therefore handle all gameplay syncing through the game clients instead since Unreal have premade functionality for this. One of the clients then act as a host of the game and pushes out all the data to the other game clients. I could then instead use events from the NodeJS socket connection to call events in the game.

I also noticed it was quite easy to implement Steam support, I therefore also made it possible to connect online through Steam.

At last I started experimenting with different events and created a chat feature across the platforms.

4. Result

Preview

The system itself is quite big and complex, so let's start with some pictures to make it easier to understand everything.

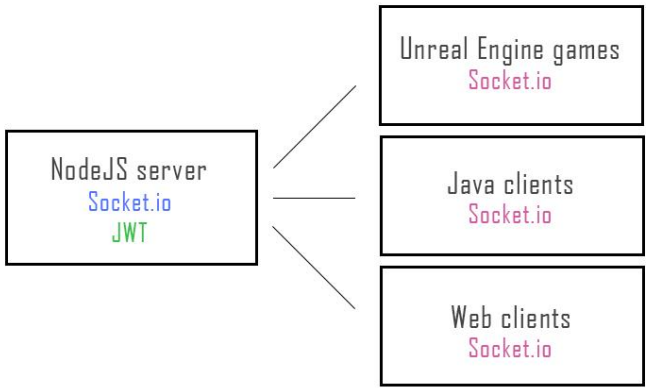

Figure 1. The different platforms in the system.

Each of the clients communicate with the server through sockets. The server can emit events to all the clients or an individual client. A typical scenario is that client 1 sends an event to the server, the server then emits a new event to all clients.

Let's show an example of the chat system:

Figure 2. The view of the web-client with 4 active users (self-included).

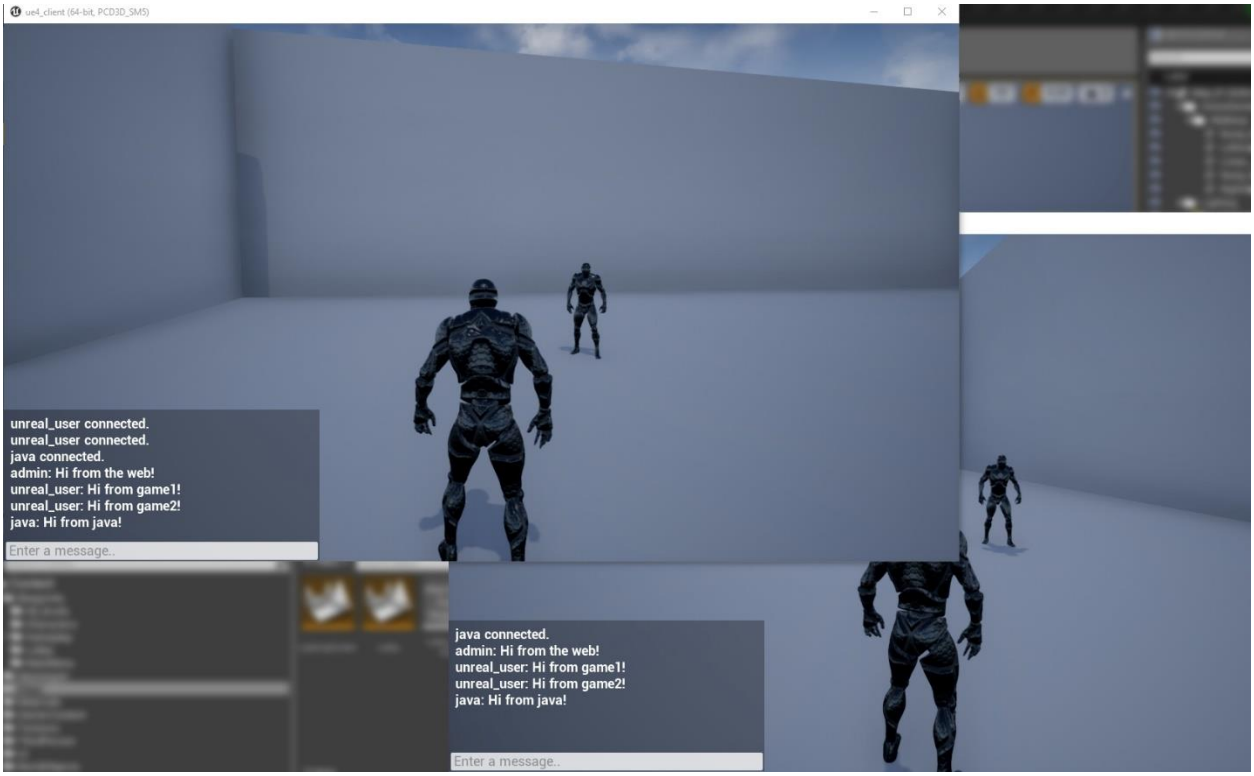

Figure 3. The view of two Unreal Engine game instances. Same session as in figure 2.

```
Login by typing: /login your_username your_password
/login java test
Successfully connected to http://localhost:3000
java connected.
admin: Hi from the web!
unreal_user: Hi from game1!
unreal_user: Hi from game2!
Hi from java!
java: Hi from java!
```

Figure 4. The view of the java-client. Same session as figure 2.

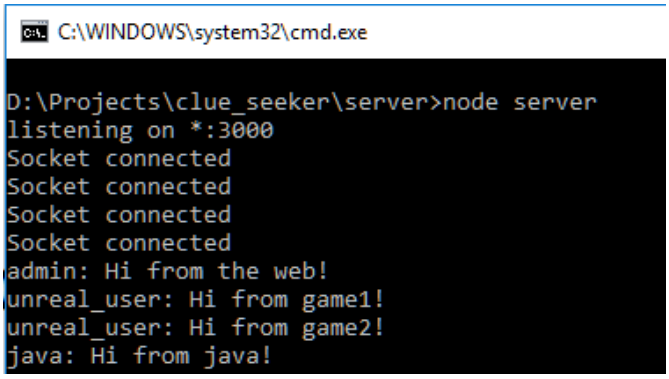

Figure 5. The view from the server console. Same session as figure 2.

As you can see in figure 2-5 the server emits an event to all other clients every time a socket-connection is established with the connected clients' username. The clients will then take that username and print out "#User has connected". The same thing happens when a client sends a new message, just that there is a different event and data containing the message instead.

I also experimented a bit with creating different events. Let's have a look at that:

Video example: https://i.gyazo.com/39f15f5df7dd6d35f4ddfacb77fccd9d.mp4

This example was created quite early in the development, as you can see there's no chat and a prototype game. This however still works on the Java-client and Web-client but is not yet implemented in the new Unreal Engine game.

Whenever the web-client presses the "Alert Users"-button all connected users will be notified and a box in the game will start burning.

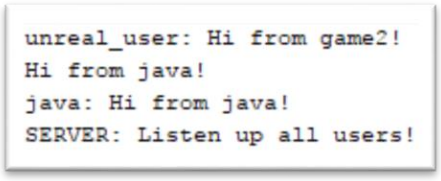

Figure 6. The view of the java-client when a user in the web-client presses the "Alert Users"-button. Same session as in figure 2.

Let's take a look on how the server handles the socket-connections:

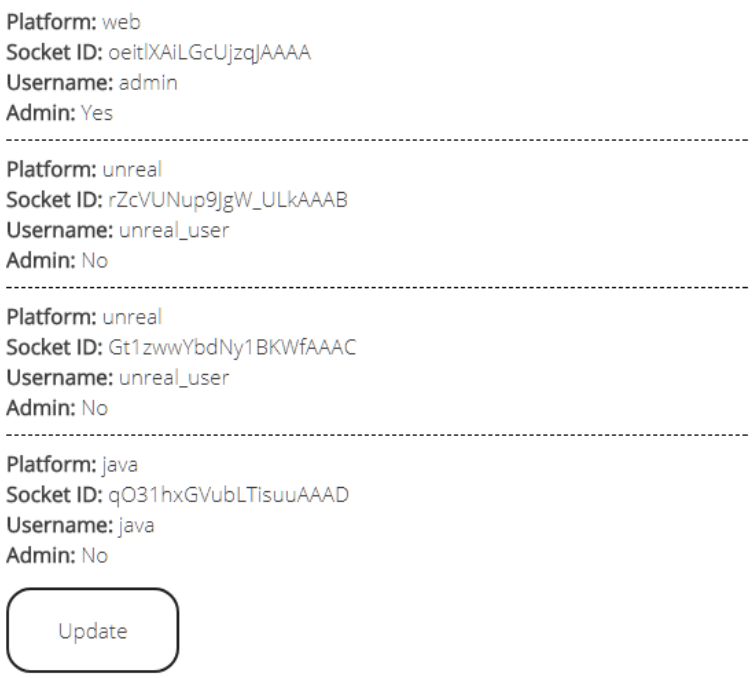

Figure 7. View of the web-client when listing all active socket-connections.

The server stores information about each connection before it's established in the middleware during the authentication process. Admins can then request this information from the sockets page. The way this works is that the client calls the RESTful API with its token attached, the API then controls the token and sends a response with all active sockets if the token is valid. If the token is invalid the server will respond with a 403 (Forbidden).

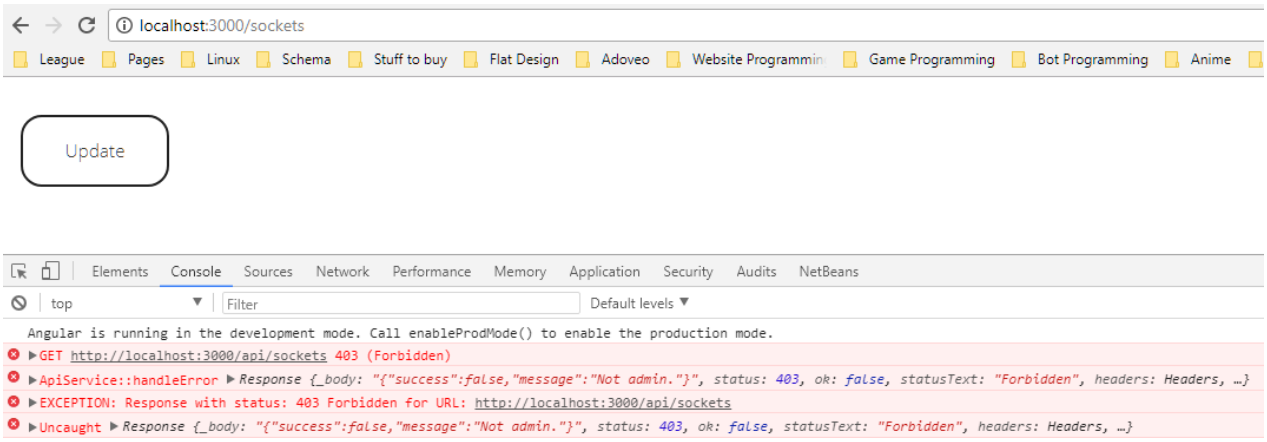

Figure 8. Example of an unauthorized user trying to access the socket information.

JWT

Let's dive down into the more specific technologies now when you got a basic understanding of how the system works.

To make a JWT token you take a payload and encode it with a secret key. The only one that knows the secret key is the server. That is why you can authenticate users through a token, since the server can decode the token with this secret key to check if it's valid.

I chose to use Username, Platform and IsAdmin as my payload since this is information I would like to access frequently.

I also chose to exclude the file containing the database and secret key information from Git. Otherwise anyone could create fake tokens if they find the key in the Git-project.

The code for creating a token can be found in the ApiHandler and the code for validating the token can be found in the AuthHandler.

At the moment I do not use a database, instead you can login with whatever user you want, if the login info is "admin – admin" then a token with admin privileges will be created, otherwise a normal token will be created. I have created and set up a MongoDB database, I did however choose to not use it for this course to keep it simpler.

Sockets

Let's explain a bit more in detail how the communication in Socket.io works.

Figure 9. Simplification of how a client is setting up a socket connection to the server with login information.

Figure 10. Simplification of how a client is setting up a socket connection to the server with token.

In figure 9 the client tries to connect to the server using the event "connect" and a query object containing their username and password, the server then controls this data and generated a token if the information is correct. It then accepts the socket connection and sends back the token through an event called "token" and an object containing the token. The client can then choose to store the token for future use.

In figure 10 the client is connecting with a token instead.

In both cases the middleware will block the connection if the data provided is invalid.

If you wish to see the middleware in action you can see this code in the SocketHandler at line 8. You can also see all the events that will be handled on the server at line 39. At line 43 you can see how the information about active socket is store on the server.

The ClientHandler is the object that stores all the active sockets.

Structure

I created Handlers to divide the different functionalities on the NodeJS server. As you can see in the Sockets chapter a handler is created for each task. One for handling API calls, one for handling Clients, one for Sockets and one for Authentication.

In the web-client I used AngularJS default file-structure. Angular2 uses TypeScript which makes JavaScript more object orientated. This makes it possible to create components for each page on the website.

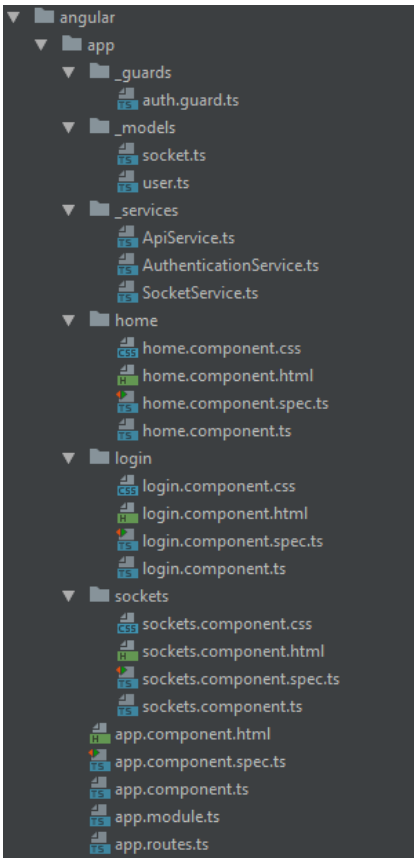

Figure 11. The structure of the AngularJS application.

In the game I created clear folders for each part of the game.

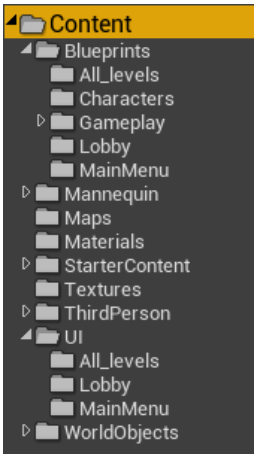

Figure 12. The structure of the Unreal Engine game.

And at last in the Java client I used a MVC-architecture without a controller. The reason I am not using a controller at the moment is because the program is too small to have good use of the controller, this will however be implemented later when I develop the Java client more.