Doing a Project With Socket.io and React.js

Installation

Make an empty folder then right click on that folder in Cloud9, ‘open terminal here.’

```
ec2-user:~/environment/gladiator/chaser/sim-3-logic-unit-1-6/socket-learner
(integrate-histo-with-rec) $ npm init -y
Wrote to
/home/ec2-user/environment/gladiator/chaser/sim-3-logic-unit-1-6/socket-learner/packag
e.json:

{
  "name": "socket-learner",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}


ec2-user:~/environment/gladiator/chaser/sim-3-logic-unit-1-6/socket-learner
(integrate-histo-with-rec) $ ls
package.json
ec2-user:~/environment/gladiator/chaser/sim-3-logic-unit-1-6/socket-learner
(integrate-histo-with-rec) $ npm i axios express socket.io
npm notice created a lockfile as package-lock.json. You should commit this file.
npm WARN ws@7.2.1 requires a peer of bufferutil@^4.0.1 but none is installed. You must
install peer dependencies yourself.
npm WARN ws@7.2.1 requires a peer of utf-8-validate@^5.0.2 but none is installed. You
must install peer dependencies yourself.
npm WARN socket-learner@1.0.0 No description
npm WARN socket-learner@1.0.0 No repository field.

+ socket.io@2.3.0
+ axios@0.19.2
+ express@4.17.1
```

```
added 102 packages from 62 contributors and audited 206 packages in 4.202s
found 0 vulnerabilities
```

Server Formation, Bind Server to socket.io, index.js setup

ec2-user:~/environment/gladiator/chaser/sim-3-logic-unit-1-6/socket-learner (integrate-histo-with-rec) $ **clear**
ec2-user:~/environment/gladiator/chaser/sim-3-logic-unit-1-6/socket-learner (integrate-histo-with-rec) $ **pwd**
/home/ec2-user/environment/gladiator/chaser/sim-3-logic-unit-1-6/socket-learner
ec2-user:~/environment/gladiator/chaser/sim-3-logic-unit-1-6/socket-learner (integrate-histo-with-rec) $ **touch app.js**
ec2-user:~/environment/gladiator/chaser/sim-3-logic-unit-1-6/socket-learner (integrate-histo-with-rec) $ **touch index.js**

Then move the index.js into a new folder (which you must create) called **routes**.

Setting client connection duration, as a parameter within app.js

Here is the server prototype we'll begin with. Open app.js in your Cloud9 editor and enter this code for **index.js**

```
const express = require("express");
const router = express.Router();

router.get("/", (req, res) => {
    res.send({ response: "cal poly express service is present"
}).status(200);
});

module.exports = router;
```

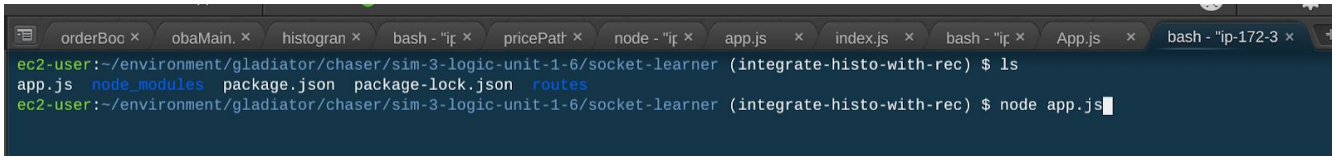
Then for the working server code, app.js, it will deliver a second by second update of the clock time, on the server, to the client.

App.js:

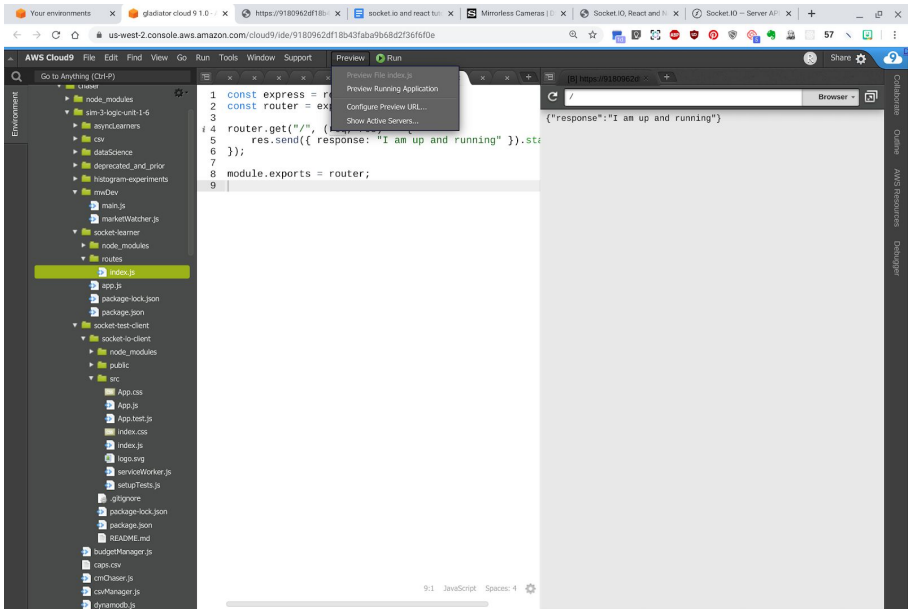
```
//app.js
//initiate an express instance, which listens for incoming
connections via socket.io protocol
const express = require("express"); //double quotes implies it comes
from the npm repo
const http = require("http");
```

```
const socketIo = require("socket.io");
const axios = require("axios");

const port = process.env.PORT || 4001;
const index = require("./routes/index"); //server listens on this
route for incoming, by default

const app = express();

app.use(index);

const server = http.createServer(app);

const io = socketIo(server);
//-- step 2, lifecycle

let interval;
io.on("connection", socket => {
    console.log("new client connected");
    if (interval) {
        clearInterval(interval);
    }
    interval = setInterval(() => getApiAndEmit(socket), 1000);
    socket.on("disconnect", () => {
        console.log("client disconnected");
    });
});

//---- configure listening, for incoming connections
server.listen(port, () => console.log(`Listening on port ${port}`));

const getApiAndEmit = async socket => {
    try {
        var d = new Date();
        let msg = "time now is " + d;
        socket.emit("FromAPI", msg); // Emitting a new message. It
will be consumed by the client
    } catch (error) {
        console.error(`Error: ${error.code}`);
    }
};
```

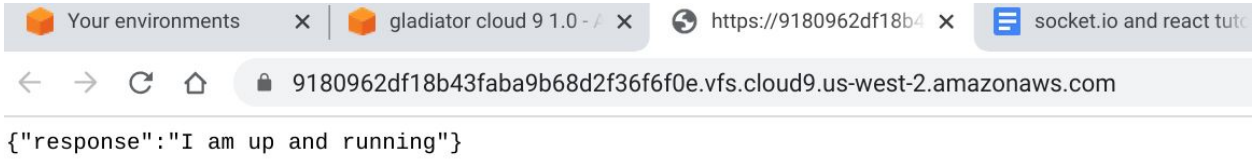
Be sure you are in the same directory as the server, app.js, then run it, using **`node app.js`**



Now that's running, confirm it by selecting index.js, then select Preview button --

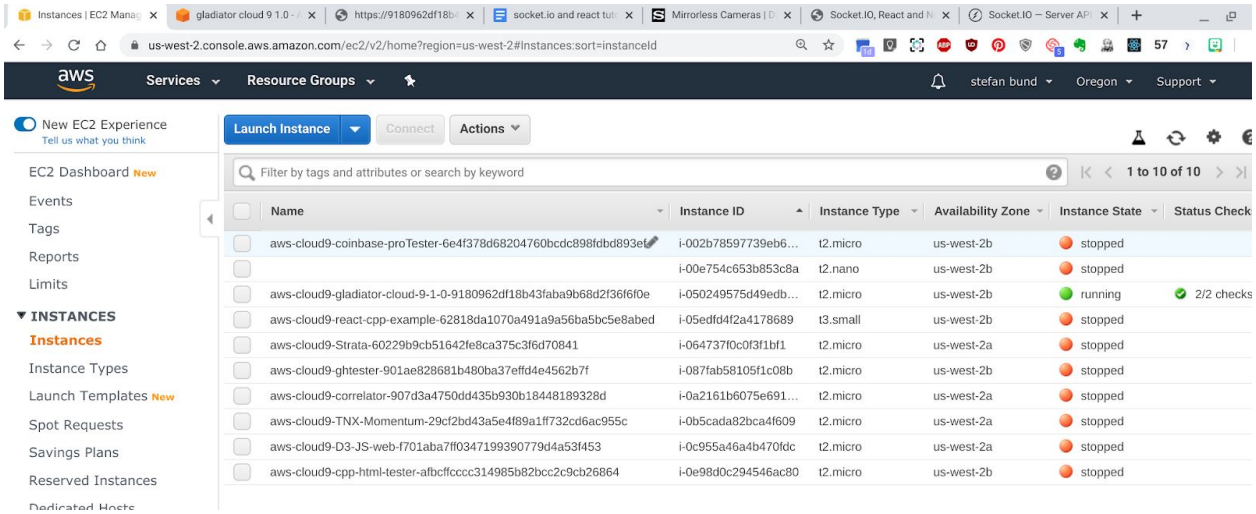


The browser will confirm the response clients get when they connect to the active, running service. You can also detach the Preview browser to get it running in a separate tab of Chrome.

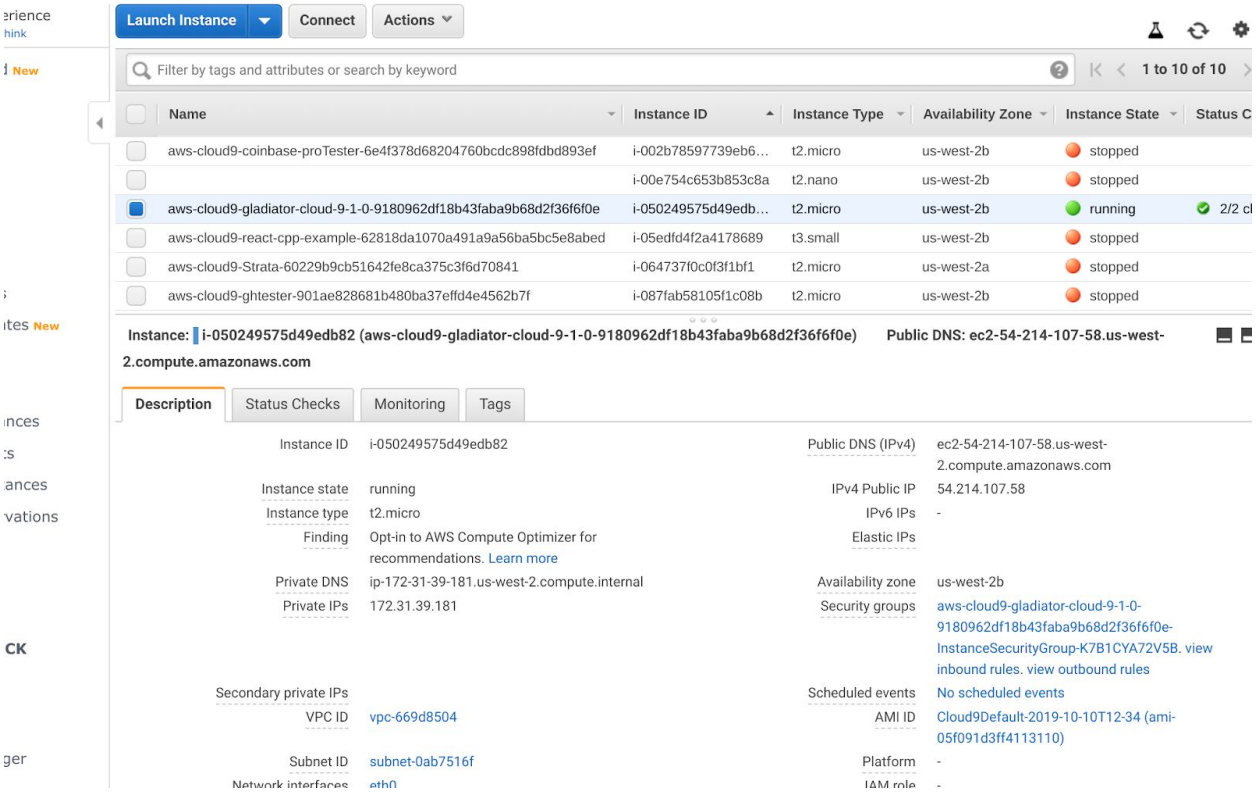


Investigating EC2 parameters for the service

Cloud 9 created a new linux instance for your service, and you can investigate its temporary IP and DNS settings by visiting EC2 in your AWS console.



When you click on the green one, it displays the current running details, below. We are looking specifically for the current IP/DNS settings and for the Security Group settings, which govern which incoming and outbound connections will work.



When you click the highlighted URL for the **security groups**, you visit the configuration for **inbound** and **outbound** connections, which I will show you how to configure, below.

Create Security Group | Actions

Group ID : sg-01df769fdf1aeb4eb | Add filter

Name	Group ID	Group Name	VPC ID	Owner
	sg-01df769fdf1aeb4eb	aws-cloud9-gladiator-cloud-9…	vpc-669d8504	165021830876

Security Group: sg-01df769fdf1aeb4eb

Description | Inbound | **Outbound** | Tags

Edit

Type	Protocol	Port Range	Destination	Descriptio
Custom TCP Rule	TCP	8080	0.0.0.0/0	
Custom TCP Rule	TCP	8080	::/0	
Custom TCP Rule	TCP	4001	0.0.0.0/0	
Custom TCP Rule	TCP	4001	::/0	
Custom TCP Rule	TCP	8082	0.0.0.0/0	
Custom TCP Rule	TCP	8082	::/0	

Create Security Group | Actions

Group ID : sg-01df769fdf1aeb4eb | Add filter | 1 to 1 of 1

Name	Group ID	Group Name	VPC ID	Owner	Description
	sg-01df769fdf1aeb4eb	aws-cloud9-gladiator-cloud-9…	vpc-669d8504	165021830876	Security group

Security Group: sg-01df769fdf1aeb4eb

Description | **Inbound** | Outbound | Tags

Edit

Type	Protocol	Port Range	Source	Description
HTTP	TCP	80	0.0.0.0/0	
Custom TCP Rule	TCP	8080	0.0.0.0/0	
Custom TCP Rule	TCP	8080	::/0	
SSH	TCP	22	34.218.119.32/27	
SSH	TCP	22	34.217.141.224/27	
Custom TCP Rule	TCP	4001	0.0.0.0/0	
Custom TCP Rule	TCP	4001	::/0	
Custom TCP Rule	TCP	8082	0.0.0.0/0	
Custom TCP Rule	TCP	8082	::/0	

Notice that we are trying to open the ports which the application will utilize; for example, the server will use **8080**, and the client, which we will build momentarily, will try to use 8080, but will have to take the next available ports (8081 or 8082), in order to start. You will see how the client will ask for permission to initialize on a port alternative to 8080, so once you are aware of that port id, you should enter it into the security group settings.

Introducing a react.js client, as a consumer for socket.io data

Exit the folder you created for the server, and create a new folder on the same Cloud9 environment. Type

```
ec2-user:~/environment/gladiator/chaser/sim-3-logic-unit-1-6/socket-test-client
(integrate-histo-with-rec) $ npx create-react-app socket-io-client
```

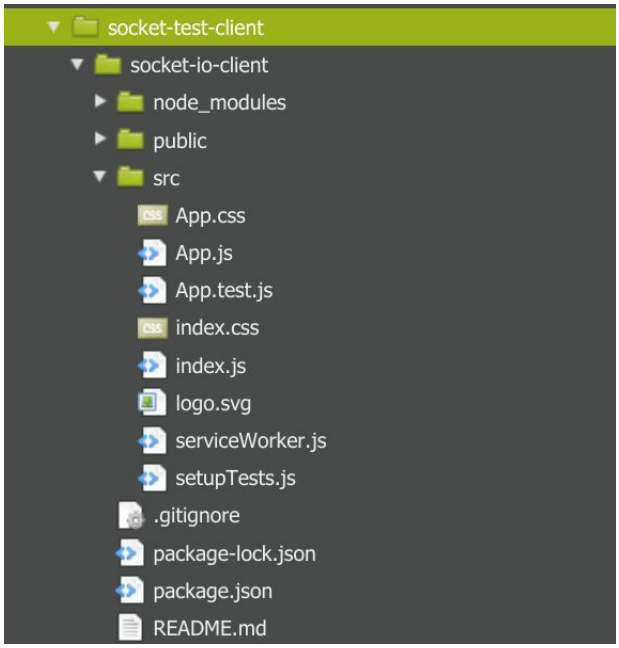
Then

```
ec2-user:~/environment/gladiator/chaser/sim-3-logic-unit-1-6/socket-test-client
(integrate-histo-with-rec) $ npm i socket.io-client
```

Navigate into the new client src directory to craft an app.js

```
ec2-user:~/environment/gladiator/chaser/sim-3-logic-unit-1-6/socket-test-client
(integrate-histo-with-rec) $ ls
socket-io-client
ec2-user:~/environment/gladiator/chaser/sim-3-logic-unit-1-6/socket-test-client
(integrate-histo-with-rec) $ cd socket*
ec2-user:~/environment/gladiator/chaser/sim-3-logic-unit-1-6/socket-test-client/socket
-io-client (integrate-histo-with-rec) $ ls
node_modules  package.json  package-lock.json  public  README.md  src
ec2-user:~/environment/gladiator/chaser/sim-3-logic-unit-1-6/socket-test-client/socket
-io-client (integrate-histo-with-rec) $ cd src
ec2-user:~/environment/gladiator/chaser/sim-3-logic-unit-1-6/socket-test-client/socket
-io-client/src (integrate-histo-with-rec) $ ls
App.css  App.js  App.test.js  index.css  index.js  logo.svg  serviceWorker.js
setupTests.js
```

New folders are now active, and available for the project and the app



Src contains the code we'll use to craft the react.js app

Customize your App.js on the socket client

First, open src/App.js and delete its default content:

```
import React from 'react';
import logo from './logo.svg';
import './App.css';

function App() {
  return (
    <div className="App">
      <header className="App-header">
        <img src={logo} className="App-logo" alt="logo" />
        <p>
          Edit <code>src/App.js</code> and save to reload.
        </p>
        <a
          className="App-link"
          href="https://reactjs.org"
          target="_blank"
          rel="noopener noreferrer"
        >
          Learn React
        </a>
      </header>
    </div>
  );
}

export default App;
```

After you’ve deleted the default content, add the following demonstration code, which will poll the socket.io server for time, every second. Note for the ‘endpoint’ parameter below, you must grab the current DNS name for the EC2 instance from the EC2 console, as above, then insert into App.js on the client. Remember to not confuse App.js in the react folder with app.js in the socket folder.

App.js, on the client:

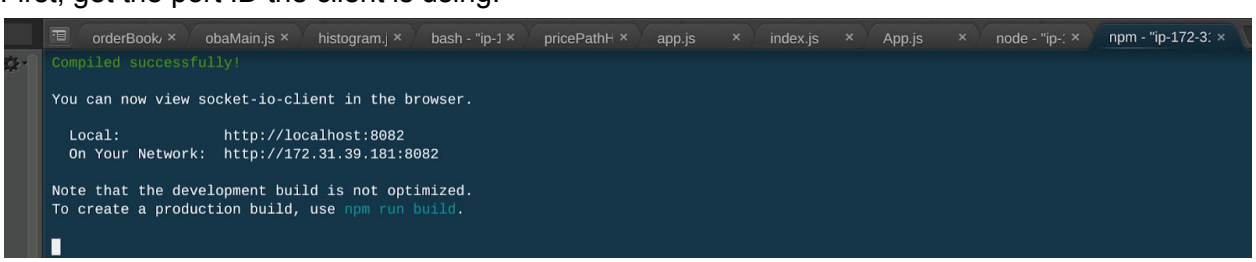
```
import React, { Component } from "react";
import socketIOClient from "socket.io-client";

class App extends Component {
  constructor() {
    super();
    this.state = {
      response: false,
      endpoint:
"http://ec2-54-214-107-58.us-west-2.compute.amazonaws.com:8080"
    };
  }

  componentDidMount() {
```

```
    const { endpoint } = this.state;
    const socket = socketIOClient(endpoint);
    socket.on("FromAPI", data => this.setState({ response: data }));
  }

  render() {
    const { response } = this.state;
    return (
      <div style={{ textAlign: "center" }}>
          {response
              ? <p>
                Cal poly <br></br>

                {response} <br></br>

                (currently online)
              </p>
              : <p>waiting for CPP...</p>}
        </div>
    );
  }
}

export default App;
```

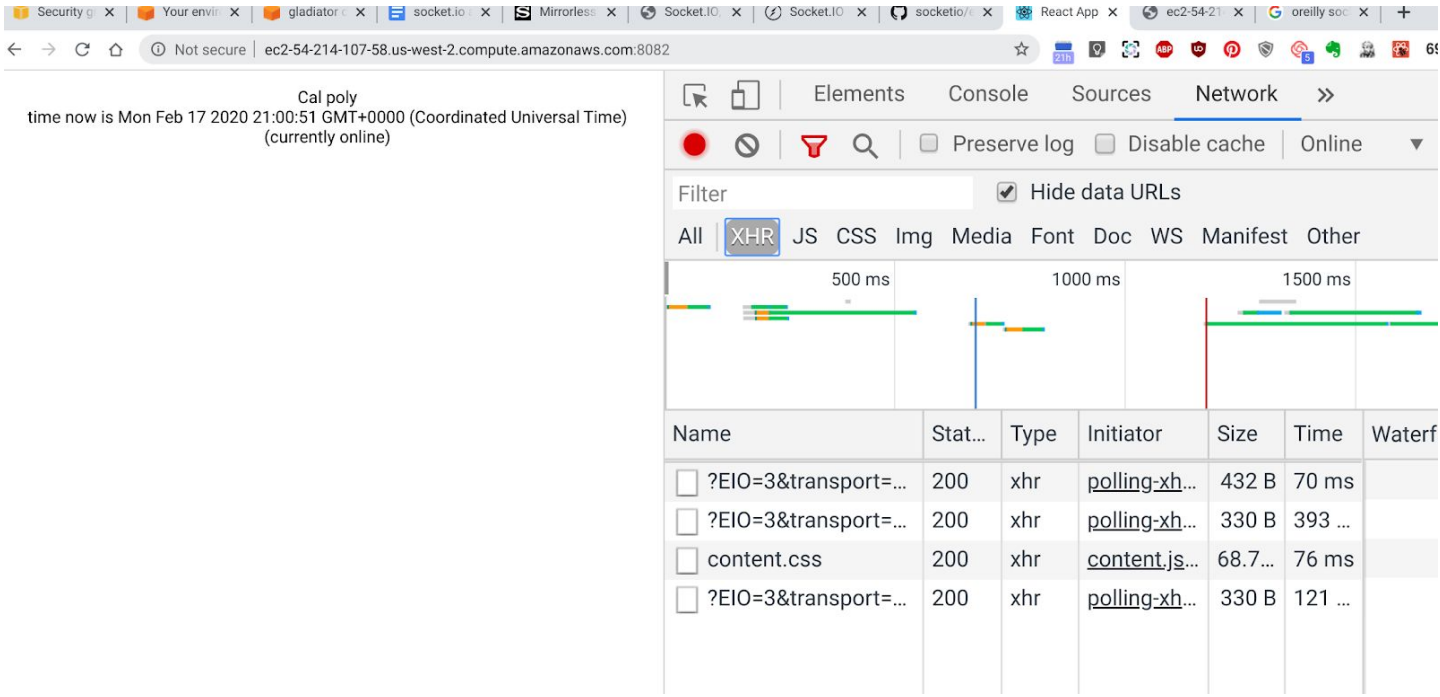
Testing in your Chrome Browser

From here, you are plugging in the client's browser address, since you are attaching to the client, which, in turn, soaks the local time from the server. Hence, plug in the DNS name and protocol for the client, as present in your console output, which tells you which port ID the client is using. Combine that port ID with the DNS name from the EC2 console.

First, get the port ID the client is using:



Then harness the DNS from EC2, to load the proper web page in your browser:



The rubric is DNS name + active react service IP, or

[AWS DNS name] : [active port ID, in React.js]

This may take some practice, but having the theory is the most helpful strategy.

Keep the page open, and watch it update the time, second by second -- remember, the React.js client is grabbing down the current time on the page with each polled interaction with Express/Socket.