EcoPRT Vehicle Network Solution Developer's Guide

Installation Guide

To install our application, please follow the installation guide provided.

Technologies Used

The following are the technologies we used for the project and their versions. There is also a description of what they are used for.

Technology	Version	Description
NodeJS	8.9.4	Web Server software. This will run on the web server and will serve the website and connect the website to the database. It will contain a REST API. The REST API will allow the website and other services to access the functions of the web application.
MongoDB	3.6.3	Database is written using MongoDB. The web server will connect to it. This database will hold information about vehicles, edges, rides, users, nodes, and user authentication.
Bootstrap	4.0.0	Using Bootstrap will give more freedom to customize the look and feel of the web application. It also makes it easier for to make a mobile friendly version of the website.
AngularJS	1.6.8	Used for the logic of the website. It works very well for data binding which is the bulk of the web application since all data is pulled from the database.
Express	4.16.2	Routes URLs to files and/or functions
HTML5	5	Used to create web pages
REST	n/a	Used to get information from the server and send it to the front end
Socket.io	2.0.4	Used to get information from the server and send it to the front end

CasperJS	1.1.0-beta4	Used for automated testing of web applications with JavaScript
Mocha	5.0.5	JavaScript framework for Node.js that allows for Asynchronous testing
Chai-HTTP	4.0.0	Assertion library that runs on top of Mocha
Istanbul (nyc)	11.7.1	Used for determining code coverage

Suggested Text Editors/IDEs

The following are text editors or IDEs we used when developing our code and recommend.

Name	Version	Reason/Description
Visual Studio Code	1.22.2	It has source control so you can use Git (ctrl-shift-p to clone a repo). It has a lot of plugins for web development. Gives suggestions when coding for html, css, and js. The files explorer menu is well placed and easy to read and follow. It has a built in terminal and console for you to use that as well.
Coda (Mac Only, Requires you pay for it)	2.6.1	Has good support for web files, including html, css, and js. Built in color coding and suggestions for these files. Also has ftp and git support.

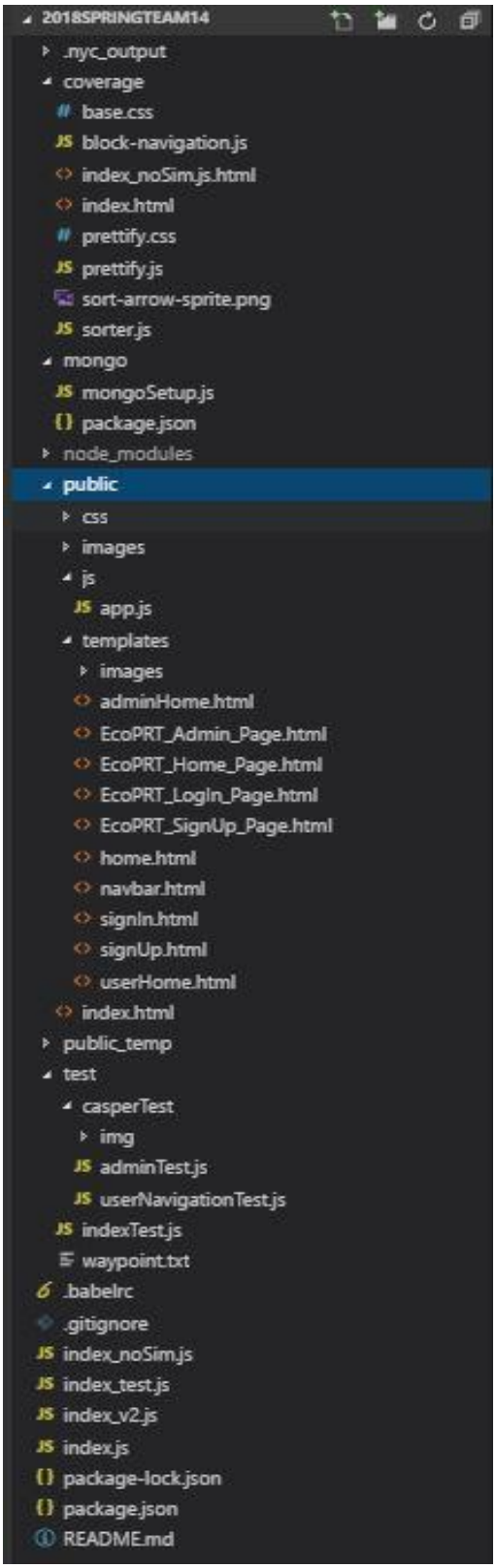
File Structure

The root of the folder contains the server files. */index_v2.js* is the main runnable server file. */package.json* contains the information about dependencies.

The *mongo* folder contains *mongoSetup.js* which creates test data for the database. This test data will help for testing the system and running the simulation in *index_v2.js.*

The *public_temp* folder contains basic HTML forms that uses the REST API for adding and viewing the different items in the database. This was used early in our development to test the REST APIs. This is not used in the final application in anyway.

The *public* folder contains all files that will be served to the user in their web browser. Users will not be able to access files outside of the *public* folder. In the root of the *public* folder there is an */public/index.html* that imports any files like *js/css*. The user never leaves *index_v2.js*. It has a section which always includes a snippet of html that corresponds to the current URL. The page use AngularJS routes to display the different snippets. The file *app.js* has the AngularJS front end logic.



Testing File Structure

The root folder contains the server file /*index_noSim.js* and two folders called *coverage* and *test*. The server file */index_noSim.js* contains a copy of the server code from */index_v2.js* with the simulation code removed. This is the server file that the Mocha tests test.

The folder *test* contains a folder called */casperTest* and two files called */indexTest.js* and */waypoints.txt*. */indexTest.js* is the file that contains all the Mocha tests and */waypoints.txt* is an example waypoints file that is used by the Mocha tests. The folder */casperTest* contains two files and another folder. The files */adminTest.js* and */userNavigationTest.js* contain the CasperJS tests and the folder /*img* contains screenshots that are taken when the CasperJS tests are run.

The folder *coverage* in the root folder contains files that are automatically generated when the Mocha tests are run with the coverage tool Istanbul. The file */index.html* inside the *coverage* folder contains all coverage data that is obtained when running the Mocha tests with the coverage tool Istanbul and can be viewed by opening the html file in a web browser.

Database Information

There is a database schema document provided. It has all the different collections we used for the MongoDB database. It has all the field names, format type and a short description of the field.

REST API Information

There is a provided REST API document that has all the different REST APIs for Admins and Riders.

Extending REST API

/index_v2.js contains all the server functions. To add a new REST API endpoint, copy and paste one of the current ones.

There are functions for converting the authentications tokens to users. Use findUser() for user endpoints and findAdmin() for admin specific endpoints. findUser() will also return admins.

When you are about to send data back to the user:

- Use functions resSuccess() and resError() to create uniform replies.
- resSuccess() automatically applies a 200 status to the reply.
- resError() automatically applies a 500 status to the reply, but can be changed via parameter.

Example:

```
app.post('/addvehicle', function (req, res) {
    var form = req.body;
    if(form && form.name) {
        findAdmin(req.get("authToken"), function(admin) {
            if(admin) {
                addVehicle(form, res)
            } else {
                resError(res, "Account Not Found", 400);
            }
        })
    } else {
        resError(res, "All fields must be filled out", 400)
    }
})
```

- app is express
- .post is which REST call you want to use. Currently only get and post are being used, however any REST call can be put there

- /addvehicle is the url to use
- function(req, res) {} is the callback, the contents of this function will be called when the URL is called. req stands for request (parameters that were sent with the call), res stands for response (used to send things back to the user).
- If you include a JSON form in the request when sending it, req.body will contain the form.
- This function should only be used by admins, so we call findAdmin() with the authToken passed in by the request. function(admin) will be called afterwards and the variable admin will contain either the accounts object of the admin calling the URL, or will be false if the authToken is not a valid admin authToken. Use findUser() instead of findAdmin for URLs that are only for users.
- addVehicle() has all the actual logic for the request.
- resError() sends the result back to the user. res is passed in as the first parameter, the error is the second parameter, and the status code is the third parameter.
- resSuccess() sends the result back to the user with statusCode 200. res is passed in as the first parameter, the data to return is passed as the second parameter.

Extending Web Sockets

Since data, especially vehicle data, can be sent very often from the server to the client, websockets are used to cut down on the amount of data sent.

The library socket.io (https://socket.io/) is used for websockets. If an admin wants to get all updates on something, they subscribe to a corresponding room, and when the server updates something, it sends will send the updated data to every client in the room.

Example:

```
io.on('connection', function(socket){

  socket.on('joinAllRidesInfo', function(authToken) {
      findAdmin(authToken, function(admin){
          if(admin) {
              socket.join('allRidesInfo');
              getAllRidesInfo(function(rides){
                  socket.emit('allRidesInfo',rides);
              })
          }
      })
  })
```

- Io is the socket.io library
- .on('connection') is used when someone connects via websocket
- function(socket){} gets called when the connection happens. The socket variable corresponds to that specific connection. So, if you you use the socket variable to send information, it will only go to one user.
- socket.on() tells the server what to do if the client send specific information. In this example, if the client sends 'joinAllRidesInfo', it will do the corresponding function.
- Since this function is an admin function, we have the client pass in their authentication token and verify it with the findAdmin function, similar to the REST API.

- socket.join() adds this specific client to a room. In this case, the client is added to the 'allRidesInfo' room.
- In this example, when a client connects, they will also want the current Rides info. socket.emit() sends a message to a specific client. In this case we send the message 'allRidesInfo' with the Rides data as a parameter

```
io.to('allRidesInfo').emit('allRidesInfo',vehiclesInfo);
```

- io.to() sends a command to every client in a room. This sends a message to every client in the 'allRidesInfo' room.

Adding Database Functions

```
var url = 'mongodb://localhost:27017/ecoprt';
var MongoClient = require('mongodb').MongoClient;

var connectToDB = function(callback) {
    MongoClient.connect(url, function(err, db) {
      if(err) {
        console.log(err);
        callback(false);
      }
      else callback(db);
    });
}

var insertDocument = function(collectionToUse, docToInsert, callback) {
    connectToDB(function(db){
        var collection = db.db("ecoprt").collection(collectionToUse);
        collection.insert(docToInsert, function(err, result) {
            db.close();
            if(err) {
                console.log(err);
                callback(false);
            } else {
                callback(result);
            }
        })
    })
}
```

The bottom of the *index_v2.js* file contains database functions.

- url is a mongodb connection string
- MongoClient is the import used for connecting to the database.
- connectToDB() does the initial connection to mongo. This is a helper function used in all other database functions insertDocument() is an example database function.
- connectToDB() is called initially to connect to the database.
- The collection variable lets mongo know which collection we will be querying from.
- collection.insert() is a mongoDB function. Any command you can use in a mongoDB shell can be used here.
- function(err, result) is called once a response is received from mongoDB. An error (or null if no error) will be given as the first argument, and the second argument is the queried document(s).
- db.close() closes the current connection to the database

Adding a New Webpage

/public/templates/ has html snippets of all the pages. Add new html web pages here.

To define the urls, we use angular routes. */public/js/app.js* contains the code for the routes in the app.config function. Each angular route also has a corresponding controller that contains the logic for that page. All of the controllers are at the bottom of the page. After adding the .html webpage to */public/templates*, add another .when case in the */public/js/app.js* app.config. Then create a new */public/js/app.js* app.controller with the controller name specified in the new .when case.

The header has its own controller in *app.js*. The header has multiple html snippets that correspond to different pages. To add a new header, add another snippet in the header html with a new conditional corresponding to the page it will be on. Then in the header controller in *app.js*, set the variable when the url is loaded.

Adding New Test Cases

Testing Installation and How to Run

CasperJS

- Go to the official CasperJS website at casperjs.org
- Follow the installation instructions on that page
- Use the command "*casperjs test test/casperTest/*js*" to run the casper tests.

Mocha and Istanbul

- All dependencies for Mocha testing is included in package.json.
- Run *"sudo npm install"* to install dependencies.
- Use the command *"sudo npm test"* to run the Mocha tests with coverage.
- Coverage can be viewed in the terminal after all tests are run and at */coverage/index.html*

CasperJS Tests

The folder */casperTes*t located at */test/casperTest* contains all files for CasperJS testing. All CasperJS test files are structured similarly. They begin with an object called config containing the url of the website that needs testing. The next parts of the code begins the CasperJS Tests by preparing the testing environment and loading the webpage that is to be tested. The following sections of code contain the individual steps that occur (such as individual steps in Black Box tests). Each step begins with "*casper.then*". In the example below, we load the webpage and click on the Log in link on the top navbar. We then proceed to type in the log in credentials for the admin account.

```
1  var config = {
2      url: 'http://sd-vm02.csc.ncsu.edu/#!/',
3  };
4
5  casper.test.begin('Testing admin', function suite(test) {
6      localStorage.clear();
7      test.comment('loading ' + config.url + '...');
8
9      casper.start(config.url, function() {
10         this.waitForResource(config.url, function() {
11             casper.capture('test/casperTest/img/1.png');
12             test.assertTitle("ECO PRT", "ecoPRT title is the same");
13             test.assertUrlMatch(config.url, 'you should be on the home page');
14         });
15     });
16
17     casper.then(function() {
18         this.waitForResource(config.url, function(){
19             test.comment('loading...');
20             test.comment('clicking on Log In...');
21             this.clickLabel('Sign Out', 'a');
22             this.clickLabel('Log In', 'a');
23             test.assertTitle("ECO PRT", "ecoPRT title is the same");
24             test.assertUrlMatch(config.url + 'ecoPRTLogin', 'you should be on the Log in page');
25             casper.capture('test/casperTest/img/2.png');
26         });
27
28     });
29
30
31     casper.then(function() {
32         casper.capture('test/casperTest/img/3.png');
33             this.wait(5000, function(){
34                 casper.capture('test/casperTest/img/4.png');
35                 this.fillXPath('form#login-form', {
36                     '//input[@id="email"]': 'admin@example.com',
37                     '//input[@id="password"]': 'admin'
38                 }, true);
39                 test.assertUrlMatch(config.url + 'ecoPRTLogin', 'you should be on the Log in page');
40                 this.clickLabel('Sign In', 'button');
41                 casper.capture('test/casperTest/img/5.png');
42             });
43     });
44
45
46     casper.then(function() {
47         this.wait(1000, function(){
48             casper.capture('test/casperTest/img/admin.png');
49             test.assertUrlMatch(config.url + 'adminHome', 'you should be on the adminHome page');
50             casper.capture('test/casperTest/img/7.png');
51         });
52
53     });
54
```

- Lines 1 - 3 creates the config object
- Lines 6 - 15 prepares the testing environment and loads the webpage
- Lines 17 - 28 the Login link is clicked
- Lines 31 - 43 the admin credentials are filled into the login form
- Lines 46 - 53 checks if the user is on the admin home page

For more details on CasperJS, visit the official documentation located at docs.casperjs.org/en/latest

Mocha Tests

The Mocha tests are all located at */test/indexTest.js.* The first line of the file is *"require('../index_noSim')"*. This starts the server code that will be tested. If the server code that needs to be tested is a different file, this line needs to be replaced with the file of the code. In order to add test cases, copy and paste the section of code that begins with *"it("testname",function(done){ test code});"*. Below is an example and a brief break down of what each section of the code does.

```
34          it("testing signup form bad email",function(done){
35              chai.request('http://localhost:80')
36              .post('/signUp')
37              .type('form')
38              .send({
39                  username : 'testname',
40                  email : 'admin@example.com',
41                  firstName : 'Test',
42                  lastName : 'Name',
43                  birthday : '06/09/1969',
44                  password : 'password'
45              })
46              .end(function(err, res) {
47                  expect(res).to.have.status(400);
48                  done();
49              });
50          });
```

- Line 34 begins the individual test with the name of the test in the quotes.
- Line 35 lets the test know what the server to test is.
- Line 36 specifies which API call is tested
- Line 37 - 45 specifies the type of request is and what data to send if any
- Lines 46 - 49 verifies whether the correct response was received.

Testing the admin functionality requires the user to be authenticated as an admin user. The following is an example of how to test a REST API call that requires authentication.

```
135
136         it("testing add vehicle",function(done){
137             chai.request('http://localhost:80')
138             .post('/signIn')
139             .type('form')
140             .send({
141                 email : 'admin@example.com',
142                 password : 'admin'
143             })
144             .end(function(err, res) {
145                 var token = res.text
146                 var vehiclenum = (Math.random() * 1000)
147                 chai.request('http://localhost:80')
148                 .post('/addvehicle')
149                 .type('form')
150                 .set('authToken', token)
151
152                 .send({
153                     name : 'vehicle' + vehiclenum
154                 })
155                 .end(function(err, res) {
156                     vehicleID = JSON.parse(res.text).id
157                     expect(res).to.have.status(200);
158                     done();
159                 });
160             });
161         });
```

- Line 137 - 143 logs in the admin user
- Line 145 creates a variable called token and sets it to the authentication token
- Line 150 sets the authentication token

For more details please visit the official Chai-HTTP website at www.chaijs.com/plugins/chai-http/