Term Project Documentation

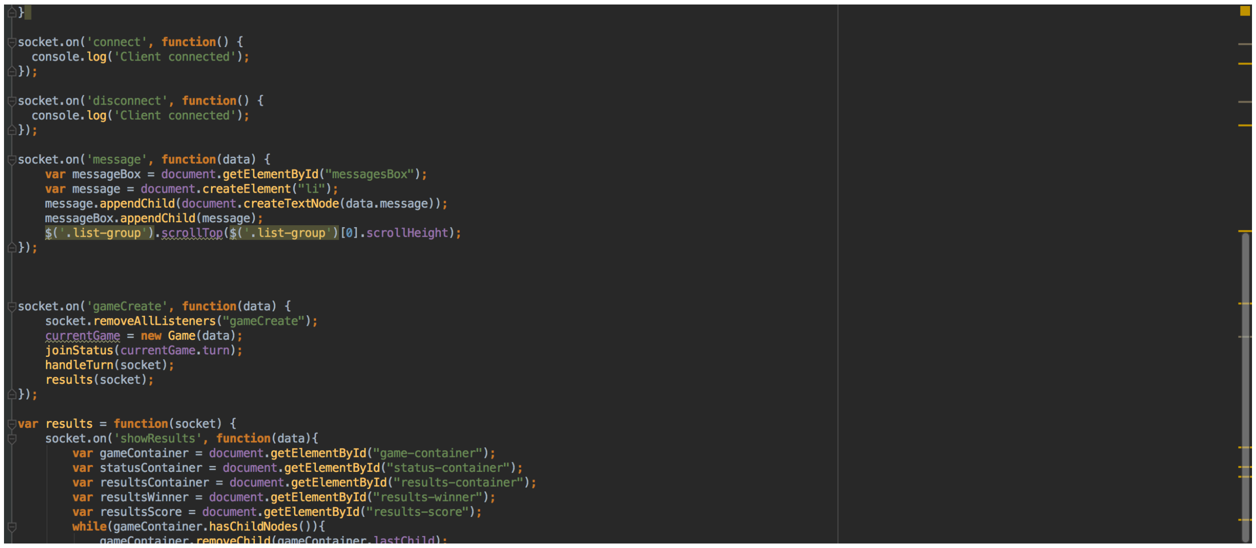
Vikram Grewal, Udara Gunawardena, Grant Gaviglio, Daniel Herrera
ID Numbers (In Order): 917280499, 913147513, 912490506

Github Link: https://github.com/sfsu-csc-667-fall-2017/term-project-csc667-termproject-udara-vikram-grant-daniel

Project Discussion

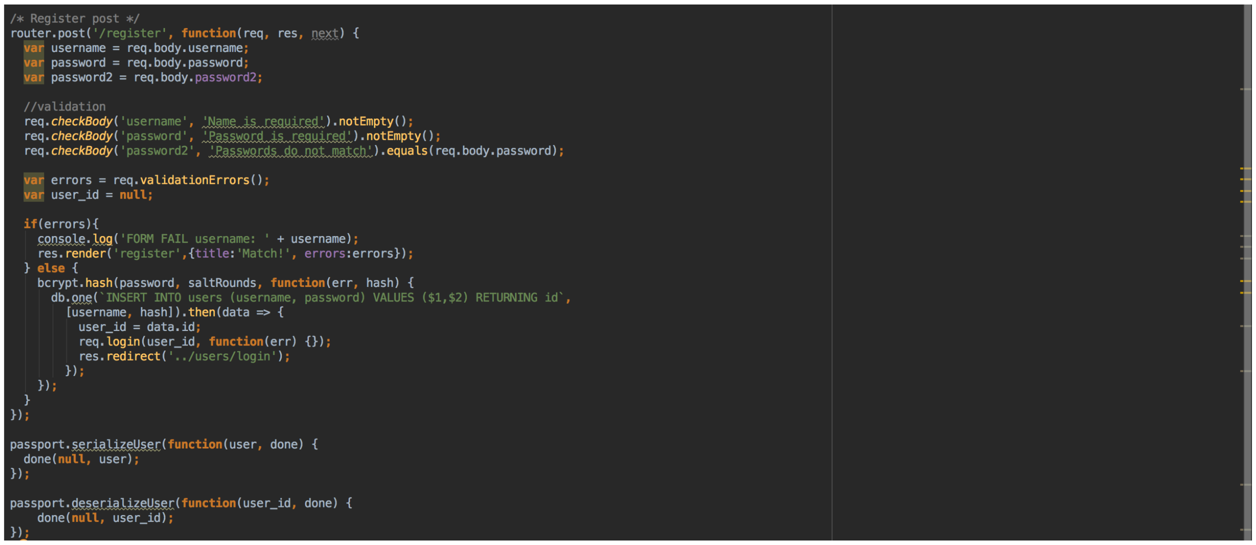
Architecture:

Using Node.js with Express.js framework for the app/game. This allowed us to follow a MVC architecture guideline. PostgreSQL was used for the database as thats the technology heroku uses. Bootstrap was used as the front-end framework and Handlebars.js was used to template our views. The socket.io node module was used to accomplish real-time communication between the clients and server. This was necessary for the chat box and the turn-taking logic for the game. The emit method for socket.io was used to broadcast messages to the different game rooms, such as informing the user that their turn has started. While our express server was running, we were constantly listening to these messages to broadcast to the correct users about what needs to be done next.



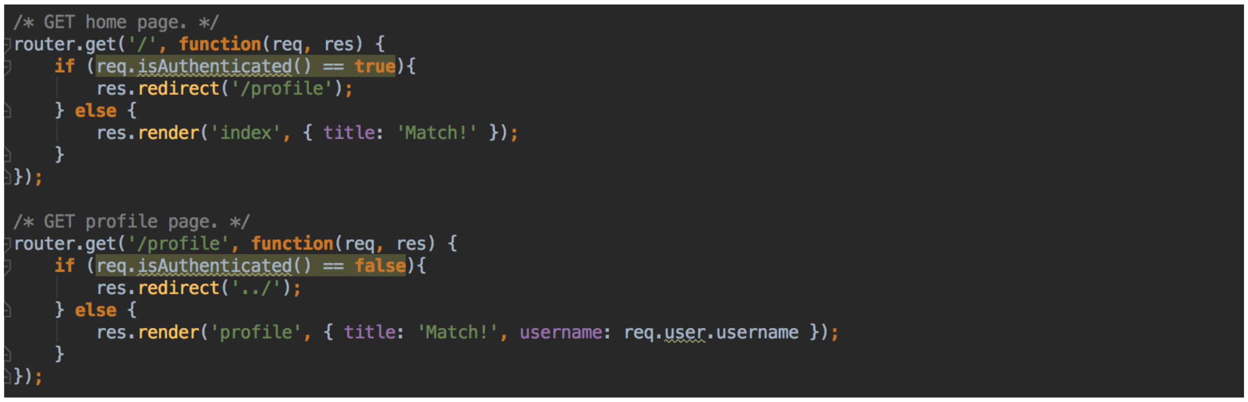
```
}

socket.on('connect', function() {
  console.log('Client connected');
});

socket.on('disconnect', function() {
  console.log('Client connected');
});

socket.on('message', function(data) {
    var messageBox = document.getElementById("messagesBox");
    var message = document.createElement("li");
    message.appendChild(document.createTextNode(data.message));
    messageBox.appendChild(message);
    $('.list-group').scrollTop($('.list-group')[0].scrollHeight);
});


socket.on('gameCreate', function(data) {
    socket.removeAllListeners("gameCreate");
    currentGame = new Game(data);
    joinStatus(currentGame.turn);
    handleTurn(socket);
    results(socket);
});

var results = function(socket) {
    socket.on('showResults', function(data){
        var gameContainer = document.getElementById("game-container");
        var statusContainer = document.getElementById("status-container");
        var resultsContainer = document.getElementById("results-container");
        var resultsWinner = document.getElementById("results-winner");
        var resultsScore = document.getElementById("results-score");
        while(gameContainer.hasChildNodes()){
            gameContainer.removeChild(gameContainer.lastChild);
```

Passport module allowed us not to worry about the registration and login to much since it handles most of the logic. Bcrypt was used to decrypt and encrypt user passwords in the database upon registration and login. Express-sessions was used during development to hold user sessions instead of forcing us to re-login in the event that we needed to reset the server.



```
/* Register post */
router.post('/register', function(req, res, next) {
  var username = req.body.username;
  var password = req.body.password;
  var password2 = req.body.password2;

  //validation
  req.checkBody('username', 'Name is required').notEmpty();
  req.checkBody('password', 'Password is required').notEmpty();
  req.checkBody('password2', 'Passwords do not match').equals(req.body.password);

  var errors = req.validationErrors();
  var user_id = null;

  if(errors){
    console.log('FORM FAIL username: ' + username);
    res.render('register',{title:'Match!', errors:errors});
  } else {
    bcrypt.hash(password, saltRounds, function(err, hash) {
      db.one(`INSERT INTO users (username, password) VALUES ($1,$2) RETURNING id`,
        [username, hash]).then(data => {
          user_id = data.id;
          req.login(user_id, function(err) {});
          res.redirect('../users/login');
        });
    });
  }
});

passport.serializeUser(function(user, done) {
  done(null, user);
});

passport.deserializeUser(function(user_id, done) {
    done(null, user_id);
});
```

Since our application used a MVC architecture, our routes would be the controller of our application. These routes point to different views of our application including our home, game, login, registration, and about views. When a user first arrives at the root of our site, the index route will check if the user is authenticated. If they are not, the user is served the home page view which provides a login or register button. If the user was already authenticated they will be given the profile view for their account.



```
/* GET home page. */
router.get('/', function(req, res) {
    if (req.isAuthenticated() == true){
        res.redirect('/profile');
    } else {
        res.render('index', { title: 'Match!' });
    }
});

/* GET profile page. */
router.get('/profile', function(req, res) {
    if (req.isAuthenticated() == false){
        res.redirect('../');
    } else {
        res.render('profile', { title: 'Match!', username: req.user.username });
    }
});
```

The registration page provides a form which is submit to the database, after this the user is redirected to the login view. From the user profile page the join game button can be clicked which will route to the game view. At this point a connection is made with server via socket.io, and the client checks if there are any open rooms to be joined. If not, a new room is created for this user.

Issues & Resolutions

One of the major problems encountered was getting socket.io to work when it was deployed on to Heroku. Developing locally it was working but eventually we learned that Heroku only allows one port to be used at a time. Socket.io in our initial build was opening a separate port than the server. Solving it involved the server opening an instance of socket.io. Giving it that functionality without having socket.io opening up its own port.

Another problem was with using the socket.io.connect( ) method. We spent time trying to figure out what parameter to give that would work locally and on Heroku and eventually realized it could be left blank and that socket.io would figure it out. A lot of the other problems were very minor and quickly fixed by finding a typo or rereading some documentation.

Test Plan

To test the app we made up use cases for possible commands users and tried them multiple times to make sure they didn’t break. With the game specifically we had players join and leave games in different orders to find some mistakes in our room creating/joining logic. After testing new features locally we would also push the app to Heroku and do the same tests to make sure no other issues were created.