Will Thomas
CS 496
Final Project Summary

Video demo: https://youtu.be/eYuaMiqMXk4
API: URI available here: https://cs496final.herokuapp.com/

The API

This project is mainly an extension of the API I developed for Assignment 3. It's a beer festival tracking app that connects to the API and MongoDB backend that I developed in that assignment. The primary URIs are as follows:

To test the API you will need to make a POST to the registration endpoint with values for the following keys in the request body: email, name, password, confirmPassword.

In all subsequent requests you need to take that token and set it as a header with the key: 'x-access-token'.

Routes following /user/

This route is for users to log in. It checks that their username/PW match what is in the database, and sends back a JSON Web Token for them to use in subsequent calls.

```
//User authentication endpoint. Gives user access JWT
router.route('/authenticate')
        .post((req, res, next) => controller.authenticate(req, res, next));
```

This route is for registration. Does complete error handling for missing fields and mismatched passwords.

```
//User sign-up endpoint
router.route('/register')
         .post((req, res, next) => controller.register(req, res, next));
```

This route requests the user's personal data. It essentially returns the list of events in that user's profile, which is used to populate the "my lists" section of the app.

```
router.route('/mydata')
        .get((req, res, next) => controller.getmydata(req, res, next));
```

This route adds an event to the user's profile, which essentially just modifies the user document in MongoDB by adding the event id to an array which is then populated by MongoDB/Mongoose. It will not submit duplicates, so only one of each event can be listed.

```
//add event to user's list endpoint
router.route('/event')
```

```
            .put((...args) => controller.addEvent(...args))
```

This route is for deleting events from the user's personal list. I had to change it to a POST because I ran into a known issue with the Volley library that I was utilizing to make these requests. When set to a delete the params and headers were coming back empty via Volley. Returns the user's data to populate the list.

```
//Should be a DELETE call but there is a known bug with Volley right with empty params on
//DELETE calls
  .post((...args) => controller.removeEvent(...args));
```

Routes following /event/

I used the GET to populate the "all events" list. I currently do not have an interface for allowing users to submit their own event data, so that was more for testing.

```
router.route('/')
  .get((...args) => controller.find(...args))
  .post((...args) => controller.create(...args));
```

Mobile Feature

I implemented the camera functionality for my app. From the User's personal list area you can invoke the camera and take photos, which get saved to app storage. If I were to continue working on this app I would allow users to upload images to the events themselves, which would have their own profile pages with user submitted photos, reviews, etc.

Account System

I implemented my own account system. In my database there is a user schema to store user information including name, email, password. The user enters their data and it is sent to a function on the server that does error handling before creating the user. The code for that is here:

```
  register(req, res, next) {
    if (req.body.email &&
    req.body.name &&
    req.body.password &&
    req.body.confirmPassword) {

      if (req.body.password !== req.body.confirmPassword) {
        var err = new Error('Passwords do not match.');
        err.status = 400;
        return next(err);
      }

      var userData = new userSchema({
        email: req.body.email,
        name: req.body.name,
        password: req.body.password
      });

      userData.save(function(err) {
        if (err) {
          return next(err);
        }
        res.json({success: true});
      });

    } else {
      var err = new Error('All fields required.');
      err.status = 400;
      console.log(err);
      return next(err);
    }
  }
```

If all fields are incomplete, or if the passwords do not match, it returns an error. This would be easy to extend for further error handling as well, such as email format, etc.

Once the user has registered they can go back to the login page, where they can now enter their name and password. At this point the server looks up the user via the submitted email, and verifies that the email submitted by the form matches what is in the database, as seen below:

```
  authenticate(req, res, next) {
    //console.log(req.body);
    userSchema.findOne({
      email: req.body.email
    }, function(err, user) {
      if (err) throw err;

      //console.log(user);
      if (!user) {
        res.json({success:false, message: 'Authentication failed. User not found'});
      } else if (user) {
```

```
      if (user.password != req.body.password) {
        res.json({success:false, message: 'Authentication failed. Invalid Password'});
      } else {

        var token = jwt.sign(user, thisisasecretmessage, {
          // expiresInMinutes: 1440 // expires in 24 hours
        });

        res.json({success: true, message: 'JWT will go here', token: token});
      }
    }
  });
}
```

If the password matches the server generates a JWT token and sends it back to the application.

The application stores this token in Android's sharedPreferences storage area, which allows you to store it in a key:value format. The token is then used on each subsequent call to the server. When a call is sent, the server has middleware that runs before any of the routes are hit. This middleware was designed as such:

```
router.use(function(req, res, next) {

	//Look for token somewhere in body or header of request. (It should come via headers)
	var token = req.body.token || req.query.token || req.headers['x-access-token'];

	// console.log(req.headers);

	if (token) {

	//Verify token -- providing same secret as it was generated with.
	jwt.verify(token, 'thisisasecretmessage', function(err, decoded) {
		if (err) {
			//Bad token -- error returned.
			return res.json({success: false, message: 'Failed to authenticate token.'});
		} else {
			//If successful, add decoded JWT to request body so it can be used in endpoint.
			req.decoded = decoded;
			//console.log(decoded);
			next();
			}
		});
	} else {

		//No token provided at all.
		return res.status(403).send({
	success: false,
	message: 'No token provided.'
		});
		}
});
```

It verifies the token utilizing the jwt module for express, utilizing the same secret. The user email is pulled from the token and is used to look up user data in future requests. This allows the server to be stateless and not have to consider the user at all. I did not have time to really implement any security features. In reality I would want to ensure I was using encrypted communication and that I was hashing the passwords and sensitive user data. Currently it is all being saved as plain text.

When the user opens the app it checks that a token is available. If there is a token in storage it directs the user straight to the my list page. If there isn't one the user must enter their username and password to log in. If the user chooses to log out then the token is deleted and they are directed back to the login page.