(Added this to pom.xml)

```
<properties>
  <java.version>1.8</java.version>
  <maven.compiler.source>${java.version}</maven.compiler.source>
  <maven.compiler.target>${java.version}</maven.compiler.target>
  <jjwt.version>0.11.1</jjwt.version>
   <maven-jar-plugin.version>3.1.1</maven-jar-plugin.version>
</properties>
```

Description:

This is a 2 day review project that extends the Week 3 review project. Last week, we built a Vue.js front-end to display and filter through a list of homes. The goal behind this week is to continue the application and tie the Vue.js front-end from last week, to the Java backend we built in earlier modules. Another goal is to show them the capstone base project and how to use it (database scripts, JWT, etc)

We are going to use the Final Capstone project as our base.

Database Setup

1. The capstone project comes with a database script that takes care of creating the database, setting up an application user and and capstone user. Walk the students through the basic project setup on the Java side and creating the database:

   a) Show the location of the database scripts and instructions
   b) execute the script `./create.sh` from the `database` directory
   c) Walk through the two users it created:

   `final_capstone_owner` - This is the admin user with full privileges
   `final_capstone_appuser` - SELECT, INSERT, UPDATE, DELETE privileges only. This is the one configured in your application datasource

2. Connect the database to DBVisualizer using the `final_capstone_owner` id so we can make changes if necessary. `finalcapstone` is the password.

3. Show in DBVisualizer that a users table has been created with two default users:

   user
   admin

4. Review the database design - For simplicity, we are going to create a single table

5. Add the following to the

```
DROP TABLE IF EXISTS home;



CREATE TABLE home
(
  id serial,
  mlsNumber varchar(32) not null,
  address varchar(64) not null,
  city varchar(50) not null,
  state varchar(50) not null,
  zip varchar(10) not null,
  price numeric(12,2) not null,
  imageName varchar(50),

  constraint pk_home primary key (id)

 );


INSERT INTO home (mlsNumber, address, city, state, zip, price, imageName)
VALUES ('1000', '123 Java Green Lane', 'Columbus', 'Ohio', '43023', '1,222,345.00',
'1000.jpg');
INSERT INTO home (mlsNumber, address, city, state, zip, price, imageName)
VALUES ('1001', '123 Vue Street', 'Grandview', 'Ohio', '43015', '400,250.99',
'1001.jpg');
INSERT INTO home (mlsNumber, address, city, state, zip, price, imageName)
VALUES ('1002', '5555 Java Blue Ct.', 'Columbus', 'Ohio', '43023', '450,000.00',
```

```
'1002.jpg');
INSERT INTO home (mlsNumber, address, city, state, zip, price, imageName)
VALUES ('1003', '99 C-Sharp Road', 'Dublin', 'Ohio', '43017', '345.00', '1003.jpg');
INSERT INTO home (mlsNumber, address, city, state, zip, price, imageName)
VALUES ('1004', '175 Cohort Lane', 'Pickerington', 'Ohio', '43065', '400,000.01',
'1004.jpg');
```

6. execute the script `./create.sh` from the `database` directory to load the database

7. Verify everything is ready to go…

Eclipse

Walk through Eclipse:

1. Show application.properties → Show the datasource properties
2. Show AuthenticationController and Security pieces (We will look at JWT later)

3. Create a domain object: Home

```
        private String mlsNumber;
        private String address;
        private String city;
        private String state;
        private String zip;
        private double price;
        private String imageName;


** generate Getters/Setter
```

4. Create a HomeDAO Interface

```
List<Home> getHomes();
```

5. Create a HomeSqlDAO Implementation

```
package com.techelevator.dao;

import java.util.ArrayList;
import java.util.List;

import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.support.rowset.SqlRowSet;
import org.springframework.stereotype.Service;

import com.techelevator.model.Home;
import com.techelevator.model.User;

@Service
public class HomeSqlDAO implements HomeDAO {

   private JdbcTemplate jdbcTemplate;

   public HomeSqlDAO(JdbcTemplate jdbcTemplate) {
      this.jdbcTemplate = jdbcTemplate;
   }

         @Override
         public List<Home> getHomes() {

                  List<Home> homes = new ArrayList<Home>();

                  String sql = "SELECT * FROM home";
                  SqlRowSet results = jdbcTemplate.queryForRowSet(sql);


      while(results.next()) {
         Home home = mapRowToHome(results);
         homes.add(home);
      }

      return homes;
```

```
        }

    private Home mapRowToHome(SqlRowSet rs) {
        Home home = new Home();
        home.setMlsNumber(rs.getString("mlsNumber"));
        home.setAddress(rs.getString("address"));
        home.setCity(rs.getString("city"));
        home.setImageName(rs.getString("imageName"));
        home.setPrice(rs.getDouble("price"));
        home.setState(rs.getString("state"));
        home.setZip(rs.getString("zip"));

        return home;
    }

}
```

6. Create a HomeController

```
package com.techelevator.controller;

import java.util.List;

import org.springframework.web.bind.annotation.CrossOrigin;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;

import com.techelevator.dao.HomeDAO;
import com.techelevator.model.Home;


@RestController
@CrossOrigin
public class HomeController {

        private HomeDAO homeDAO;

    public HomeController(HomeDAO homeDAO) {
```

```
        this.homeDAO = homeDAO;
    }


        @RequestMapping(path = "/homes", method = RequestMethod.GET)
        public List<Home> getHomes() {
                return homeDAO.getHomes();
        }

}
```

7. Run the app and test using Google chrome. You should now have a web service response with an array of Homes in JSON format.

8. Be sure to have the Eclipse server running on port 8080 before working on Vue. Tomcat in Eclipse must use port 8080.

Vue.js

1. Load the Vue.js application in VSC

2. Walk Through the readme.

    a) Run npm install
    b) Open up the .env file and delete the C# URL
    c) Run npm run serve to show the basic homepage

3. Walk through the router's index.js file…

    a) Show the meta: {
        requiresAuth: true in the Home route
    }

    b) We are going to change it to false so we can load the main home page. The main

home page in our case does not require login. Show how these are navigation guards.

4. Now show the /register page. Explain how this is working to create a new User account, but that it will need styled. **Do not register a new user now. We will do that later. We will come back and look at Authentication (Login/Logout) and Registration at a future time**

5. Now we need to start pulling in our code from the previous Review day. Since the capstone includes a lot of new stuff (axios, authentication, registration, router, etc) rather than try to make the old project work, we can just pull in the files we need from our old project…

   a) Let's pull in the asset folder which contains our images
   b) Let's pull in our 3 components (Header, Footer, HomeSearch)
   c) Let's copy the state object from the old vuex store that contains our hardcoded data (We will only use this to verify our component works, then we throw away and replace with our web service call)
   d) Add the following before the token and user props inside the state object

```
homes: [
  {
    mlsId: '1000',
    address: '123 Java Green Lane',
    city: 'Columbus',
    state: 'Ohio',
    zip: '43023',
    price: '1,222,345.00',
    imageName: require('../assets/1000.jpg')
  },
  {
    mlsId: '1001',
    address: '123 Vue Street',
    city: 'Grandview',
    state: 'Ohio',
    zip: '43015',
    price: '952,345.72',
    imageName: require('../assets/1001.jpg')
  },
  {
    mlsId: '1002',
    address: '123 Java Blue Court',
    city: 'Columbus',
    state: 'Ohio',
```

```
      zip: '43023',
      price: '750,000.00',
      imageName: require('../assets/1002.jpg')
    },
    {
      mlsId: '1003',
      address: '999 C-Sharp Rd.',
      city: 'Dublin',
      state: 'Ohio',
      zip: '43017',
      price: '99.97',
      imageName: require('../assets/1003.jpg')
    },
    {
      mlsId: '1004',
      address: '555 Cohort Lane. Apt. 1',
      city: 'Columbus',
      state: 'Ohio',
      zip: '43022',
      price: '1,000,000.01',
      imageName: require('../assets/1004.jpg')
    }
 ],
```

6. Now let's go modify our existing App.vue file so we can add in the header and footer.

   a) Add the following to our App.vue:

```
<script>
import TheHeader from './components/TheHeader.vue'
import TheFooter from './components/TheFooter.vue'

export default {
  name: 'App',
  components: {
    TheHeader,
    TheFooter
  }
}
</script>


<style>
```

```
#app {
  font-family: Avenir, Helvetica, Arial, sans-serif;
  -webkit-font-smoothing: antialiased;
  -moz-osx-font-smoothing: grayscale;
  text-align: center;
  color: #2c3e50;
  margin-top: 60px;
}
</style>
```

b) Now we need to move the nav bar stuff from the existing App.vue because we want these in our Header.vue, not our main component. Be sure to explain what they are doing after we move them.

```
<div id="nav">
      <router-link v-bind:to="{ name: 'home' }">Home</router-link> | 
      <router-link v-bind:to="{ name: 'logout' }" v-if="$store.state.token != 
"">Logout</router-link>
</div>
```

c) Now we need to add the header and footer to our App.vue file and place the router-view in between the two

d) Let's clean up the nav bar…. We will create a new route later for the Home Search but for now we will be leaving it set to 'home' so it will compile.

<u>Based off the Logout link, how could we create a login button that will show when a user is logged out?</u>

```
 <router-link v-bind:to="{ name: 'home' }">Home Search</router-link>
<router-link v-bind:to="{ name: 'login' }" v-if="$store.state.token == "">Login</router-link>
```

7. So before we hook up the web service, let's get the route set up to load the HomeSearch component and get it tied into the Navigation bar properly.

   How would we do that?

   For starters we need to create a view to represent the Home Search Page.

   a) Let's create a new view component in the views folder, called Search.vue
   b) Let's import the HomeSearch component:

```
<template>
  <div>
    <home-search></hom
  </div>
</template>

<script>

import HomeSearch from '@/components/HomeSearch';

export default {

   components: {
      HomeSearch
   }

}
</script>

<style>

</style>
```

   c)

8. Now we can set up the route to link the nav bar router-link to the Search page. Go to the route configuration and add:

```
import Search from '../views/Search.vue'


{
      path: "/search",
      name: "search",
      component: Search,
      meta: {
        requiresAuth: false
      }
    },
```

*** We will not require login at this time….

9. Go to the Header.vue and update the router link to point to ‘search’.

```
<router-link v-bind:to="{ name: 'search' }">Home Search</router-link>
```

10. Let’s test it!

11. Now let’s go back to the route and set the search route to require authorization. Let’s test it!

Vue Web Service

Now let’s hook up the web service.

1. We need to create a service class to call our web service. Let’s call it HomeService.js

```
import axios from 'axios';
```

```
export default {

  search() {
    return axios.post('/homes')
  }

}
```

2. We need to add the following props to the data section

```
  homes: [],
  isLoading: true
```

3. Fix the filteredHomes() to use the homes[] we just added:

```
  computed: {
      filteredHomes() {

        return this.homes.filter(home => {
          return this.zipFilter === '' ? true : this.zipFilter === home.zip;
        });
      }
  },
```

4. Now let's add the created() hook to call the service and update the homes[]

```
    created() {
        homeService.getHomes().then(response => {
        this.homes = response.data;
        this.isLoading = false;
    });
```

5. Now when we run it, we should see some results…. Test the filter, etc…

6. Now let's fix the broken images….

```
<div class="divTableCell">
   <img v-bind:src="getImageUrl(home.imageName)" />
</div>
```

We need to call a function so we can use the require() function. This forced the image to be loaded at run-time, not before… Note: We only need to do this with dynamic images coming from data rendering {{ }} …

7. Let's create the function….

```
methods: {
  getImageUrl(pic) {
    return require('../assets/' + pic)
  }
}
```

8.