Project Part 2: Server-side Buildout
Due: (tentative) 4/17
150 Points

Purpose:
We are going to start building out a front end framework that will be grown upon in future labs and projects. We are going to base our project on an old game called “Oregon Trail” that was released in the 70’s. Here is a link to play the game online:
https://classicreload.com/oregon-trail.html

The second part will get our initial server side pieces in place. We will be creating our backend folders and files that will be our restful API that our front end will call to get and send data. Once we have, we will create 3 layers that together constitute the back end of our system. They are the Model, Controller and Routing layers. We will build these up for all of the server side functionality that our server will need to provide the front end. Once we are done with this lab we will have a complete RESTful API ready for any client software to connect to and use. You will even be able to play your game directly using the APIs

Preparatory:
You will need everything working from our previous labs, and also you should have finished the first project (Part 1: Client side).

Submission:
Create a **release (tag)** of your project when it is ready in github and submit the iLearn lab assignment with a link to your git repository release.

Summary of **Major** Pieces:

1. Create back end folders and files for controllers and models
2. Create the **data model** that will manage our in memory representation of the games data. For now there will only be provisions to use this information for a single player using your game. Later in the course we will discuss sessions which we can use to allow us to save a copy of all of the games data for each player.
3. Create **controllers** that will receive the requests to get or save data. The controllers responsibility is to translate the message received (in our case this will be a JSON message) to the action that needs to be performed, which usually includes interactions with our data model (either getting, updating, deleting or saving data).
4. Create **RESTful APIs (routing layer)** using our oregonTrail.js. The routes responsibility is to receive the message as it came into the webserver from the client, determine what type of action needs to be taken, and call the correct controller method to satisfy the request.

5. Create and manage **instances** of our data model. Putting **static** and **non-static** data in the appropriate places.
6. Test our APIs using our browsers and tools like **Postman (we will discuss in lab)** to send requests.

Part 1: Create your backend files and folders and link them together

Just like we built out the client side files and structure in the first part of the project, we are now going to create the server side structure.

1. Create a sub-folders in the server folder called models and controllers
2. Create the following 5 files in the models folder. These files represent your data models. You will create instance(s) of these classes in your code to store and manipulate data in memory. For now all data will be in memory (we will add a database component in Part 3 of the project).
    a. gameData.js
    b. terrain.js
    c. weather.js
    d. topTen.js
    e. pace.js
3. Create the following 3 files in the controllers folder. These files are the controllers which will handle the business logic of your server side code. They will be invoked from the routes you create in your oregontrail.js file and will manage non-static instances of data created from your data model classes.
    a. gameController.js
    b. setupController.js
    c. topTenController.js
4. Understand and use exports: In order for your oregontrail.js (routes) file to utilize your code in your controllers, and for your controllers to utilize your models, we use an array called exports. This array is called module.exports and is managed by Node.js. It is a way to expose anything you want in a module while leaving everything else private. This is basically a really creative approach to encapsulation in JavaScript. You can also use “exports” as shorthand for “module.exports”. In your classes you will want to assign anything (values or functions) that you wish to be accessible where you include the file with a require() to the exports array. I may be an object like this;

    ```
    exports.gameSettings = new gameData();
    ```

    Or a function like this:

    ```
    exports.getAllPaces = function() {
    ```

```
        return(allPaces);
    }
```

In another file that requires the one with the above getAllPaces function, i could call that function like this:

```
var pace = require('../models/pace');
pace.getAllPaces();
```

But the above will not work from another file if you do not include the function in exports.

We will discuss this more in Lab, you can also see here: https://stackoverflow.com/questions/5311334/what-is-the-purpose-of-node-js-module-exports-and-how-do-you-use-it

5. Understanding Require (think import statements in Java): When you want to use code in another file in your models and controllers, you use the require() method as seen above when we wanted to access a function that we had added to the exports array.
    a. We have already seen this when we used the require statement to include libraries that we wanted to use within node.js. Now we are going to use to use other files from our own program. If a controller needs to use data or methods from one of your model classes, you should use require() in the controller file, passing in the path (without extension!) to your local file. Here are some examples, these include 3 model classes, allowing you to access them locally in the terrain, weather and pace variables.

```
/**
* link to required models
*/
var terrain = require('../models/terrain');
var weather = require('../models/weather');
var pace = require('../models/weather');
```

Part 2: Server Side Requirements

Now that we have gotten creating the folders and files and some basic node.js concepts we need down, we can start building the game.

Before we jump in though, we need to understand the games requirements, so we can break down the details of what objects we need, what data they need to save and what operations or methods our controllers and classes need to have in order to satisfy the requirements.

Here are some high level requirements to get us started, please spend so time playing the actual game that we are modeling this off of (see link at the top), as it will help add context to everything below. Some of the requirements are modified from the actual game however to keep the scope under control.

Part 2.1: High-level requirements:

You have to travel a total of 500 miles and maintain the health of your players during the journey which must be completed in 45 days.

Health is represented by a number between 0 and 100. You start with 100 health, different things in the game will affect your health in both positive and negative ways, these include your pace (how many miles your group travels in a day), the weather and if you take time to rest.

Your score at the end of the game will be determined by your group health level and the number of days you finish ahead of schedule. If you do not complete the journey of 500 miles within 45 days your party will be lost in the snowy mountains and eat each other.

You can change your pace (steady, strenuous, grueling, resting) by pressing spacebar during the game play and choosing a new pace.

Your game screen UI should include a graphic for each of the possible terrain types (Mountains, grassland, plains, forest, desert), It should also have a wagon graphic that makes its way across the screen from right to left. The starting point on the right represents mile 0 and the ending point on the right will represent mile 500.

Your UI should also have a info box that reports status updates such as weather and health issues and members of the party dying. (details of weather and health below)

Part 3 Building the 3 Layers (Data model, Controllers, Routes)

We are going to have 3 distinct software “layers” on our server side, they will be the model, controllers, and routing layer. This document will describe the layers and outline requirements in the order I just listed them, starting with the model, then building the logic for the model and finally exposing the API in the route.

Part 3.1 Data Model Layer

Now I will start you off with suggested model classes based on the requirements section above, you are free to change these as long as the requirements are met.

You will need to create the following objects in the corresponding javascript files in the models folder:

- GameData
- Pace
- Terrain
- Weather
- TopTen

Pace will need to store:

- name: name of the pace, there are 4 in real game and they are the minimum requirement for our class: Steady, Strenuous, Grueling and Resting
- miles: # number of miles a player will move in day if they are traveling at this pace.
    - Steady: 20 miles
    - Strenuous: 30 miles
    - Grueling: 35 miles
    - Resting: 0 miles
- healthChange: The effect moving at this pace has on health
    - Steady: no effect
    - Strenuous: -3 health / day
    - Grueling -8 health / day
    - Resting +5 health / day

Terrain will need to store:

- name: The name given to this terrain. You should have at least 4 different types of terrain.
- imageUrl: the url of the image that will be displayed in the UI when the user is traversing this type of terrain.
- Optional: I am not requiring that the terrain have an effect on your miles covered, but it would make an excellent addition.

Weather will store:

- id: integer id used to identify it.
- type: You need all of the ones in the chart below (ex. “Very Hot”, and “Rain”)
- healthChange: amount this weather type effects player health per day (see chart)
- mileChange: percentage of pace player can cover in this weather type (see chart)
- probability: the chance of getting this weather on any given day (see chart)

More weather details ----------

- There are 11 types of weather (see weather chart)
- Severe and inclement weather has an affect on group health.
- Severe and inclement weather has an affect on miles traveled in a day
- There are probabilities of occurrence for each type of weather, one type of weather is chosen for each day

Weather breakdown:

Weather	**Health change**	**Mile change**	**Occurrence Probability**
Very Hot	-8 / day	.7	.1
Hot	-3 / day	.9	.1
Warm	+1 / day	1	.2
Cool	+1 / day	.95	.1
Cold	-5 / day	.8	.1
Very Cold	-12 / day	.7	.1
Rain	-4 / day	.6	.1
Heavy Rain	-8 / day	.4	.05
Snow	-15 / day	.3	.05
Blizzard	-30 / day	.1	.05
Heavy Fog	-3 / day	.5	.05

TopTen will store:

- playerName: name of the player who earned the score
- playerScore: score earned for game
- dateEarned: date the player earned this score

GameData is the core object we will be working with for our game here is what you have to store:

- playerNames: store the names of the 5 players in the wagon team. (chosen at start)
- playerStatus: You need to keep track of whether or not each of the 5 players is alive
- playerProfession: The profession chosen by the player at the start (choices are: banker, carpenter or farmer) (chosen at start)
    - Note that the profession chosen has an effect on the starting money, see real game link at the top
- playerMoney: the amount of money the wagon team has
- startMonth: The month the game starts in (chosen at start)
- milesTraveled: you have to keep track of how far the wagon team has progressed
- groupHealth: A single representation of wagon team health (see **Group Health** section below for more detail!)

- currentPace: assign the pace (see objects above) that the wagon team is currently traveling at
- daysOnTrail: how many days (1 day = 1 update) has the wagon team been on the trail
- currentWeather: assign the current weather (see objects above) for this update
- currentTerrain: assign the current terrain (see objects above) for this update
- messages: Text to show the user on the client side (ex. Player dies, etc)

Group Health:

- Represented by number in the range of 0-100
- Each player must track whether they are alive / dead individually (boolean) but the health number is representative of the group as a whole. Low health will increase the chances of a player dying as noted below. If all 5 players die the game is lost.
    - >=80: good
    - >=50 and <80: fair
    - >=20 and <50: poor (3% chance each player dies per day)
    - >0 and >20: very poor (10% chance each player per day)
    - 0: everyone in the party is dead, you lose.
- Some elements of gameplay will affect the group health number, they include: (specifics on how they affect health included with each type, below)
    - Weather
    - Pace

Part 3.2: Controllers

We are going to look at examples of controllers in our lab, and also when we talk about common application architectures and RESTful APIs in class. Your controllers are the glue that is sent the request from the route, opens the request (gets any information sent from the client) and then works with the model, querying or changing the model as needed. The controller then usually responds to the client with information requested or a confirmation that everything worked or failed.

The controllers we are working with will each handle functionally that is grouped together.

- gameController: Will have all the logic for playing the game, methods should include
    - changePace()
    - updateGame()
    - resetGame()
    - getGameData()
    - and possibly more.
- setupController: Will contain all the methods and logic to support creating a new game. This is the controller that contains the logic that will be called when you go through the setup process creating the names of the players, picking a profession and start month. This is best experienced by using the link above to play the real game. Methods should include:

    - getGameScreen()
    - saveProfession()
    - savePlayerName()
    - saveStartMonth()
    - And possibly more.
- topTenController: This controller will contain all methods and logic needed to get and save to the top ten list. This controller will manage saving topTen scores and providing the sorted list of scores to be shown in the UI. Methods should include:
    - getTopScores()
    - saveTopScore()
    - And possibly more

Here is an example of one controller method that will be in gameController:

```
exports.getGameData = function(req, res) {
        // return json of the game data
        res.setHeader('Content-Type', 'application/json');
        res.setHeader('Access-Control-Allow-Origin', "*");
        res.send(gameData);
}
```

This is a good first example as it has the simple function of returning the current instance of the gameData object. It simply sets the correct mime type (application/json) and sends the object (gameData) back in the response to the client.

Here is another example of a method that will be in setupController:

```
exports.saveStartMonth = function(req, res) {
        gameController.getData().startMonth = req.body.month;
        res.setHeader('Content-Type', 'text/plain');
        res.send(gameController.getData().startMonth);
};
```

This method responds to a request by a client to save data. The data was sent in the body of the request via the POST method. This code extracts the month value from the body and saves t in a member called startMonth in an instance of the gameData object in another controller (gameController).

Part 3.3: Routing

Our routes for our back end are going to be proper RESTful routes. We will discuss these in class and in the lab. You need to include the following in all your routes:

- The base url for the route

- The internet media type (mime type)
- Standard HTTP methods (we will use GET, POST, (PATCH or PUT), and DELETE)

I will be giving out a final list of routes you must support soon. These are the routes I will be setting up my test script for so they must work.

Here is an example RESTful api, this one includes the route that calls the example controller above which is doing a update operation, but it also contains routes to handle getting information, saving new information, and deleting information. There are also 2 variations on the url for this route, the second one includes a url parameter that is the id of the player you want to get, update or delete. The first route acts on the collection of players, the .get there will return all players, the .post will save a new player (they do not have an id yet).

```
app.route('/api/setup/player')
        .get(setupController.getAllPlayerNames)
        .post(setupController.savePlayerName);
app.route('/api/setup/player/:id')
        .get(setupController.getPlayerName)
        .patch(setupController.changePlayerName)
        .delete(setupController.deletePlayerName);
```

Part 4: Discussion on Static vs Instance data

An important thing to consider is where you instantiate your model classes. You will have to ensure that the data is available to you when you need it, and protected when you do not. Remember that static data is data that is not expected to change verse non-static data that is going to change. The rule of thumb on where to create your model instance follows this rule:

- If the data is static it the instance can live in the model class
- If the data is non-static (you expect it to change) it should live in a controller.

Let's look at an example. The weather objects you create (there are 11 of them according to the specs) are not expected to change. Once you create them, you can use the same 11 throughout the whole game. This means that having the instances created and accessible from the weather.js file (using exports) in the models folder is a good idea. If you have more then one controller importing the model, both will be able to see the same data.

But what about the gameData object? This object is going to have information like the health of your group and what the **current** weather is. These pieces of data change, therefore they should not be stored in the model. You can use an analogy in Java to help show the point. If you have a User.java class that define a User, you should not create non-static instances of the user in the User.java class, instead you would create the instance in any java files utilizing the data model, It is the same in node.js. So where should we create the instance? The answer for

us is the controller. We can create the gameData instance of one of our controllers (most likely gameController). Now what if you need to use the gamaData instance in more than one controller? (this will likely be the case, as gameData also includes all your playerNames, startMonth, and other setup information that is defined in setupController) The answer is you can import one controller from another. (gameController can be imported by setupController) What you want to make sure you do not do is create an instance of gameData in both gameController and setupController as the data will be different between them!

Part 5: Grading breakdown

- All folders and files required exist
- All model files are created
- All controller files are created and work properly
- Routes are created
- Routes use correct RESTful API approach
- All APIs work (I am going to have a script to test this!)
- Model classes have data needed to fulfill requirements
- Controllers service requests, modify/add/get/delete data in model and return response