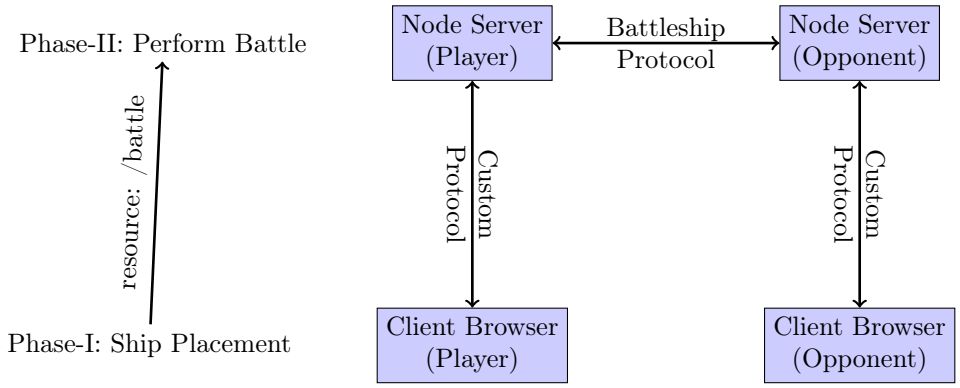
Assigned: 2019-03-23
Due Date: 2019-04-06

Summary:

Abstract:	Battleship Game Protocol
Objectives:	1. Node Route Class 2. Node Modules - Export and Import 3. Application Protocol 4. REST Design, End-Points, Methods 5. SSE, JSON 6. HTTP Client in Node Server
Grading:	45 pts - A $\geq$ 41.85; A- $\geq$ 40.50; B+ $\geq$ 39.15; B $\geq$ 37.35; B- $\geq$ 36; C+ $\geq$ 34.65; C $\geq$ 32.85; C- $\geq$ 31.75; D+ $\geq$ 30.15; D $\geq$ 28.35; D- $\geq$ 27
Outcomes:	R2 (CAC-a,c,i, j, k; EAC-a, b, c, e, k, 1, 2); R5 (CAC-a, i; EAC-a, k) (see syllabus for description of course outcomes)

Project Description:



During Phase-I of the game, the end-user uses the client browser to either (a) place their ships on their ocean grid, or (b) load a pre-configured ocean grid from their node server. Once their ships have been placed, the user initiates a battle request by performing a request to the `/battle` resource (see below for details). This should disallow any changes to the player's ocean grid and initiate Phase-II of the game, performing battle with an opponent.

During the Phase-II of the game, the user initiating the session request will be known as the *player* and the user fulfilling the session request with a response will be known as the *opponent*. The process of battle follows the sequence of *request* and *response* pairs between the player's node server and the opponent's node server.

Create a new route class to implement the Battleship game protocol. The game protocol must be implemented exactly according to the specification provided herein or the ability to participate in a game with another player will not work as expected. The route class will need to import a reference to a function exported from the SSE module that will allow code within the protocol module to send messages to the front-end client game board application via the HTML5 Server Sent Events API.

Also, implement a tile selection strategy used when firing on your opponents ships. This strategy can be as sophisticated as you like, or completely random. The starter files for this project already implement

a completely random tile selection strategy. A reasonable strategy should operate in two phases - the first when no opponent ship is presently hit, and the second when an opponent ship has been hit. Be careful with this component of your project as the tile selection strategy should not take a large amount of processing time given the single-threaded nature of the Node server.

Battleship Game Protocol:

Resource End-Point: [GET]/battle/:filename[/:url] Change to the battle mode and optionally engage with opponent at `url`. Load the game state from the file given in `filename`. If the `url` is not specified, load the game state and then wait for an opponent to request a session (see `/session` endpoint below).

Request Body: *There is no request body.*

Status Codes:

200 OK - Entered Battle Mode (see response message)

400 Bad Request - Server could not understand the request

401 Unauthorized - Not an authorized client of the node server

404 Not Found - Game state filename could not be found on the server

Response Body if Status Code 200:

```
{
   // game model - bsState JSON
}
```

Postcondition: Server is in phase-II, battle mode.

Response Body if Status Code 400:

```
{
   'request': // full request provided
}
```

Postcondition: Server remains in phase-I.

Response Body if Status Code 401:

```
{
   'links': {
      'auth': {
         'href': 'https//csdept16.mtech.edu:XXXXX/auth',
         'rel': '/auth'
      }
   }
} // XXXXX - port assigned to each student node instance
```

Postcondition: Server remains in phase-I.

Response Body if Status Code 404:

```
{
   'filename': // filename provided in request
}
```

Postcondition: Server remains in phase-I.

Resource End-Point: [POST]/session Request to engage in a battle session with an opponent identified in the `url` of the `battle` request.

Request Body:

```
{
   'opponentURL': // your url with port,
   ['latency': 5000] // optional latency between 2000 - 10000 ms
}
```

Status Codes:

200 OK - If the opponent creates a session with the player (requester)

400 Bad Request - Server could not understand the request

403 Forbidden - Opponent is already in another session

412 Precondition Failed - Opponent is not in *battle* phase

Response Body if Status Code 200:

```
{
   'session': // session identifier ,
   'roll': // 1 or 0,
   'names': // [<system name text>, <player name text>],
   'epoc': // unix time stamp (in ms since 1/1/1970 00:00:00),
   'latency': 5000 // value between 2000 - 10000 (in ms)
}
```

- Roll = Math.floor(Math.random()∗2);
  Player's turn when roll == 0;
  Opponent's turn when roll == 1.
- epoc = (**new** Date()).getTime();
  Number of milliseconds since Unix EPIC (1/1/1970 00:00:00)
- System name is the text name of the system providing the response.
  Must be between 3 and 20 characters.
- Player name is the text name of the user of the system.
  Must be between 3 amd 20 characters.
- The session-id is comprised of:
  - md5(localIPAddr + removeIPAddr + epoc);
  - localIPAddr is the IP address of the node server forming the response
  - removeIPAddr is the IP address of the node server of the requester
  - epoc is as previously computed
  - MD5 is the message digest hash of the concatenated strings as shown
- Latency request:
  - Is just a request,
  - must be between 2000 and 10000 ms (2s and 10s respectively)
  - Is completely optional for honoring the request
  - default should remain 5000 ms

POSTCONDITION: Battle session established; servers take turns issuing `/target` endpoint requests until the game is completed.

Response Body if Status Code 400:

```
{
   'request': // full request provided
}
```

POSTCONDITION: Server remains in battle mode; session with opponent is not established; error should be reported to client front-end.

Response Body if Status Code 403:

```
{
    'opponent': ['system name', 'player name']
}
```

POSTCONDITION: Server remains in battle mode; session with opponent is not established; error should be reported to client front-end.

Response Body if Status Code 412: *There is no response body.*

POSTCONDITION: Server remains in phase-I.

Resource End-Point: [DELETE]/session/<session−id Request to terminate existing session with an opponent and reset game state

Request Body: *There is no request body.*

Status Codes:

200 OK - If the opponent terminates the session

400 Bad Request - Server could not understand the request

403 Forbidden - No access rights to delete session-id

404 Not Found - Provided session-id is not valid

412 Precondition Failed - Opponent is not in *battle* phase

Response Body if Status Code is 200:

```
{
    'session': 'session-id',
    'duration': // total time spent in session (in ms)
}
```

POSTCONDITION: Session is destroyed; server leaves phase-II; server enters phase-I.

Response Body if Status Code is 400:

```
{
    'request': // full request provided
}
```

POSTCONDITION: Session remains intact.

Response Body if Status Code is 403:

```
{
    'session': 'session-id'
}
```

POSTCONDITION: Session remains intact.

Response Body if Status Code is 404:

```
{
    'session': 'session-id'
}
```

POSTCONDITION: Session remains intact.

Response Body if Status Code is 412: *There is no response body.*

POSTCONDITION: Server remains in phase-I.

Resource End-Point: [POST]/target Fire onto your opponent's target grid at a given tile location.

Request Body:

```
{
   'session': 'session-id',
   'tile': // tile specification
}
```

- tile == [A..J][0..9], in *row*, *column* format.

Status Codes:

200 OK - Opponent accepts the target request

400 Bad Request - Server could not understand the request

401 Unauthorized - Not a valid session-id

403 Forbiddien - Not the opponent's turn

412 Precondition Failed - Opponent is not in *battle* phase

Response Body if Status Code is 200:

```
{
   'status': 'MISS' | 'CARRIER' | 'BATTLESHIP' |
             'CRUISER' | 'SUBMARINE' | 'DESTROYER',
   'tile': 'RC',
   'disposition': 'INPROGRESS' | 'WIN'
}
```

POSTCONDITION: After processing response, appropriate SSE message should be sent to the front-end indicating disposition of the target attempt; turn should be updated.

Response Body if Status Code is 400:

```
{
   'request': // full request provided
}
```

POSTCONDITIN: Turn should not be updated.

Response Body if Status Code is 401: *There is no response body.*

POSTCONDITION: Turn should not be updated.

Response Body if status Code is 403: *There is no response body.*

POSTCONDITION: Turn should not be updated.

Response Body if Status Code is 412: *There is no response body.*

POSTCONDITION: Turn should not be updated.

Obtaining Project Files:

1. Logon to `gitlab.cs.mtech.edu` and locate the project `bsBattleProtocol` under the `S20 CSCI470` (sub)group, and then fork this project into your own account.
2. Navigate to your own account and locate the `bsBattleProtocol` project just forked, and copy the project url
3. Next, logon to `csdept16.mtech.edu` using your Department username and password.
4. Execute the `mkdir ~/CSCI470/Projects/` command, which will create a projects folder if not already created.
5. Execute the `cd ~/CSCI470/Projects` command, which will change the current working directory to the specified parameter.
6. Issue the command `git clone <project_url>`, where `<project_url>` is the url you copied in the above step. This will create the project folder inside your `~/CSCI470/Projects` directory,
7. Execute the command `cd bsBattleProtocol` to enter the project directory.
8. Continue with the specific project activities below.

Project Activities:

Please perform the following activities in the completion of the lab assignment.

1. After cloning the project, navigate to its folder and incorporate your previous project files (route classes and and changes in `app.js`) with this project.
2. Change to the `routes` directory and review the code and comments in the `bsProtocol.js` file.
3. Implement the resource end-points in the file to conform to the REST protocol as documented herein.
4. Review the code and comments in the `bsStrategy.js` file. This is an implementation of a completely random tile selection strategy. You should feel free to use this strategy or update this strategy with an alternative.
5. Modify the *server sent events* route class from the previous project such that the *protocol* module (this module) can send events to its client to update either the *ocean grid* or the *target grid*. This likely will require an update to the exported function from the *sse* module as well.
6. On your game board (client) implement the following:
   (a) Attach a listener to the *New Game* menu item that:
      i. Will prompt the user for:
         A. the name (filename) of a previously saved game state
         B. (optionally) a url of an opponent, including any port
         C. (optionally) a latency between 2000 and 5000
      ii. will form a request to the client's `/battle` resource end-point.
      iii. will disallow the placement or movement of any ships on the game board
      iv. disable the load and save game menu items
   (b) Attached a listener to the *Exit* menu item that:
      i. will request from the use to confirm termination of the current game
      ii. if the user terminates a current game, reset the game board
      iii. terminate the SSE - via closing the `EventSource` object
   (c) If the SSE messages a `WIN`, indicate to the user and reset the game board.

Figure 1: Programming Project Grading Rubric

Attribute (pts)	Exceptional (1)	Acceptable (0.8)	Amateur (0.7)	Unsatisfactory (0.6)
Specification (10)	The program works and meets all of the specifications.	The program works and produces correct results and displays them correctly. It also meets most of the other specifications.	The program produces correct results, but does not display them correctly.	The program produces incorrect results.
Readability (10)	The code is exceptionally well organized and very easy to follow.	The code is fairly easy to read.	The code is readable only by someone who knows what it is supposed to be doing.	The code is poorly organized and very difficult to read.
Reusability (10)	The code could be reused as a whole or each routine could be reused.	Most of the code could be reused in other programs.	Some parts of the code could be reused in other programs.	The code is not organized for reusability.
Documentation (10)	The documentation is well written and clearly explains what the code is accomplishing and how.	The documentation consists of embedded comments and some simple header documentation that is somewhat useful in understanding the code.	The documentation is simply comments embedded in the code with some simple header comments separating routines.	The documentation is simply comments embedded in the code and does not help the reader understand the code.
Efficiency (5)	The code is extremely efficient without sacrificing readability and understanding.	The code is fairly efficient without sacrificing readability and understanding.	The code is brute force and unnecessarily long.	The code is huge and appears to be patched together.
Delivery (total)	The program was delivered on-time.	The program was delivered within a week of the due date.	The program was delivered within 2-weeks of the due date.	The code was more than 2-weeks overdue.

The *delivery* attribute weights will be applied to the total score from the other attributes. That is, if a project scored 36 points total for the sum of *specification*, *readability*, *reusability*, *documentation* and *efficiency* attributes, but was turned in within 2-weeks of the due date, the project score would be $36 \cdot 0.7 = 25.2$.

Project Grading:

The project must compile without errors (ideally without warnings) and should not fault upon execution. All errors should be caught if thrown and handled in a rational manner. Grading will follow the *project grading rubric* shown in figure 1.

Collaboration Opportunities:

You may collaborate with up to one other person on this project - but you must cite, in the code (html, css, js) each students' contribution.