COMP3207/2324/Coursework/1 2

From NotesWiki
< COMP3207 | 2324
Jump to navigationJump to search

Contents

Specification

- Coursework 1 Part 2: NodeJS, Javascript and Google Cloud
- Deadline: Week 11
- Effort: 25 hours per student
- Weighting: 20% of module evaluation

The second part of the coursework involves the creation of the two clients (each interactive device, and a display device that everyone will see simultaneously), and the holding of the game state on a NodeJS server. Prompts and users will be called against the API created in the first part of the coursework. This part of the coursework will run the game aspect: People will submit the answers to the prompts, then score each others answers. Scoreboards will be displayed.

Template Application

You can clone the template application by running

```
git clone https://git.soton.ac.uk/comp3207/coursework1-2.git
```

This application works with Google App Engine and Heroku. You can use npm run gdeploy (Google) or npm run hdeploy (Heroku) to call the relevant scripts in package.json.

The template application provides a basic client and display, static files, Express, Socket.IO and a simple chat implementation for messages between clients and the server.

Key Concepts and Requirements

These are the functional requirements for your implementation:

- Players: People playing the game who join at the start. There should be a minimum of 3 players and a maximum of 8 players
- Audience: People who are above the maximum number of players, or those who join after the game has started
- First Player: The first player will be the admin and is responsible for starting the game or advancing the game forwards
- Rounds: There will be 3 rounds of prompts and answers, each worth more than the last
- Prompt collection: All players and audience members can submit their own prompts for other players. These get added to the API for use in later games and for use in the current game
    - A game should have 50% past prompts if available and 50% newly submitted prompts this game
- Answer submission: Players, but not audience members, are given 1-2 prompts (shared with one other player) to answer in a funny or amusing way
- Voting: For each prompt in the round, all players and audience members will see the prompt and the 2 submitted answers and vote for the funniest.
    - People should not be able to vote on their own answers
- Voting results: After each prompt is voted on, the players who submitted each answer and the number of votes each achieved is displayed
- Total scores: After the results for each prompt is displayed, the current scores for all players is displayed

These are the additional technical requirements for your implementation:

- A VueJS and Javascript powered client display (e.g. from a mobile) for player specific input, interactions and output
- A VueJS and Javascript powered master display (e.g. from a projector/screen) to show information to all players
- A NodeJS and socket.io powered server able to be deployed on Google App Engine and/or Heroku with socket.io to facilitate WebSocket communication
- It must be possible to extract your application and run it with npm start and deploy it with npm gdeploy / npm hdeploy
- **Important**: You must ensure you have followed the API specification in Part 1 exactly in terms of inputs and outputs - when we mark your submission, we will mark it against our own implementation of the API - if you have deviated from the API specification and we are unable to mark your coursework as a result, you may lose marks!

Suggested Implementation

Below is a suggested walkthrough and implementation guide for the coursework. Your implementation may differ, but you should ensure the above requirements and the Mark Scheme criteria are met.

Server Basics

For the first stage, we will need to get a NodeJS server running on Google App Engine to maintain the server state, using Websockets to communicate with connected clients.

- Download the template application and test running it both locally and remotely
- Update the server with state to hold connected clients, a list of players, a list of audience members

- Add functions, similar to the wrapper in the first part, to call the Azure functions with appropriate parameters
  - You will need to find an appropriate way to call your web-based function APIs from within NodeJS
- Deploy the server to the cloud

Interactive Client Basics

- Get to grips with the VueJS application (both client and display) provided in the template application
- Create clientside state holders in the client
  - We suggest at least a game state and player state and list of players
- Create functions to send messages to the server through socket.emi
  - Make use of the ability socket.emit to send both an event name and a payload
  - register: Register to the system with a username and password
  - login: Login and join the game with a username and password
  - prompt: Submit a new prompt
  - answer: Submit an answer to a prompt
  - vote: Submit a vote for an answer to the prompt
  - advance/next: Allow the person who started the game to advance the game to the next state
- Create a function to handle receiving a message and updating the local state variables
- Create a username and password prompt which can be used for signing up and logging in and is displayed when no game state is present
- Create a button to advance the game which sends the advance/next message
- Create a login and signup button which sends appropriate username and password data

Building State on the Server

Incoming Events

- Create appropriate socket.on listeners for the events expected to be sent from the client (register, login, prompt, answer, vote, next)
- Create initially empty (for now) handling functions to be called by those listeners (e.g. handleRegister, handleLogin, handlePrompt, handleAnswer, handleVote, handleNext)
- Create a function to handle errors and send back an error message

Logging in and Registering

- Complete the login and register functions by calling your API functions
  - On successful login and registration, update the state back to the client and add them as a player to the game
  - If more than 8 people join, they should be added to the audience
  - The audience can suggest prompts and vote on answers, but will not be given prompts to answer

Holding Game State

- Create variables to hold local state information
  - We suggest including a state number, a state object, a list of players, list of audience members, active prompts, answers received, votes received, the current prompt (when cycling through prompts for voting), round scores and total scores
  - You will also want to have a player state per player
- Use the game state variable to represent what's currently happening in the game

1. Joining: waiting for players
2. Prompts: players and audience suggest prompts (these will be then used as prompts combined with random prompts from the API)
3. Answers: players give answers to prompts (2 players to 1 prompt. Players will either get 1 prompt or 2 prompts in a round depending on number of players)

4. Voting: players and audience vote on answers to each prompt (cycle through all prompts in the round)
5. Results: voting results (show votes and points for this prompt)
6. Scores: total scores (tally up total overall scores)
7. Game Over: end of game (show final scores)

Relaying Game State

- Create a function to update a given player with the current game state and any player specific data (updatePlayer)
  - You will want to construct a JSON object and send it to that player via their socket
  - We suggest sending the game state, player state for that player and list of players
- Create a function to update all connections with the current game state (updateAll)
  - This can loop through all the players and call the above

Manipulating Game State

- Create initially empty functions (for now) to handle the start of each game state and the end of each game state
  - For example, startPrompts, endPrompts, startAnswers, endAnswers, startVotes, endVotes etc.
- Create a function to advance the game state and handle transitions and call the appropriate start and end of state functions
- Call the advance function when the next button is clicked

Building the game

- Update the end of players joining transition to start the game
  - Create a function to start a game which should initialise all players
- Update the function for receiving a prompt
  - This should store the prompt locally and advance the player state
- Update the function to handle the end of the prompts submission transition
  - This should store a local list of all prompts added so they can be used this game
  - Prompts should be submitted to the API for usage in later games
- Update the function to handle starting the answers transition
  - This should take 50% of the prompts from the API (if the API is not empty) and 50% from prompts entered by players/audience for this game
  - Assign prompts to players (prompts are shared between 2 players only. Players will get either 1 or 2 prompts in a round depending on number of players)
- Update the function to handle receiving an answer
  - If two prompts are required and one is complete, advance to the next prompt
  - If only one prompt is required or both are complete, advance the player state
- Update the functions to handle receiving a vote
  - Once a vote is cast, advance the player state
  - If all votes are cast, advance to the next prompt and answers
- Update the function to handle the end of voting
  - Set round scores (but do not yet update total scores) for players to be round_number * votes * 100
  - Advance to the next prompt to vote on answers if any remain in this round
  - Advance to scoring if no more prompts
- Update the function to handle end of round scores transition
  - Add round scores to player total scores
- Update the function to handle end of total scores transition
  - If no more rounds to come, set the game state to end of game

Make the interactive client work

In your Javascript interactive client, by reading the local state, create appropriate interfaces, templates, functions and buttons for:

- Waiting for players
- Suggest a prompt
  - Awaiting input
  - Prompt submitted
- Answer a prompt
  - Awaiting input
  - Answer submitted
  - In the audience (cannot answer)
- Voting
  - Choose an answer
  - Answer chosen
  - Unable to vote (if it's your own prompt)
- Waiting screen (when in any other state)

Make the display client work

The display client is what will be shown on the projector or main screen, separate to the individual clients, and shows a general state for all players to watch (like on Jackbox). The display client is found at /display. The display client should not need to be logged in.

In your Javascript display client, by reading the local state, create appropriate templates for

- Waiting for players
  - Showing the players that have joined so far, and provide the URL to join the game.
- Prompt suggestion
  - Showing how many prompts have been suggested and by who
- Answers being received
  - Showing which players have answered and who is being waited on
- Voting in progress (per prompt)
  - Show the prompt and the answers (but not who they were by)
- Voting results
  - Show the prompt, the answers, who wrote both answers, and the points the answers received
- Scores
  - Show the current status of the overall leaderboard
- Game over
  - Show the final state of the overall leaderboard

Testing

The functionality and correctness of your implementation will be validated by a test suite.

- The test suite Test Cases
- We will provide you with an implementation of the Part 1 API that you can use in your testing. It will have some additional features to help you with the testing:
  - Use a browser to access the tester homepage http://localhost:8181
  - Click on a specific row to configure the user and prompt data for the relevant tests

- Downloads for the tester can be found at https://git.soton.ac.uk/comp3207/coursework1-2/-/wikis/Backend-Test-server-executables

Verification of testing results

You will document how well your implementation passes the test cases by way of a report. We will take a random test category from the the suite and check your implementation against the test suite. If your reported results for this selected test category matches our findings then we will award you marks based on the rest of your test results. If your reported results do not match our findings then we will verify each test and you will forfeit 50% of the resulting marks due to an inaccurate report.

Mark Scheme

Component	Description	Marks
Server basics	Basic server build and deployment	10%
Test 1	Registration and Login	15%
Test 2	Game Start Up	30%
Test 3	Game round	30%
Test 4	9th player audience	5%
Test 5	Prompts	5%
Test 6	4 Players	5%

Submission

You are required to submit two files - a full zip of code that can be deployed and run directly, and an instructions file with any additional instructions or notes for your implementation.

- part2.zip: This should be a zip file that contains the root of your project in a way that can be deployed to Google App Engine
  - The structure should match that of the template application
  - This should include app.yaml, app.js, package.json, package-lock.json, public, views and any other additional files you have added
  - Do not include node_modules as it is not needed and will not be used and will be overwritten
  - It must be possible to run 'npm start' to execute your application
  - It must be possible to run 'npm run gdeploy' to deploy to Google
- A report
  - This should show how your submission handles the test cases outlined above. It will include any evidence that is required for the test cases.

Please note that we will be marking based on your submitted.zip file and not your cloud deployment, so please make sure what you submit is what you want marked!

Cloud Deployments

You can safely shut down both your Google and Microsoft cloud services, they are not needed for the marking of this coursework.

We will deploy your Google container ourselves for marking, so do not leave your Google application active to avoid accruing any charges.

We will mark against our own implementation of the API, so do not leave your Azure application or database active to avoid accruing any charges. However, this means you must have followed the Part 1 API specification exactly in terms of inputs and outputs.

FAQ and Clarifications

In this section we will collect Frequently Answered Questions and list any clarifications following requests from students. Those requiring an amendment in the spec will be signalled.

1. **When can prompts be added to the system? Is it just at the beginning?**
   Answer: Prompts can be added by players and audience members at any time whilst the game is running.
2. **How do I log in? I cant provide a body with the GET request?**
   Answer: We have extended the API to allow POST requests for `/player/login`, `/utils/leaderboard` and `/utils/get`. You can, however, make provide a body to a GET request as follows:

```
const request = require('request')

request.get('
http://localhost:8181/player/login', {
  json: true,
  body: {'username': 'aaaa', 'password':'aaaaaaaaaa'}
}, function(err, response, body) {
  console.log(response.statusCode);
  console.log(body);
})
```

```
Gives:
```

```
200
{ result: true, msg: 'OK' }
```

1.
3. **How do I refer to the backend API in my code?**
   Answer: In the javascript, you can use the variable `BACKEND_ENDPOINT` to refer to the api see app.js line 25]app.js line 25 (https://git.soton.ac.uk/comp3207/coursework1-2/-/blob/main/app.js?ref_type=heads#L25). Set the `BACKEND` environmental variable in npm if you want to specify your cloud API.
4. **I'm using the axios library but having problems sending a get request to login**
   Answer: You have to do either:

```
axios.get(BACKEND_ENDPOINT + "/player/login",
  data: {username: "a", password: "b"}
}).then(response => {
  console.log(response.data);
});
```

```
or
```

```
axios.request({url: BACKEND_ENDPOINT + "/player/login",
  method: "GET",
  data: {username: "a", password: "b"}
}).then(response => {
  console.log(response.data);
});
```

```
Note the second parameter to axios.get is an options object not a body.
```