COMP 2406A
Winter 2020 – Tutorial #5

Objectives

- Add a template engine to an existing HTTP server
- Create templates capable of accepting data and producing corresponding HTML

Problem Background

Download the server.js and cards.json files from the course cuLearn page. The server.js file contains a template for an HTTP server that currently only responds with 404 errors. The card dataset used by the server is loaded from the cards.json file. Each card object in the dataset includes the following properties:

- id: a string that uniquely identifies the card
- name: the name of the card
- cardClass: a string indicating the player class that can use the card
- set: a string indicating the collectible set the card is from
- type: a string indicating the type of the card
- artist: a string with the name(s) of the artist(s) who created the art for the card
- text: a string representing the text shown on the card's face

In addition to these properties, some cards may also have:

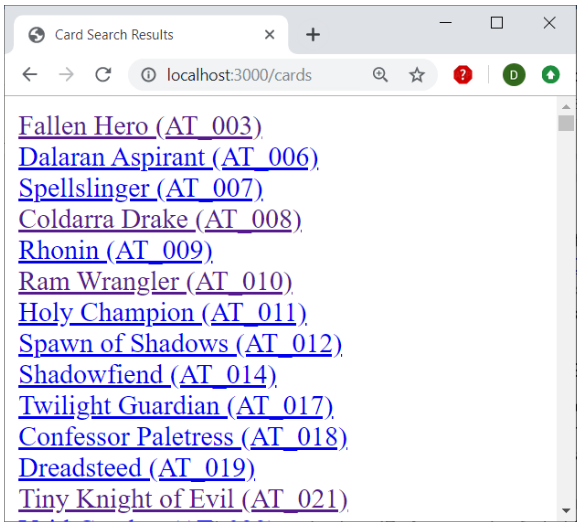
- rarity: a string representing how rare the card is
- mechanics: an array of strings indicating special rules for the card

The goal of the server will be to define route handlers for GET requests to /cards and /cards/*someCardID*, where *someCardID* is a parameter representing a unique card ID. The responses the server sends should be generated using a template engine of your choosing.

Problem 1 (Adding Support for /cards)

To start, add a template engine of your choosing to the server. As discussed in lecture, this will involve installing the template engine using NPM (e.g., `npm install pug`) and requiring that module in your server code (e.g., `const pug = require("pug")`). Add a route handler to the server that will respond to GET requests for the /cards route.

Create an HTML template which takes in an array of card objects and uses the data contained within to generate a page to display the list of cards. As there are so many cards in the dataset, it may be beneficial to only return a subset of all cards (e.g., the first 25). For each card in this list, you should have a link that points to the URL for that specific card (e.g., http://localhost:3000/cards/that_card's_id) and the link text should indicate the card's name and unique ID. Here is an example of what the page may look like:



Fallen Hero (AT_003)
Dalaran Aspirant (AT_006)
Spellslinger (AT_007)
Coldarra Drake (AT_008)
Rhonin (AT_009)
Ram Wrangler (AT_010)
Holy Champion (AT_011)
Spawn of Shadows (AT_012)
Shadowfiend (AT_014)
Twilight Guardian (AT_017)
Confessor Paletress (AT_018)
Dreadsteed (AT_019)
Tiny Knight of Evil (AT_021)

Problem 2 (Adding Support for /cards/*someCardID*)

Now that your server can generate a list of cards with links to the individual card pages, we need to add support for the individual card routes. The general pattern for this route will be */cards/someCardID*, where *someCardID* represents the unique ID of the card being requested. To implement this handler, you will have to perform some string operations to extract the unique ID from the request URL (do this first before implementing the rest of the problem).

Once you have extracted the card ID successfully, you will have to find a matching card (if it exists) within the `cards` object. As the cards are stored in an object containing card IDs as properties, it is relatively easy to look up a card given the ID (look up the Object.hasOwnProperty method if you are unsure). This is an advantage of storing data in a Javascript object – with a simple array, you would have to perform linear search to try and find a matching ID. If a card with the requested ID does not exist, you may respond with a 404 error. If the card does exist, then you should send a response containing the page for that card.

To render the page for a card, implement another template file that will accept a card object as input and produce HTML containing a title indicating the card’s name and unique ID. The page’s content should also include, at minimum, the card’s name, ID, class, set, type, artist, and text attributes.

In addition to this basic content, implement a way to visualize the card’s rarity, if it has a rarity property. This could involve displaying a number of * characters to signify the rarity (e.g., 1 star for free, 2 for common, 3 for rare, 4 for epic, 5 for legendary) or modifying the color of the card name on the page. A list of the possible rarity values is included at the end of this document.

If the card has a mechanics array property, your template should include a “Mechanics:” heading and list each of the mechanics strings contained in the array.

Problem 3 (Adding a Query Parameter)

Currently, when a GET request is made to the server for /cards, your server responds with all the cards (or the first 25, as suggested in Problem 1). Ideally, the user will be able to search for specific types of cards using query parameters. To start, add a query parameter that allows the user to specify text that should be found in the name of the card. The URL for this request can be expected to look like this:

http://localhost:3000/cards?name=*SomeName*

To implement this, you will have to perform further processing of the request URL. Add another route handler to your server that handles GET requests that *start with* "/cards?". For now, you can assume that the only text after the ? in the URL will contain name=*SomeName*, where *SomeName* is a name value they want to search for. Perform more string operations (the split method may be useful) to extract the name supplied within the URL. Now, instead of responding with all cards, your server can send back only cards that contain the given search parameter.

To determine the set of cards, you will have to go through the array and find matching cards. This may be a good time to use the array.filter method we used earlier in the course. You can combine this filter method with a Boolean expression for checking if the given search name is contained in the name of the card. Once you have found the matching cards, responding with the correct HTML is just a matter of re-using the same template you used for Problem 1. In this case, though, you supply only the matching cards as the data instead of all the cards.

If you are looking for a greater challenge, consider adding support for more query parameters. You may match cards based on class, type, rarity, or any other property the cards have. Writing code to manually find cards that match multiple query parameters can start to get difficult. Later in the course, we will discuss databases and see some convenient ways to do this. We will also look at the Express module soon, which will provide a number of handy functionality for automatically parsing query parameters and other parts of HTTP requests.

Valid ‘class’ values, with their frequency in brackets:

MAGE (61), HUNTER (63), PRIEST (59), NEUTRAL (713), WARLOCK (70), ROGUE (66), DRUID (70), SHAMAN (60), WARRIOR (60), PALADIN (56)

Valid ‘type’ values, with their frequency in brackets:

MINION (1277), WEAPON (1)

Valid ‘set’ values, with their frequency in brackets:

TGT (96), BOOMSDAY (102), BRM (23), GANGS (101), CORE (50), EXPERT1 (150), HOF (9), NAXX (25), GILNEAS (140), GVG (92), ICECROWN (94), KARA (52), LOE (36), LOOTAPALOOZA (102), OG (100), UNGORO (106)

Rarity values, from most common to most rare, with their frequency in brackets

FREE (50), COMMON (432), RARE (309), EPIC (169), LEGENDARY (218)
Additionally, 100 cards have no defined rarity value