Homework #4

# Kaomoji Texting

## Overview

### Repository Creation

👀 Click on this link: https://classroom.github.com/a/bDXX1A23 (https://classroom.github.com/a/bDXX1A23) to accept this assignment in GitHub classroom.

- This will create your homework repository
- Clone it once you've created it.

### Description

Create a site that takes a text message and replaces the words representing emotions with kaomoji!
(ʘ‿ʘ)
E.g. "Today's weather is nice and it makes me happy" -> "Today's weather is nice and it makes me ♥(^‿^ゝ)"

> Kaomoji is a Japanese version of emoticons (such as `:)` for a smiling face). They tend to be more expressive than traditional emoticons, hence more fun to play with.

Additionally, users will be able to manage the mapping of words with kaomoji through the manipulation of an in-memory data store.

In this homework you'll be working with:

- serving static files
- middleware
- handling and creating forms: GET and POST
- storing and managing data in-memory

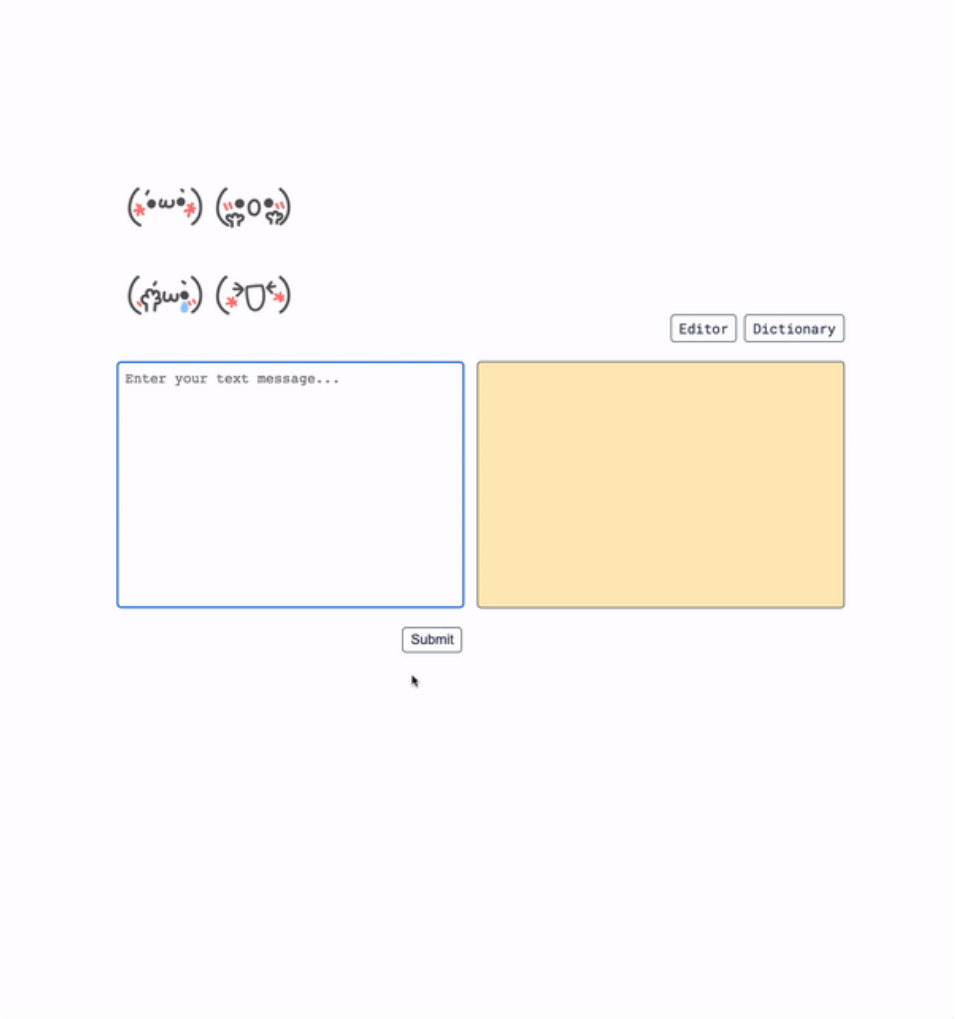
You'll be creating the following pages:

- **root** - `/` : root path that redirects to the `/editor` page.
- **editor** - `/editor` : editor page with text editor that adds a corresponding kaomoji to the text.
- **dictionary** - `/dictionary` : page that shows the list of all available kaomojis and their corresponding emotions.

Your directory layout should look like the following **once you're done with the assignment**:

```
├── app.mjs
├── code-samples
│   └── kaomojiData.json
├── kaomoji.mjs
├── package-lock.json
├── package.json
├── public
│   ├── css
│   │   └── main.css
│   └── img
│       └── kaomoji.png
└── views
    ├── editor.hbs
    ├── layout.hbs
    ├── list.hbs
    └── any additional hbs files you'd like to create
```

## Example Interaction




## Submission Process

Submit **homework through gradescope** along with GitHub:

- gradescope can be accessed through Brightspace
- gradescope will only accept submissions via GitHub

You will be given access to a private repository on GitHub, and it will contain some starter code.

1. stub source files in the `src` directory (`web-lib.mjs`)
2. some images you can use for testing in the `public/img` folder as well as a markdown file in `public/markdown`
3. a `private` folder to test configurable root directory
4. a sample `json` configuration file that specifies settings for your web server
5. you'll have to create your own `package.json`, `package-lock.json`, `.gitignore` __
    - (these are required and part of grading)
    - use `npm init` to create `package.json` (you can just press enter all the way through)
    - remember to put `node_modules` in your `.gitignore` so that it is not included in your repository
6. if `.eslintrc.cjs` isn't present, you can copy over your linting configuration `.eslintrc.cjs` from previous assignments * as usual, you'll have to clean up any eslint warnings or modify eslint config to match your coding style
    - remember to lint only your source code `*.mjs`

**Push** your code to the homework repository on GitHub so that your latest code is present.

Submit your repo through gradescope. The gradescope assignment will close, so make sure you submit before the deadline.

Note that if you make changes to your repo after you submitted through gradescope, you must resubmit through gradescope (gradescope does not sync changes).

### Make at Least 4 Commits

- Commit multiple times throughout your development process.
- Make at least 3 separate commits - (for example, one option may be to make one commit per part in the homework).

# Part 1 - Setup

### Installing Dependencies

- create a `package.json` (a `package-lock.json` should be created for you as well once you start installing modules)
- **install** the following **dependencies** (make sure you use the `--save` option):
    - `express`
    - `hbs`
    - `nodemon` (optional, to help you make changes and run them more easily)

### .gitignore

- create a `.gitignore`
- ignore the following files:
    - `node_modules`
    - any other files that aren't relevant to the project… for example
        - `.DS_Store` if you're on OSX
        - etc.

### linting

- an eslint configuration file (for example `.eslintrc.cjs`) should be in the root directory (or copy one from a previous project if it doesn't exist)
- make sure that any linting tools are installed (`eslint`)

- periodically lint your program as you work
- minor deductions (1 point) will be taken off for each **class** of error, w/ a maximum of 5 or 6 points total

# Part 2 - Homepage and Static Files

## Enabling Static Files

First, we would like to make sure that we can serve static content on our site - like css and images. So let's get started.

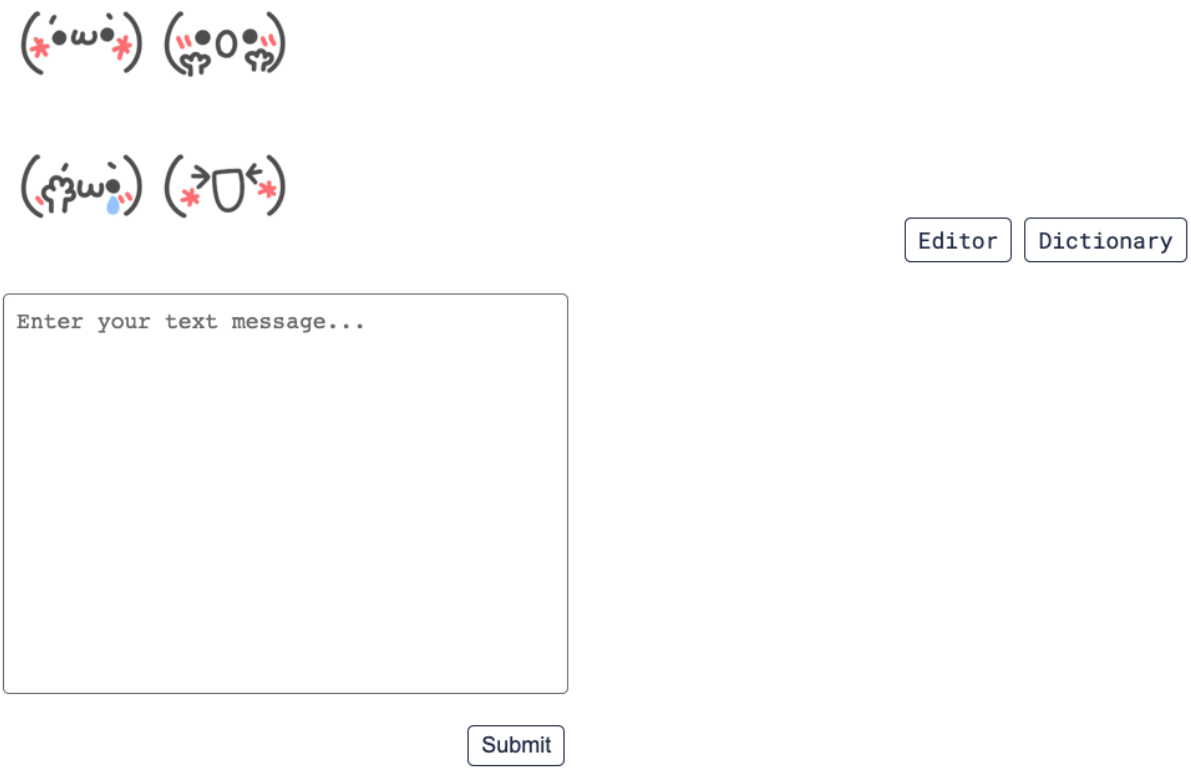
- create the following directory / folder structure in your project's root directory
  - `public`
  - `public/css`
  - `public/img`
- add a blank css file in `public/css/main.css`
- add an image that has something to do with emoji or texting in `public/img/logo.png` *tip: check the resources like flaticon.com (flaticon.com) for some free visuals and don't forget to mention the author!
- create a basic express skeleton application in `app.mjs`
  - make sure that your application is **served over port 3000**
  - **after calling `app.listen(3000)`**
    - print out "Server started; type CTRL+C to shut down " to the console (the terminal window)
    - this will give you feedback that your server has started correctly
- just add the appropriate `import` statements (express, path, etc.) and middleware to enable static file serving:
  - refer to class slides for reminders / snippets on how to do this
- test that both the css files and image work
  - for example, try to go to `http://localhost:3000/img/logo.png` in your browser
  - -> your image should be displayed in browser

## Creating the Editor Page

- for the main page, your app should accept `GET` requests on the path `/`
- we will be using handlebars as our templating engine, so do the required steps to set that up
  - get all the requirements and config setup
  - create the appropriate views folder, along with an initial layout file:
    - `views`
    - `views/layout.hbs`
- in your `layout.hbs` , add the html
- recall that this surrounding html will go on every page
  - add a reference to include your `main.css` stylesheet (so that each other `.hbs` page has it too)
  - include a header containing both your logo image and/or the title of your site, they should lead to the main page on click
  - additionally, add a 'nav bar' consisting of two links that will let you navigate across the 2 pages in your site:
    - a link to the home / main page that is an editor page ( `/editor` ) or root url `/` that redirects to the editor page
    - a link to **dictionary** page ( `/dictionary` ) that lists out all the available kaomoji
  - Important! - don't forget `body` , surrounded by triple curly braces, or else other templates are not going to get rendered
- now that you have the layout, add your editor page template

    - Create a form that has a text fiekd and a submit button.
    - you will add more html and templating to this file later
- set up a route and a `render` call so that going to the root url via a `GET` request it would redirect to `/editor`
- add some css to change the styles on the page. Some examples / inspiration:
    - change the font see the mdn article on fonts (https://developer.mozilla.org/en-US/docs/Web/CSS/font) and font-family (https://developer.mozilla.org/en-US/docs/Web/CSS/font-family) (if you'd like a larger variety of fonts, you can use Google Fonts (https://developers.google.com/fonts/docs/getting_started))
    - change the background color
    - change the font color
    - add padding, alignment
    - keep the design minimalistic
    - etc.

Here's an example of what the page could look like (you don't have to use the same exact styles, but add enough styles so that you can see that the style sheet is correctly served and applied to the html):




If you add stuff to your css file but no styles change, check the console for errors and the html in `layout.hbs` to make sure the path to the css file is correct.

# Part 3 - Middleware and Logging

In order to help us debug our code on backend during development, let's set up some initial logging in form of middleware. We want to log **for each request**:

- The method (GET, POST)
- The route
- The query string

First, activate the body parsing middleware (`express.urlencoded`) by passing it to `app.use`; this will allow you to access the content of the request's body.

Additionally, create a custom middleware function and `app.use` it in `app.mjs`. Hint: don't forget to call `next()`.

Again, this function **has to** print

- the request's **method**, **path**, and request's **query string**
- Example:
  - Method: GET
  - Path: /
  - {}

# Part 4 - Adding a Kaomoji Database

We will store our kaomojis in-memory, and display them in a list format later. Each kaomoji has two components:

1. the kaomoji
2. the emotions associated with it

## Kaomoji class

In `kaomoji.mjs`, create a class that represents a kaomoji. Your class should behave as follows:

```
const kao = new Kaomoji("ಠ_ಠ", ["wat", "what", "confused"]);

console.log(kao.value); // ಠ_ಠ
console.log(kao.emotions); // ["wat", "what", "confused"]
console.log(kao.isEmotion("happy")); // false
console.log(kao.isEmotion("Confused")); // true
```

Export the class to use it in `app.mjs`.

## Initial data

The initial data for the site should be stored as an `Array` of `Kaomoji` instances. It can be a global variable in your `app.mjs` file.

The `Kaomoji`s should be created by reading the `kaomojiData.json` file in the `code-samples` directory. Do this by using `fs.readFile`. ⚠️ Do not use the synchronous (the ones that end in "sync") version of these functions! (we're doing this to practice callbacks and handling async operations…).

The general requirements for this part are:

- read the code from the `json` file and for each object in the file create the corresponding `Kaomoji` object

- save the `Kaomoji` objects in a global variable in `app.mjs`
- once everything is read, print out the global variable of kaomojis :

```
[
  Kaomoji {
    value: '(ಥ﹏ಥ)',
    emotions: [ 'angry', 'mad' ]
  },
  Kaomoji { value: '( ゜Д゜)<!!', emotions: [ 'frustrated' ] },
  ... // other Kaomoji objects follow
]
```

To ensure that server only starts **after** the file is read in, you can call `listen` on your app object from within the callback to `fs.readFile`.

## Page to display the data

- pass in your kaomoji data (in an array of objects) to the `render` call in the route handler for the `/dictionary` so that our template has access to it
- add html to display the kaomoji data in an organized way
    - HINT: can iterate through the list of snippets using the `#each` helper
    - HINT: can put each snippet & it's data (value, emotions) in a list item (`li`) or a table idem `th` and style it
        - nested each statements are possible
        - (so, for example, you can loop through each kaomoji…and then loop through an individual kaomoji's emotion tag)

**Reload and visually check.**

- Page `/dictionary` should display the list of kaomoji that you printed out earlier after reading the `json` file.
- Styling does not have to be exact!

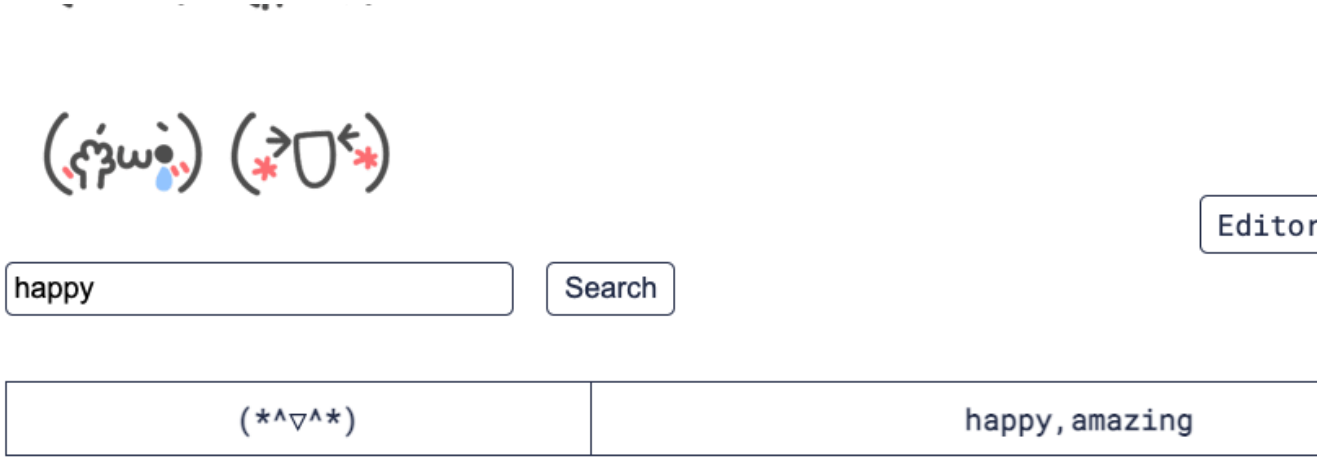
Editor | Dictionary

| | |
|---|---|
| (ಠ益ಠ) | angry, mad |
| ( ˙Д˙ )<!! | frustrated |
| ヽ(≧Д≦)ノ | upset, infuriated |
| ヾ(@^▽^@)ノ | excited |
| (⊙_⊙;) | puzzled, lost |
| ( ┐ ˙•ꈊ•)┐ | dorky, goofy, funny |
| (。⊙ʁ ⊙。) | insane, weird |
| へ(。□°)へ | crazy |
| (*≧∀≦*) | thrilled, elevated |
| ╰(✧∀✧╰) | hyped |
| (*^▽^*) | happy, amazing |
| v(▢)v | hungry |
| ¯\( ◔ ʁ◔)/¯ | indifferent |
| (♡°▽°♡) | love |
| ¯\(ツ)/¯ | meh, dunno, shrug, clueless |

# Part 5 — Search kaomojis

Now that we have a list of kaomojis, it would be nice to be able to search for the right kaomoji by typing in the emotion.

- **In the same** template as the `/dictionary` page, add a form to search for kaomojis.
  - submitting the form makes a request to the same path that the form page is on
  - the form uses the appropriate HTTP request method for the functionality of this form (it should be `GET` since it's just reading data!)
- The form should have one `input` element for entering form data with a `name` attribute `emotion`; use the appropriate `type` attribute for entering in the search term
- Add a bit of styling to match the overall theme.




**Now, onto the server side:**

Modify your route handler for your dictionary page (e.g. `/dictionary`) to **only** select kaomojis that match the keyword and send it to the `render` call.

And your console should look something like this (for the case where tag and text are specified):

```
Method: GET
Path: /dictionary
{ emotion: 'happy' }
```

# Part 6 - Adding Data

We should be able to add kaomoji to the dictionary! (Note, this *does not have to be written* back to the original file). For some initial setup, we'll need to use body parsing middleware that comes with express:

- see the slides for using the express body parsing middleware (../slides/10/forms.html#/6)
- add the code in the slides to activate the body parsing middleware (`express.urlencoded`)
- (no need to install a separate module)
- this takes the body of an incoming http request that is in the format of `name=val` (form urlencoded) and adds it to a property on the request object called `req.body`

For example, this request:

```
POST /foo HTTP/1.1
Content-Type: application/x-www-form-urlencoded

bar=baz&qux=corge
```

...results in `req.body` containing:

```
{bar: baz, qux: corge}
```

Now we're ready to add another form!

- **In the same** template as the `/dictionary` page, add a form to add kaomoji.
  - submitting the form makes a request to the same path that the form page is on, but you'll use a different HTTP Request method
  - the form uses the appropriate HTTP request method for the functionality of this form (it should be `POST` since it's modifying data!)
- The form should have the following `input` elements:
  - an element with `name`, `emotions`, which should allow for the entry of a text, such as `mad,angry`

- an element with `name`, `value`, which should allow for entry of kaomoji

Process the form submission

- When the user submits the form, a new `POST` request will be sent to `/dictionary`
- Add a route handler to `app.mjs` to respond to this kind of request
- In the route handler, use `req.body` to access the data the user submitted in the form
- You can use this to add a new kaomoji object!
- The response to the client should be the page showing all kaomoji
- Send back the response in such a way where refreshing the page after form submission **will not** cause a duplicate item to be added

Example interaction:

| | |
|---|---|
| (๑-﹏-๑) | sickness |
| ( ∩◡∩') | smiley,nice |
| .･ﾟﾟ･(>▂<)･ﾟﾟ･. | sobbing |
| (･□･;) | surprised,astonished |
| (•ε ๑•,) | intrigued |

(//>/▽/</) | embarrassed,shy | Add

results in

| | |
|---|---|
| .･ﾟﾟ･(>▂<)･ﾟﾟ･. | sobbing |
| (･□･;) | surprised,astonished |
| (•ε ๑•,) | intrigued |
| (/ />/ ▽ /</ /) | embarrassed,shy |

Enter the value... | Enter emotions... | Add

# Part 7 - Implementing the editor

Now let's implement the main feature — the editor that takes any text message and replaces the words representing emotions there with kaomoji.

- In the homepage `/` modify the form you already have, the same way as you did for the search in the `/dictionary` page.
- Next to the form add another area to display the edited text.
- You may want to use a `textarea` field (https://developer.mozilla.org/en-US/docs/Web/HTML/Element/textarea) rather than an `input` field within your form
- The styles **do not have to match**, but if you'd like to try to implement the design in the images, you can:
  - use `inline flow-root` or `inline-block` (inline flow-root)
  - dive into `flex` / flexbox (https://css-tricks.com/snippets/css/a-guide-to-flexbox/).

**On the server side:**

Create the function that would:

- take the message as an argument and break it down into an `Array` of words (okay to assume space as word delimiter, no need to handle punctuation)
- iterate through each word and replace it with a kaomoji if it finds the corresponding emotion in the dictionary
- returns the message
- tip: remember the case sensitivity!

For this assignment, you can leave the punctuation, it is fine if the editor returns just a pure text with empty spaces only!

Editor Dictionary

i am excited

i am ヾ(@^▽^@)ノ

Submit

```
</div>
```