Project 01: Server-side API Design Documentation

Sam Lindsey and Adam Westerholm

"Describe each of the schemas that you created for Mongoose, and why you chose the types and validation requirements you did for each of the schema's attributes"

Schemas

- **User schema**
  - **fullName**- This attribute is a *subdocument*. It consists of a nested schema which requires a **firstName (*String*)** and a **lastName (*String*)**, both of which are required when inputting the entire fullName. A **middleName (*String)*** can also be put into **fullName,** but is not required. **(Additional field exceeds basic requirements)**
  - **username**- this attribute is a type *String,* is unique and is required. The userName is unique so it can only be associated with a single user. Any reviews created link back to this unique userName so they can have attribution for the review they create.
  - **email-** this attribute is a type *String.* Like username, it is also unique and is also required when creating the entire user. Each email should be unique so that multiple emails cannot be associated with a single user.
- **Review schema**
  - **review_desc**- this attribute is of type *String.* It is a description of the review and is required when writing a review.
  - **review_rating-** this attribute is of type *Number.* It is a rating that must be between 1 and 5 and must be a whole number. **(This restriction exceeds basic project requirements)**
  - **date-** this attribute is of type *Date.* This is a default value that does not need to be specified when creating a review as the value will automatically default to Date.now which will print the date and time. This is a required value (to keep track of when the review was created)
  - **by_user-** this attribute is of type *Schema.Types.ObjectId.* This attribute references the "users" collection which contains all the user information. This specifically refers to the object id linked to that user. It is required when creating a review.
  - **Reviews *must* be created by specifying a recipe it belongs to.**
- **Recipe schema**
  - **name-** this attribute is of type *String* and is required for a recipe to be created. This is the name of the recipe.
  - **description-** this attribute is of type *String* and is required for a recipe to be created. This is the description of the recipe to be created.

- **img_url-** this attribute is of type *String* and is required for a recipe to be created. In the future, this link will refer to a picture of the recipe
- **prep_time-** this attribute is of type *Number* and is required for a recipe to be created. Assume that this time is in a given number of minutes to standardize input.
- **cook_time-** this attribute is of type *Number* and is required for a recipe to be created. Again, assume that this time is in a given number of minutes for standardization.
- **directions-** this attribute is of type [*String*] and is required for a recipe to be created. This array will list out specific directions for a recipe to be followed.
- **ingredients-** this attribute is a *subdocument.* It consists of an array of ingredients, which is a nested schema with one attribute, *ingredientName,* of type *String*; and this array is a required field. The subdocument is implemented so each ingredient has its own uniquely generated _id.
- **user_reviews-** this attribute is initialized as an empty array and automatically updated as reviews are made. It is of type [*Schema.Types.ObjectId*] where the objectIDs reference a Review JSON object in the "reviews" collection, for the respective recipe. This field is required, but cannot be generated or specified by the user upon creation of a recipe.

"Describe your RESTful API, including what routes you created, what HTTP methods are available for each route, what payloads are required and/or returned for each route, and what response codes can be returned for each route"

<u>Routes</u>

- **/api/users**
    - Uses *users.router*
    - To use the five HTTP methods for users correctly. This requires /api/users to be put before any call of get, post, put or delete for users.
    - **HTTP Methods:**
        - **GET /api/users/** (index)
            - ***Payloads required:*** none
            - ***Payloads returned:***
                - **(200 Okay)** Will return all of the users as JSON with their respective data - fullName, username and email
                - **(400 Bad Request)** in a catch block for any other errors, sends the error to the console
        - **GET /api/users/:id (*show)***
            - ***Payloads required:*** none
            - ***Payloads returned:***

                - **(200 Okay)** Will return the user specified by the given id as a JSON object with its respective data - fullName, username and email - if successful
                - **(400 Bad Request) {**message: “Not found”} if user id not found
                - **(400 Bad Request)** in a catch block for any other errors, sends the error to the console
        - **POST /api/users/** (create)
            - ***Payloads required:*** A User JSON object with fullName, username and email
            - ***Payloads returned:***
                - **(201 Created)** The created User JSON object
                - **(400 Bad Request)** in a catch block for any other errors, sends the error to the console
        - **PUT /api/users/:id** (update)
            - ***Payloads required:*** A user JSON object with fullName, username and email
            - ***Payloads returned:***
                - **(200 Okay)** No Data, if successful
                - **(404 Not Found) {**message: “Not found”} if the :id does not match any users
                - **(400 Bad Request)** in a catch block for any other errors, sends the error to the console
        - **DELETE /api/users/:id** (destroy)
            - ***Payloads required:*** none
            - ***Payloads returned:***
                - **(204 No Content)** No Data, if successful
                - **(404 Not Found) {**message: “Not found”} if the user id not found
                - **(400 Bad Request)** in a catch block for any other errors, sends the error to the console

- **/api/recipes**
    - Uses *recipes.router*
    - To be used to correctly send the five HTTP methods for recipes. This requires /api/recipes to be put before any call of get, post, put or delete for recipes.
    - HTTP Methods:
        - **GET /api/recipes/** (index)

        - ***Payloads required:*** none
        - ***Payloads returned:***
            - **(200 Okay)** An array of all JSON objects in “recipes” collection, if successful
            - **(400 Bad Request)** in a catch block for any other errors, sends the error to the console
    - **GET /api/recipes/:recipeID** (show)
        - ***Payloads required:*** none
        - ***Payloads returned:***
            - **(200 OK)** The Recipe JSON object that matches the given :recipeID, if successful
            - **(404 Not Found)** {message: “404 Not Found”} if the :recipeID doesn’t match any recipes
            - **(400 Bad Request)** In a catch block for any other errors, sends the error to the console
    - **POST /api/recipes/** (create)
        - ***Payloads required:*** A Recipe JSON object, without any user_reviews
            - user_reviews will default to an empty array; reviews cannot be created or added to the array at the same time as a recipe is created
        - ***Payloads returned:***
            - **(201 Created)** The created recipe JSON object, if successful
            - **(400 Bad Request)** In a catch block for any other errors, sends the error to the console
    - **PUT /api/recipes/:recipeID** (update)
        - Reviews cannot be updated from here. All other fields can be changed, but review updates must be routed through *reviews.router*. This organization is easier to understand, and will result in less unexpected behavior
        - ***Payloads required:*** A Recipe JSON object
        - ***Payloads returned:***
            - **(200 OK)** No Data, if successful
            - **(404 Not Found) {**message: “Not found”} if the :recipeID does not match any recipes
            - **(400 Bad Request)** in a catch block for any other errors, sends the error to the console
    - **DELETE /api/recipes/:recipeID** (destroy)

                - This not only removes the recipe from “recipes” collection, but also finds all of the reviews in the recipe’s user_reviews array and removes them from the “reviews” collection
                    - This is implemented so that there are no obsolete reviews, and to ensure that all reviews refer to an existing recipe
                - ***Payloads required:*** none
                - ***Payloads returned:***
                    - **(204 No Content)** No Data, if successful
                    - **(404 Not Found)** {message: “Not Found”} if :recipeID does not match to an existing Recipe
                    - **(400 Bad Request)** In a catch block for any other errors, sends the error to the console

- **/api/reviews**
    - Uses *reviews.router*
    - One of the acceptable routes to correctly send four of the five HTTP methods for reviews. This required /api/reviews to be put before any call of get, post, put or delete for reviews. This route is specifically looking at the reviews themselves and does not require the request to first query a recipe.
    - HTTP Methods:
        - **GET /api/reviews/** (index)
            - ***Payloads required:*** none
            - ***Payloads returned:***
                - **(200 OK)** An array of all the Review JSON objects in the “reviews” collection, if successful
                - **(400 Bad Request)** In a catch block for any other errors, sends the error to the console
        - **GET /api/reviews/:reviewID** (show)
            - ***Payloads required:*** none
            - ***Payloads returned:***
                - **(200 OK)** The Review JSON object that matches the given :reviewID, if successful
                - **(404 Not Found)** {message: “404 Not Found”} if the :reviewID doesn’t match any reviews
                - **(400 Bad Request)** In a catch block for any other errors, sends the error to the console
        - **PUT /api/reviews/:reviewID** (update)

            - Updates a Review with given data
            - Ignores **date (Date)** and **by_user (ObjectID)**. There is no reason for a reviews date to be changed, since **date** refers to its creation, and the _id of the **by_user** can never be altered
            - ***Payloads required:*** A Review JSON object
            - ***Payloads returned:***
        - **DELETE /api/reviews/:reviewID** (destroy)
            - ***Payloads required:*** none
            - ***Payloads returned:***
                - **(204 No Content)** No data, if successful
                - **(404 Not Found) {**message: “Not found”} if the user id not found
                - **400 Bad Request** in a catch block for any other errors, sends the error to the console

- **/api/recipes/:recipeID/reviews/**
    - Also uses *reviews.router*
    - This route functions similar to the **/api/reviews/** route, in that it calls the same functions in the same fashion (both routes use the same controller). However, this route is *required* to correctly send an HTTP POST request for a Review.
    - This is implemented because a review must refer to a recipe that exists, so the parameter **recipeID** is required in this call to 1) check existence of the recipe and 2) to find and update the recipe object when the review is created
    - HTTP Methods:
        - **POST /api/recipes/:recipeID/reviews/** (create)
            - ***Payloads required:*** A Review JSON object with description, rating, and the _id of the User who wrote the review
            - ***Payloads returned:***
                - **(201 Created)** The created Review JSON object with date, _id for the Review, and __v fields added
                - **(400 Bad Request)** {message: “Rating must be a whole number”} for invalid review_rating ***(Exceeds Basic Requirements)***

        - **(400 Bad Request)** {message: “Rating must be between 1-5”} for invalid review_rating ***(Exceeds Basis Requirements)***
        - **(400 Bad Request)** {message: “Review does not exist} if recipeID is not found
        - **(400 Bad Request)** in a catch block for any other errors, sends the error to the console
    - **And all of the other methods described in /api/reviews/**