# Cumulative Project: The Scoop

**In this project, you'll build a web app for users to post interesting articles, comment, and upvote/downvote them.**

# The Scoop

## Project Overview

In this project, you will build all of the routing and database logic for an article submission web app called The Scoop.

The Scoop allows users to:

- Create and log in to custom username handles
- Submit, edit, and delete articles containing a link and description
- Upvote and downvote articles
- Create, edit, and delete comments on articles
- Upvote and downvote comments
- View all of a user's articles and comments

You can view all of this functionality in action in the video below:

The current implementation of The Scoop contains all of the server logic for users and articles, but does not have the necessary database and route logic for comments. You can see all of the current server logic in **server.js**. The two main objects to pay attention to are `database` and `routes`.

- `database` is a JavaScript object containg all of the stored model instances. The `users` and `articles` properties contain JavaScript objects which will contain all of your created users and articles. Each of these objects will have keys representing the ID of the saved resource and values of the whole resource. For example if we created an article with ID `1`, the `articles` object would look like this `{1: {id: 1, url: 'article url', ...}}`. Additionally the `database` object contains a property for keeping track of the ID of the next article to create (to ensure no two articles have the same ID)
- `routes` is a JavaScript object containing all of the routes needed for The Scoop. The keys of this object are the path of the request, and the values are objects containing all of the possible HTTP verbs for that

path. Each of those verbs map to a function to call if that route is hit and will be passed the path of the request and the request object (containing the request body)

It is important to take time at the beginning of this project reading through the previously written database and route logic to ensure you understand how all of these pieces work together.

## How To Begin

To start, download the starting code for this project here. After downloading the zip folder, double click it to uncompress it and access the contents of this project.

To start your server, run `npm install` and then `node server.js` from the root directory of this project. Every time you change **server.js**, you will have to restart your server before the changes will take effect. To do this press "control + c" in the bash terminal where your server is running (or close the terminal) to shut it down and then re-run `node server.js` to start it again. While your server is running, you will not be able to run commands in the bash terminal, so open a new terminal if you want to run other commands.

To view your local version of the site, open **index.html** in Google Chrome.

## Implementation Details

To complete this project, you will need to add the database information and server routes for Comments. Below we will list the high-level information about the Comment data model and the expected functionality of each Comment route. Additional implementation information, such as edge cases, can be discovered by running the test suite (discussed later).

Database Properties

- **comments** - an object with keys of IDs and values of the corresponding comments
  - id - Number, unique to each comment
  - body - String
  - username - String, the username of the author
  - articleId - Number, the ID of the article the comment belongs to
  - upvotedBy - Array of usernames, corresponding to users who upvoted the comment
  - downvotedBy - Array of usernames, corresponding to users who downvoted the comment

- **nextCommentId** - a number representing the ID of the next comment to create (to ensure all comments have unique IDs), initializes to `1`

Route Paths and Functionality

**/comments**

- POST
  - Receives comment information from `comment` property of request body
  - Creates new comment and adds it to database, returns a 201 response with comment on `comment` property of response body
  - If body isn't supplied, user with supplied username doesn't exist, or article with supplied article ID doesn't exist, returns a 400 response

**/comments/:id**

- PUT
  - Receives comment ID from URL parameter and updated comment from `comment` property of request body
  - Updates body of corresponding comment in database, returns a 200 response with the updated comment on `comment` property of the response body
  - If comment with given ID does not exist, returns 404 response
  - If no ID or updated comment is supplied, returns 400 response
- DELETE
  - Receives comment ID from URL parameter
  - Deletes comment from database and removes all references to its ID from corresponding user and article models, returns 204 response
  - If no ID is supplied or comment with supplied ID doesn't exist, returns 404 response

**/comments/:id/upvote**

- PUT
  - Receives comment ID from URL parameter and username from `username` property of request body
  - Adds supplied username to `upvotedBy` of corresponding comment if user hasn't already upvoted the comment, removes username from `downvotedBy` if that user had previously downvoted the comment, returns 200 response with comment on `comment` property of response body
  - If no ID is supplied, comment with supplied ID doesn't exist, or user with supplied username doesn't exist, returns 400 response

**/comments/:id/downvote**

- PUT
  - Receives comment ID from URL parameter and username from `username` property of request body
  - Adds supplied username to `downvotedBy` of corresponding comment if user hasn't already downvoted the comment, remove username

    - from `upvotedBy` if that user had previously upvoted the comment, returns 200 response with comment on `comment` property of response body
    - If no ID is supplied, comment with supplied ID doesn't exist, or user with supplied username doesn't exist, returns 400 response

Bonus: YAML Saving and Loading

Currently every time you start and stop your server, your database object will get erased as it isn't being saved anywhere. There are many potential formats for saving the database object to ensure it is able to be restored. For this project, as a bonus, we encourage you to use YAML. You will write two functions, one that saves your database object to YAML after each server call, and another that loads the database object when the server starts. We have implemented the logic for calling these functions, it is your task to find appropriate JavaScript modules for this functionality and writing the following functions:

**loadDatabase**

- Reads a YAML file containing the database and returns a JavaScript object representing the database

**saveDatabase**

- Writes the current value of `database` to a YAML file

Implementing this Bonus will be a tough challenge, as you'll probably have to use a new external library, and you may need to add to or edit the rather complicated `requestHandler` function that we have provided. We use Figg in our solution code, but you can implement saving and loading in many ways, so don't feel like you have to take the same approach that we did.

# Testing

A testing suite has been provided for you, checking for all essential functionality and edge cases.

To run these tests, first, open the root project directory in your terminal. Then run `npm install` to install all necessary testing dependencies (if you haven't already). Finally, run `npm test`. You will see a list of tests that ran with information about whether or not each test passed. After this list, you will see more specific output about why each failing test failed.

As you implement functionality, run the tests to ensure you are creating correctly named variables and functions that return the proper values. The tests will additionally help you identify edge cases that you may not have anticipated when first writing the functions.