CSE 316 – Fundamentals of Software Development
Fall, 2023
Programming Assignment 01 – HTML, CSS, Javascript

Assigned: Friday, 09/08/2023
Due: Monday, 10/02/2023, at 11:59 PM

Learning Outcomes

After completing this homework assignment, you will be able to

1. Create static web pages using HTML/CSS.
2. Add dynamic behavior to web pages using JavaScript.
3. Implement a basic MVC architecture.

Introduction

In this assignment, we will create a mock stackoverflow.com web application using HTML, CSS, and JavaScript. All subsequent assignments will be about adding more functionality to what we build here so it is important that you finish the first assignment.

Stack Overflow is a question-and-answer website for programmers. In this version, we will develop a prototype where users will be able to browse questions in the system, ask their own questions, and answer questions asked.

Initial State

`index.html` – Interface for web application (mostly empty, includes reference to index.js and index.css)
`README.md` – Description of work on assignment
`css/`
    `index.css` – Stylesheet for assignment (starts empty)
`src/`
    `index.js` – module to add dynamic behavior to HTML page (starts empty)
    `model.js` – defines data model for application (contains sample data, but all functionality must be implemented by student).

Getting Started

1. Download/clone your GitHub repository.
2. The main HTML page for this application is `index.html`, which you will use to startup your application in a browser.
3. This file, `index.html`, includes a CSS stylesheet `css/index.css` that is used to specify the appearance of HTML elements included in `index.html`. Any CSS styling applied to HTML elements should be implemented here.
4. This file, `index.html`, includes a JavaScript module `src/index.js` that is used to make the static HTML page dynamic. The file is empty. *You are expected to write code in this file as per the defined requirements*.
5. Also, this file, `src/index.js`, imports `src/model.js` which defines the data model underlying the application. Any operations to query or update the model should be added to this file.

Note we are following a Model-View-Controller architecture in this homework assignment. The view part of the application is the HTML page `index.html` (along with `css/index.css`). The controller is the JavaScript module `src/index.js`. The underlying data model is in `src/model.js`. You can create other JavaScript modules in the src directory.

You will need a local web server. Follow the steps to set up a local web server and run your application.

1. Install Python3. (skip if already installed).
2. Open a terminal (Mac or Linux) or Powershell (Windows).
3. Navigate to the directory that has the repository of this homework.
   ```
   $ cd </path/to/directory>
   ```
4. Run the http module as a script from your terminal.
   ```
   # If Python version returned above is 3.X
   # On Windows, try "python -m http.server" or "py -3 -m http.server"
   $ python3 -m http.server
   ```
5. (Optional) By default, this will run the server on localhost's port 8000. You can change the port number if something else is running on 8000 by adding a port number like so: python3 -m http.server 3000.
6. Open a browser (e.g., Chrome) and open http://localhost:8000 and you will be able to see your application's *index.html*.

Setting up a web server that runs on our local machine is necessary as we are importing JavaScript code as modules from the local filesystem, which is disallowed by browsers to prevent security threats via Cross-Origin Resource (CORS). To know more about it you can read this.

What is the role of the web server? It is a program (a Python program in our case) that continuously runs on our local machine and waits till it receives a request from the browser. When the browser sends it a request it sends a response back to the browser. The response is the html file *index.html*. Why? Because the server was pre-configured to look for *index.html* in the current directory and send it as a response each time a request is received. All this happens over the HTTP protocol so the problem of cross-origin where an HTTP request tries to access a non-HTTP resource doesn't occur.

Most development teams follow a similar approach in their local development environments. In a production environment, all resources including JavaScript modules are imported from a backend server via HTTP to avoid cross-origin resource sharing due to importing modules from the local filesystem. We will learn more about this in homework 3 when we build an actual web server.

Data Model

The primary data elements we need to store for this application are questions, tags, and answers. To this end, we will use a *JavaScript Object* to represent the application data in memory. The object has the following attributes:

- *questions*. A list of *Question* objects.
- *tags.* A list of *Tag* objects.
- *answers*. A list of Answer objects.

The *Question* object has the following attributes:
- *qid*. A unique string used to uniquely identify this question.
- *title*. A string to hold this question's title.
- *text.* A string to hold this question's text.
- *tagIds.* A list of strings to hold the tag ids of tags associated with this question.
- *askedBy. A s*tring to indicate the username associated with this question.
- *askDate.* Date object to indicate the date when the question was asked.
- *ansIds.* An array of strings to hold the answers ids of answers associated with this question. See below for the answer id type.
- *views.* A number to indicate the no. of times the question has been viewed.

The *Tag* object has the following attributes:
- *tid*. A unique string used to uniquely identify this tag.

- *name*. A string to hold this tag's name.

The *Answer* object has the following attributes:

- *aid*. A unique string used to uniquely identify this answer.
- *text.* A string to hold this answer's text.
- *ansBy. A s*tring to indicate the username associated with this answer.
- *ansDate.* Date object to indicate the date when the answer was posted.

See the class definition of this object in src/model.js in the code repository.

Since we are working with a client-side scripting language the data will not be persistent. This means that anytime the application is stopped and restarted, all the data created during that session will be lost and the application will begin with the initial state. This is fine for the purposes of this homework assignment. In later homework assignments, we will see how the application state can be persisted in a backend database/filesystem using server-side scripting.

Application Behavior and Layout

We will mimic the original stackoverflow.com website as much as possible. Although, we won't be implementing all of its features. You can visit stackoverflow.com for inspiration. The layout is quite simple. It has two parts – a **header** and the **main body**. The *header* should remain constant, that is, it should have the same UI elements throughout and should be displayed at the top of the page. The *main body* will be displayed below the header and will render relevant content based on user interactions with the web page.

Home Page

When a user loads the application in the browser for the first time, the home page should be displayed as shown in figure 1. The home page has two parts, the *banner* and the *main* body which will display the content. The *banner* should be displayed at the top of the page and it should contain the following

- The title of the application **Fake Stack Overflow**
- A search bar where users can do textual searches.

The *main body* has two parts. The left side is a menu and the right side displays all questions asked in the forum.

The menu has two links – *Questions* and *Tags*. Clicking on the *Questions* link always displays the home page, i.e, the page being currently described. Clicking on the *Tags* page will display the Tags page (described later). If the user is currently on the page that shows all questions, the *Questions* link should be highlighted with a gray background color. If the user is on the Tags page, the *Tags* link should be highlighted with gray background color. Further, the right side of the *main body* of the *home* page should be displayed as shown in figure 1 with the following elements:

- A header which displays the text **All Questions** and a *button* with the label **Ask Question**.
- The total number of questions currently in the model.
- Three buttons – *Newest*, *Active,* and *Unanswered.*
    - Clicking the *Newest* button should display all questions in the model sorted by the date they were posted. The most recently posted questions should appear first.
    - Clicking the *Active* button should display all questions in the model sorted by answer activity. The most recently answered questions must appear first.
    - The *Unanswered* button should display only the questions that have no answers associated with them.
- Each question should be displayed as shown in figure 1 with the following elements:
    - The no. of answers and the no. of times a question has been viewed. Every time a user clicks on a question should increase the no. of views by 1.
    - Question title.
    - Question metadata, which includes the username of the user who posted the questions and the date the question was posted. The metadata has a particular format.
        - If a question was posted on day X, then, for the entirety of day X, the question date should appear in seconds (if posted 0 mins. ago), minutes (if posted 0 hours ago), or hours (if posted less than 24 hrs ago).
        - On the other hand, if we were viewing the page 24 hrs after the question was posted then the metadata should be displayed as *<username> asked <Month><day> at <hh:min>*.

        - Further, if the question is viewed a year after the posted date then the metadata should be displayed as *<username> asked <Month><day>,<year> at <hh:min>*.

        Here are a few examples:

        - question posted on Feb 9th, 2022, *09:20:22* and viewed on the same day at 09:20:58, the metadata should be displayed as *<username> asked 36 seconds ago*.
        - question posted on Feb 9th, 2022, 09:20:22 and viewed on the same day at 09:25:58, the metadata should be displayed as *<username> asked 5 minutes ago*.
        - question posted on Feb 9th, 2022, 09:20:22 and viewed on the same day at 11:30:21, the metadata should be displayed as *<username> asked 2 hours ago*.
        - question posted on Feb 9th, 2022, *09:20:22* and viewed on Mar 31, 2022, 09:20:58, the metadata should be displayed as *<username> asked Feb 9 at 09:20*.
        - question posted on Feb 9th, 2022, *09:20:22* and viewed on Mar 31, 2023, 09:20:58, the metadata should be displayed as *<username> asked Feb 9, 2022 at 09:20*.
    - All questions should be displayed in *Newest* order by default.
    - There should be a dotted line to divide each question entry.
    - If the total no. of questions is more than the page can hold, add a scroll bar.
- Make sure that all fonts and content are clearly legible. They don’t have to be the exact same as the fonts in figure 1.

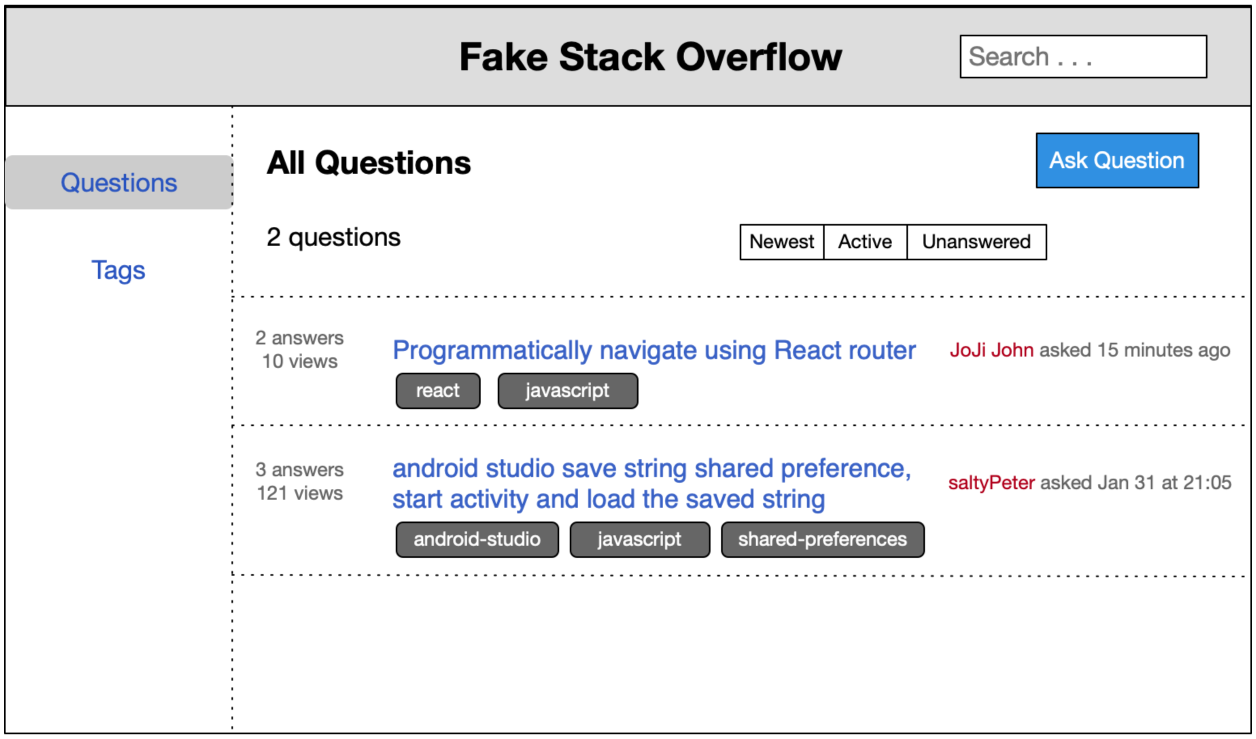
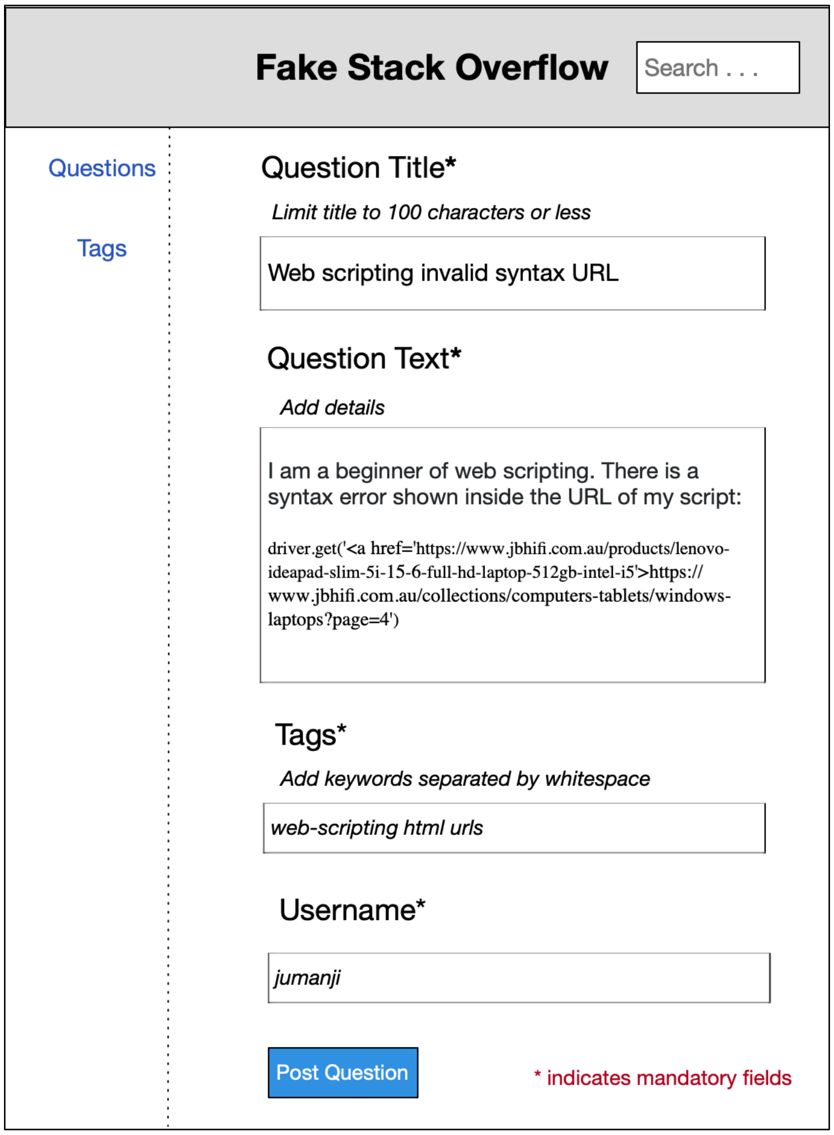


Figure 1

New Question Page

When a user clicks on the **Ask Question** button, the *main body* section of the page should display a form as shown in figure 2 with the following elements:

- A text box for question title. The title should not be more than 100 characters and should not be empty.
- A text box for question text. Should not be empty. No restriction on max length of characters.
- A text box for a list of tags that should be associated with the question. This is a whitespace-separated list. Should not be more than 5 tags. Each tag is one word, hyphenated words are considered one word. The length of a new tag cannot be more than 10 characters.
- A text box for the username of the user asking the question. The username should not be empty.
- A button with the label *Post Question*.
- Each input element in the form should have an appropriate hint to help the user enter the appropriate data as shown in Figure 2.
- Display an appropriate error message for invalid inputs below the respective input element.



Web scripting invalid syntax URL

I am a beginner of web scripting. There is a syntax error shown inside the URL of my script:

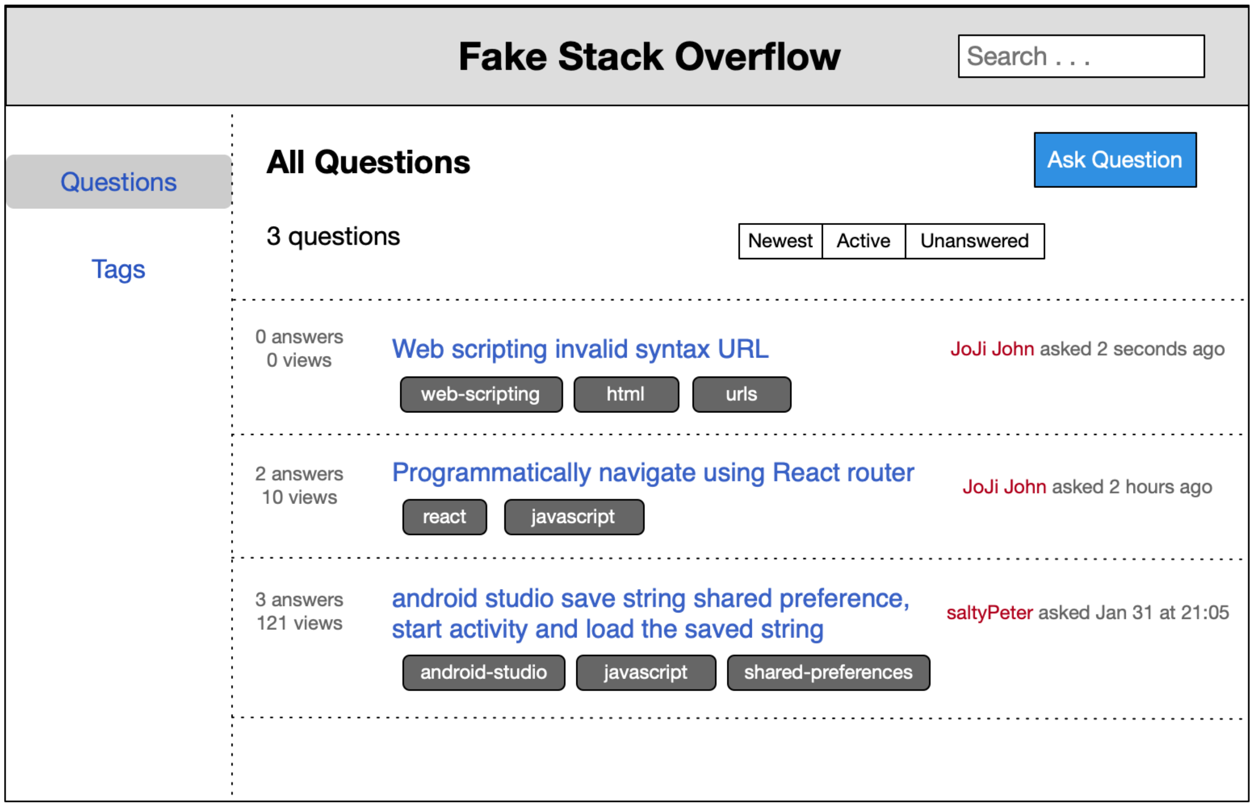
```
driver.get('<a href='https://www.jbhifi.com.au/products/lenovo-ideapad-slim-5i-15-6-full-hd-laptop-512gb-intel-i5'>https://www.jbhifi.com.au/collections/computers-tablets/windows-laptops?page=4')
```

web-scripting html urls

jumanji

Figure 2

When the *Post Question* button is pressed, the question should be added to the *data object* in *model.js*. If the question is added successfully then the user should be taken to the home page where the *main body* section should display all the questions including the question just added. Furthermore, the page should also display the total no. of questions, which should have incremented by 1. Figure 3 shows an example where the first question displayed was most recently asked by the user *JoJi John* using the inputs on the new question form.



0 answers
0 views
Web scripting invalid syntax URL
web-scripting html urls
JoJi John asked 2 seconds ago

2 answers
10 views
Programmatically navigate using React router
react javascript
JoJi John asked 2 hours ago

3 answers
121 views
android studio save string shared preference, start activity and load the saved string
android-studio javascript shared-preferences
saltyPeter asked Jan 31 at 21:05

Figure 3

Searching

A user can search for certain questions based on words occurring in the question text or title. On pressing the ENTER key, The search should return *all questions for which their title or text contains at least one word in the search string*. For example, in figure 4, there is only one question in our data model with title or text that matches the search string ‘Shared Preference’.

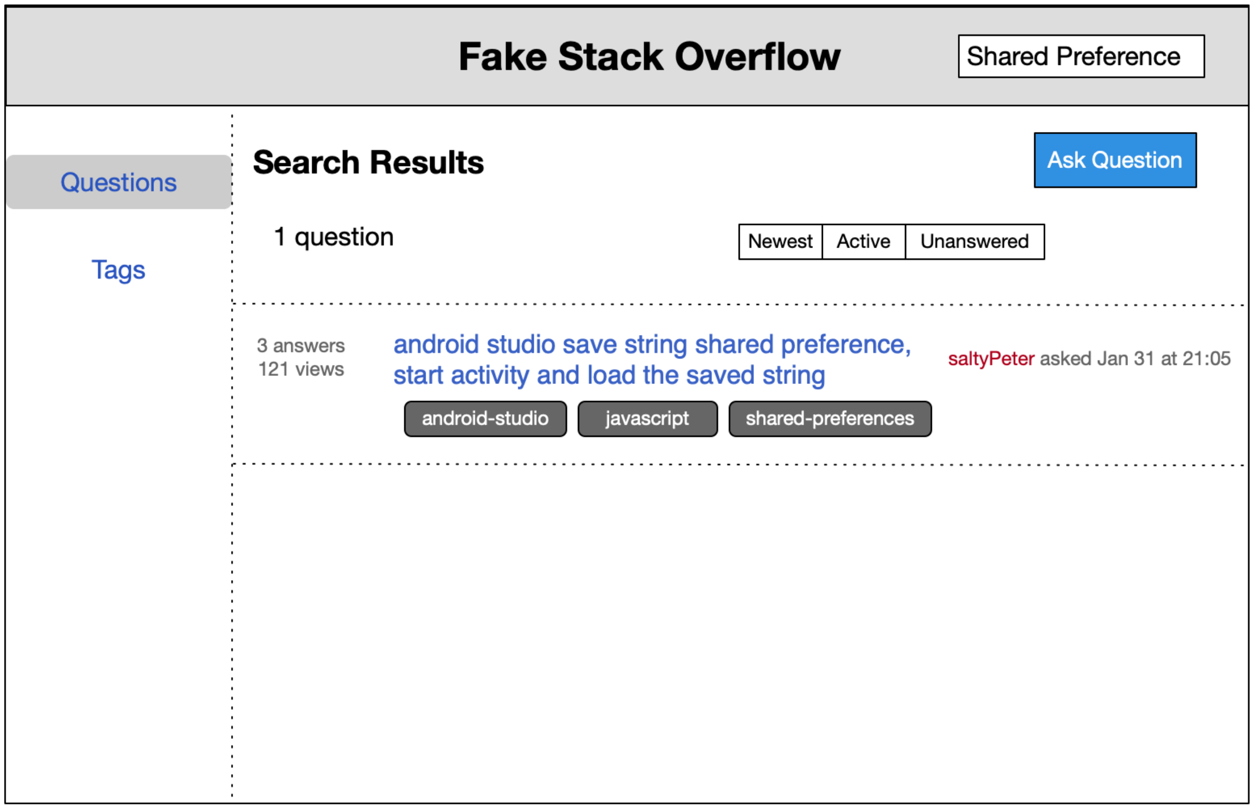


Figure 4

Furthermore, if a user surrounds individual words with [] then all questions with a tagname in [] should be displayed. *The search results should be displayed when the user presses the **ENTER** key*. See figure 5.

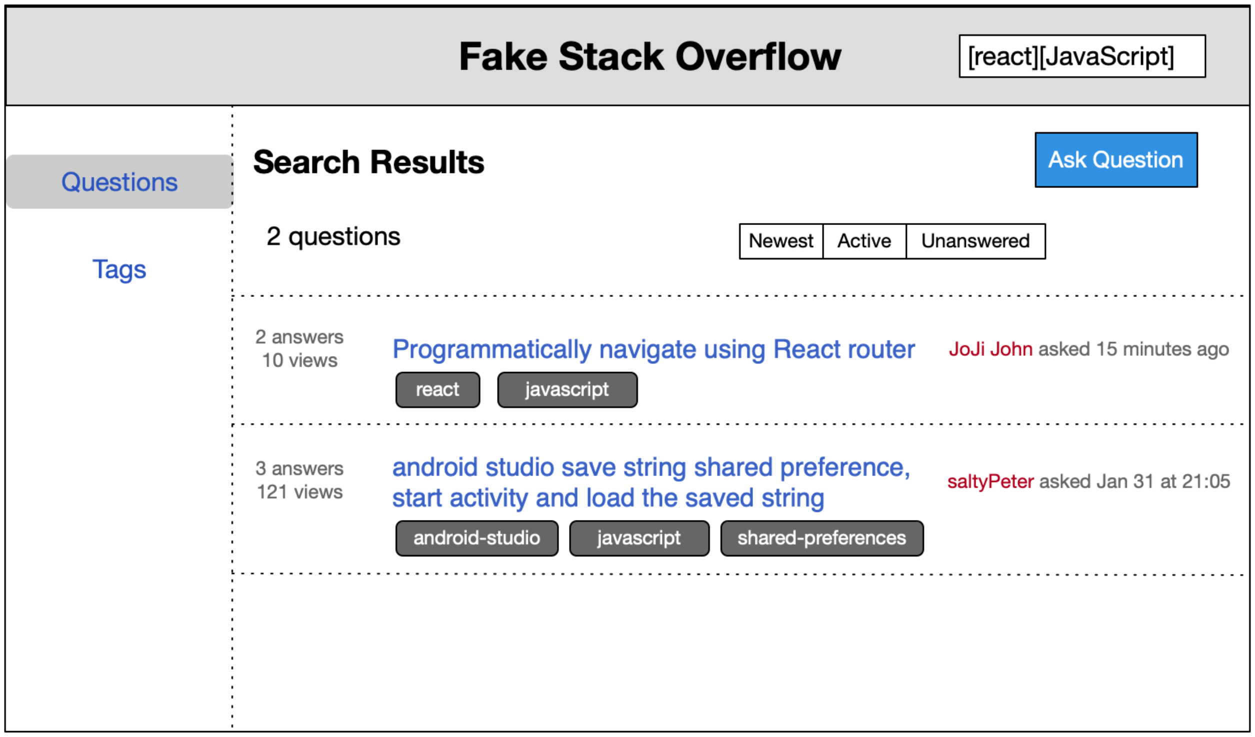
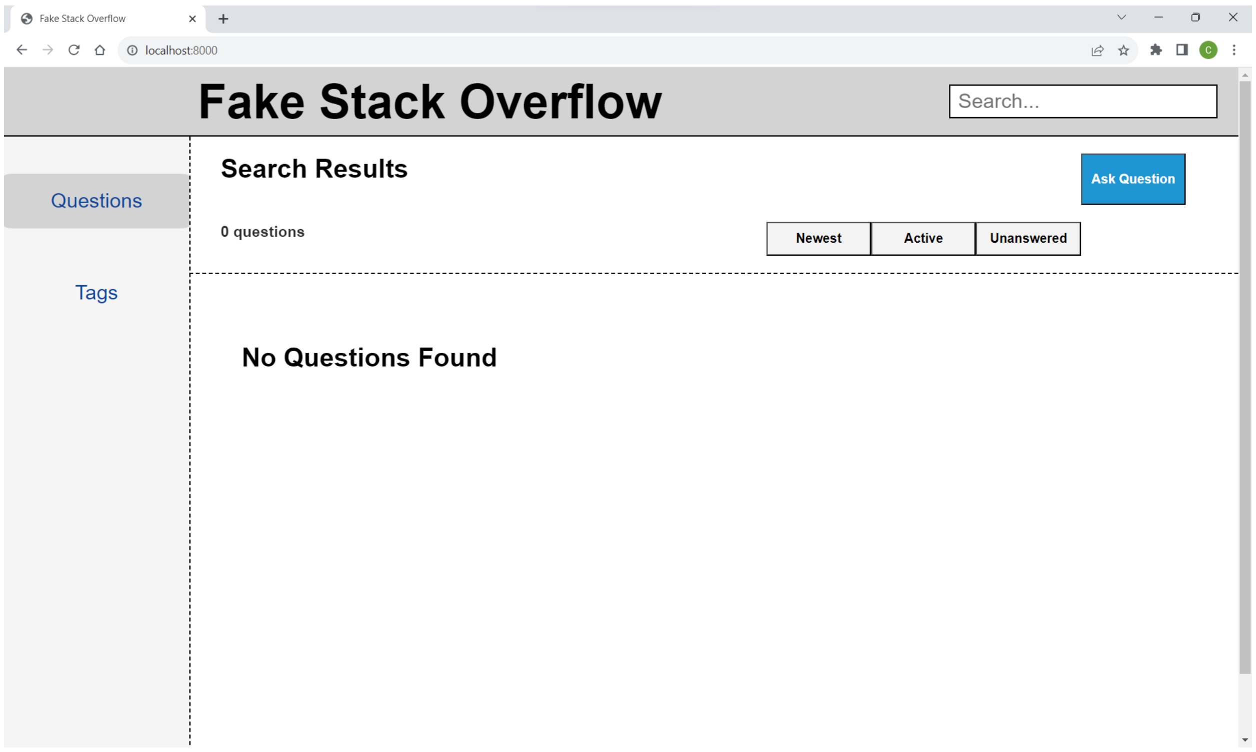


Figure 5

Note the searching is case-insensitive. Also, a search string can contain a combination of [tagnames] and non-tag words, that is, not surrounded with []. In this scenario, all questions tagged with at least one tag in the search string or text/title containing at least one of the non-tag words should be displayed. For example, if the search string is [react][android] javascript then all questions tagged with *react* or *android* or both should be considered. Also, questions with the non-tag word *javascript* in their text/title should be considered.

If the search string does not match any question or tag names then display the **No Questions Found**. The total number of questions displayed should be 0 and the page title and the button to ask a question should remain.



Fake Stack Overflow

Search Results

0 questions

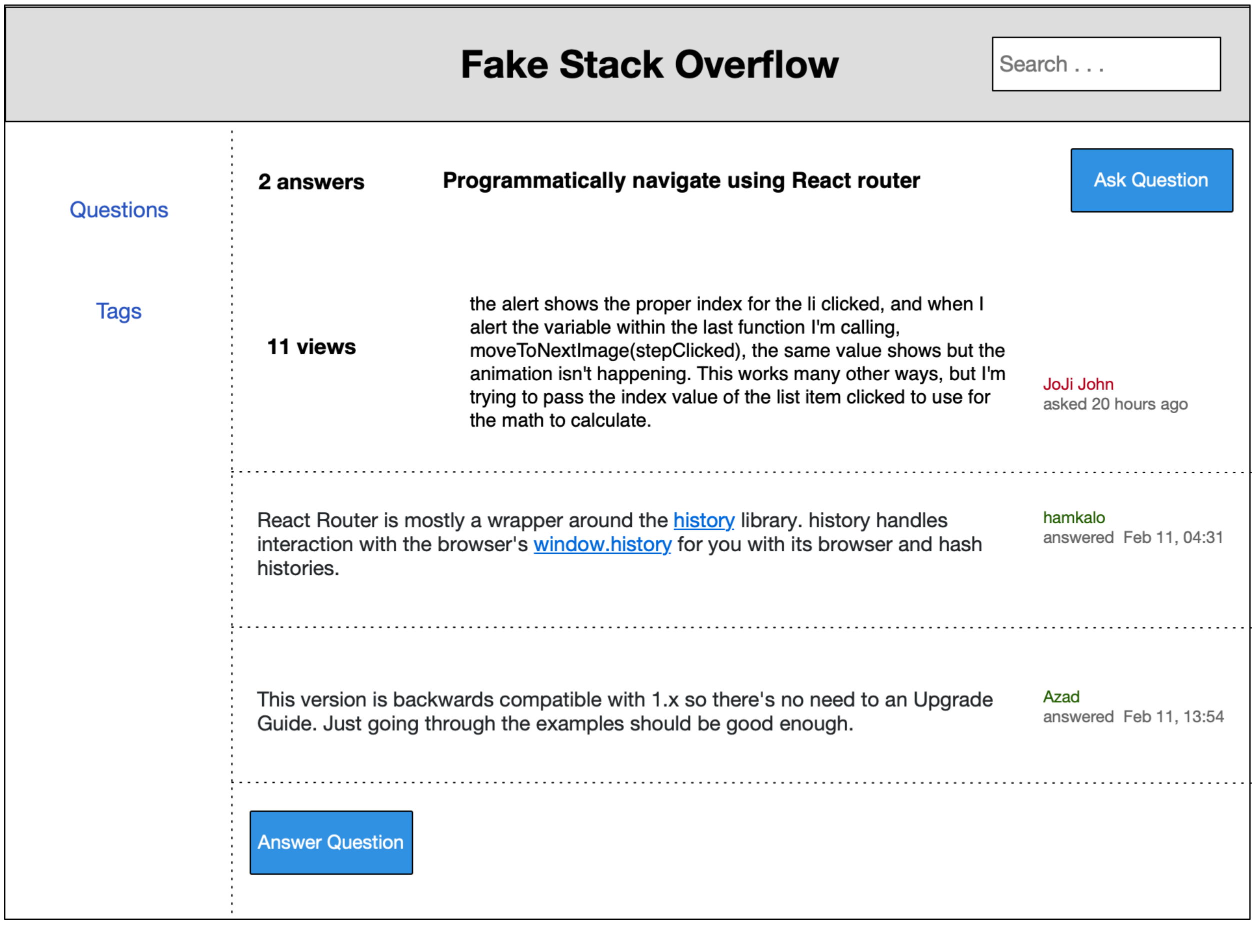
No Questions Found

Answers Page

Clicking on a question link should increment by 1 the no. of views associated with the question and load the answers for that question in the *main* section of the home page. Note the banner should still remain at the top of the page. The answers should be displayed as in figure 6 with the following elements:

- The text **N answers**, where N is the total no. of answers given for the question. The title of the question. A *button* with the label **Ask Question**. You are free to add other style constraints to the elements other than what has already been shown. However, make sure that they are clearly legible.
- The text **N views** indicating the no. of times the question has been viewed (including this one).
- The question text.
- The metadata for the question **<user> asked <date>**. This metadata format is the same as the one described in the page that displays all questions.
- The answers to the question. An answer has 2 parts as shown in figure 6
    - the answer itself,
    - the answer metadata in the format **<user> answered <date>**. The date format requirements in the metadata are exactly the same as the requirements for questions (see home page described before).
- If no. of answers does not fit on the page, then add a scroll bar.

- The answers should be displayed in descending order of the day and time they were posted. In other words, display the answers that were posted most recently first.
- A *button* with the label **Answer Question** at the end of all answers. Make sure that the button and the label are clearly visible. You are free to add a different style from the one shown.
- Answers must be divided by a dotted line as shown in figure 6.



the alert shows the proper index for the li clicked, and when I alert the variable within the last function I'm calling, moveToNextImage(stepClicked), the same value shows but the animation isn't happening. This works many other ways, but I'm trying to pass the index value of the list item clicked to use for the math to calculate.

React Router is mostly a wrapper around the history library. history handles interaction with the browser's window.history for you with its browser and hash histories.

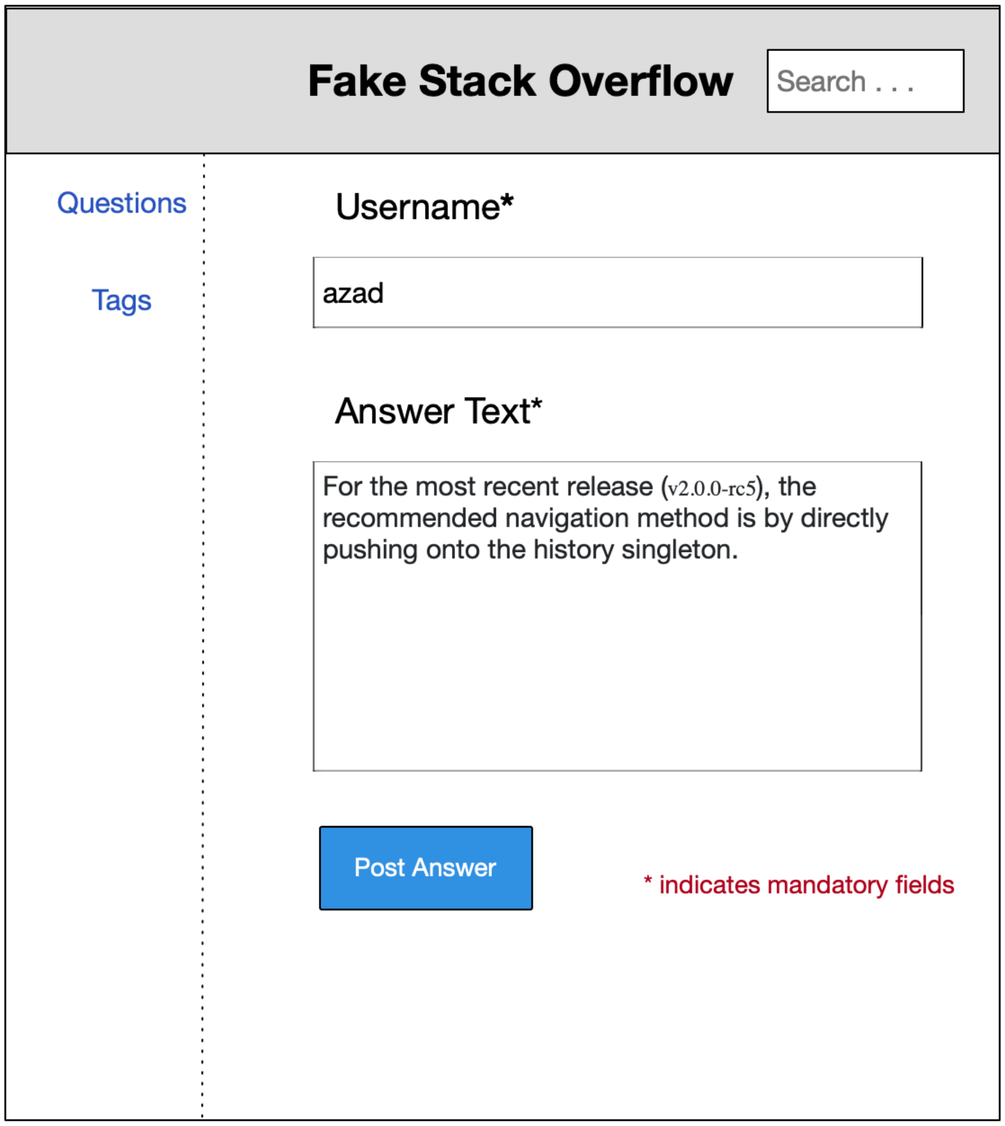
This version is backwards compatible with 1.x so there's no need to an Upgrade Guide. Just going through the examples should be good enough.

Figure 6

Pressing the *Ask A Question* button on this page will render elements as described in the **New Question Page** section.

New Answer Page

Pressing the *Answer Question* button will display a page with input elements to enter the new answer text and username. Note the menu should remain on the left of the page as shown in figure 7.



azad

For the most recent release (v2.0.0-rc5), the recommended navigation method is by directly pushing onto the history singleton.

Figure 7

Pressing the *Post Answer* button, should capture the answer text and the username and update the data model. If the inputs have no errors, then all the answers are displayed as shown on the **Answers Page.** Note there should now be 3 answers for this question since a new answer was posted and this answer must be the first one shown in the list.

If the answer text or username is empty, then display appropriate error messages below the respective input fields.

Tags Page

Clicking on the *Tags* link in the menu should display the list of all tags in the model. Tag names are case-insensitive so the tag name ‘React’ and ‘react’ should be considered the same for all practical purposes. Additionally, the *Tags* link in the menu should be highlighted with a gray background since the *Tags* page is being displayed currently. The page should render the following elements as shown in figure 8:

1. The text **N Tags**, where **N** is the total number of tags.
2. The text **All Tags**.
3. A button with the label **Ask Question**. This is the same button that was described in the *Questions* and *Answers* pages.
4. Tag names in groups of 3, that is, each row should have at most 3 tags. Each tag should be displayed in a box with dotted borders. The block should display the tag name as a link and the no. of questions associated with the tag in a new line in the same block.

Figure 8 shows an example of the page.

Upon clicking a tag link, all questions associated with the tag should be displayed. For example, if the *javascript* tag is clicked then the page should show all questions with the *javascript tag*. The style is similar to the home page.

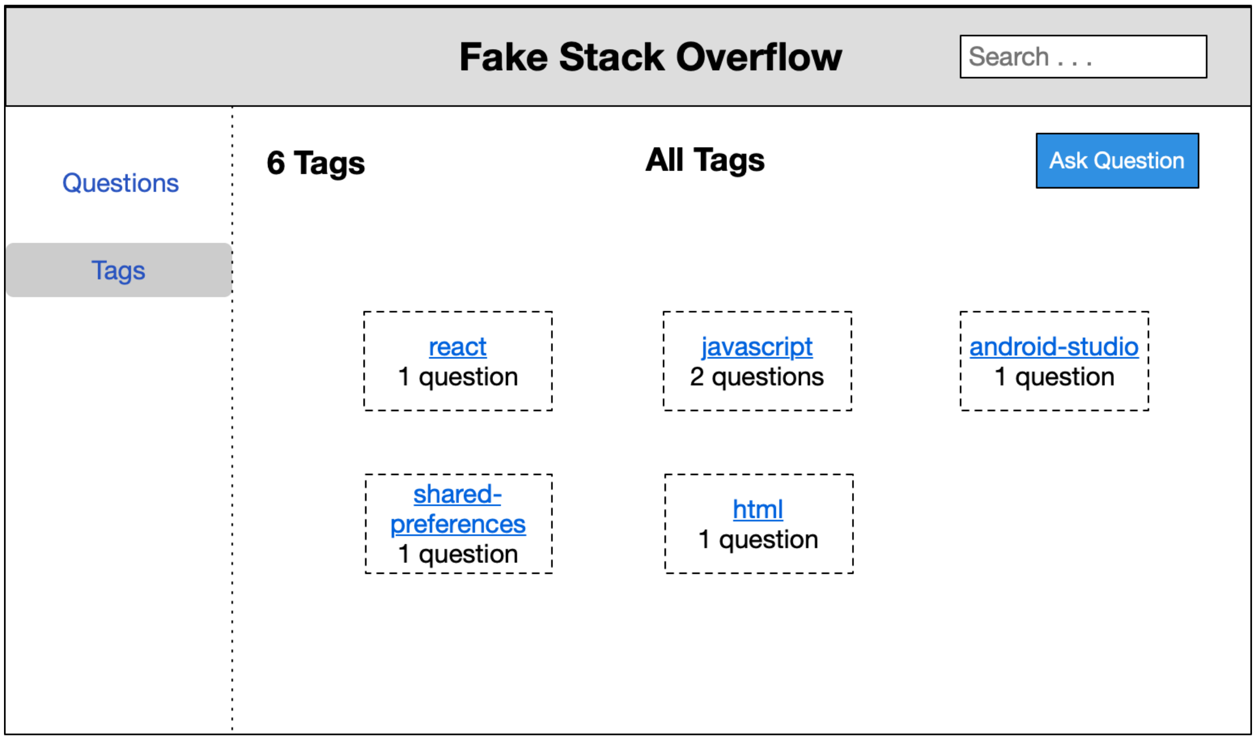


Figure 8

Submission Instruction

You can submit code to your GitHub repository as many times as you want till the deadline. After the deadline, any code you submit will not be considered. To submit a file to the remote repository, you first need to add it to the local git repository in your system, that is, the directory where you cloned the remote repository initially. Use the following commands from your terminal:

```
$ cd /path/to/cse316-hw1-<username> (skip if you are already in this directory)
```

```
$ git add <filename-you-want-to-add>
```

To submit your work to the remote GitHub repository, you will need to commit the file (with a message) and push the file to the repository. Use the following commands:

```
$ git commit -m "<your-custom-message>"
```

```
$ git push
```

IMPORTANT: In the README.md file of your repository each team member should list their contribution. Without this description neither of you will get any points.

Grading

We will clone your repository and test your code in the Chrome web browser. You will get points for each functionality implemented. **Make sure you test your code in Chrome**. The rubric we will use is shown below:

1. Home Page: 20 pts.
2. Post a New question: 10 pts.
3. Searching by text: 10 pts.
4. Searching by tags: 10 pts.
5. Answers Page: 10 pts.
6. Post a new answer: 10 pts.
7. All Tags page: 20 pts.
8. Questions of a tag: 10 pts.

Total points: 100 pts.