CSE 316 – Fundamentals of Software Development
Fall, 2023
Programming Assignment 02 - React

Assigned: Monday, 10/02/2023
Due: Monday, October 16, 2023, 11:59 PM

Learning Outcomes:
After completing this homework assignment, you will be able to

1. Learn the principles of reactive applications.
2. Develop dynamic web pages using React.

Introduction:
In this assignment, we will re-create a mock stackoverflow.com web application using the React Library. Since we will be using the same style for this application, you can use the same CSS that you made for Programming Assignment 01.

Getting Started
Install Node.js. We will need Node.js to manage the react app and its dependencies. *We do not actually need to know Node.js yet*. When you install *Node.js*, it will come with the **npm** package manager, which will also get installed. We will use **npm** to install dependencies and start the react application.

Download/clone your GitHub repository (link posted in Brightspace). The repository is structured as a react application. The `public/` directory has metadata about the application such as images and logos and the main HTML page, `index.html`. **You should not change anything in this file.**

The `src/` directory has three directories, `components`, `models`, and `stylesheets`, and two files `index.js` and `App.js`. The `index.js` file is used to render the component exported in `App.js` in `pubic/index.html`'s "*root"* div. You do not need to change `index.html`. You can change `App.js` to return a different root component if you so wish.

All components that you define as part of the application must be defined in files in the `components` directory. Also, you should have several component files in the `component` directory. You will lose points if you violate this requirement. The idea is to make your code modular instead of having a monolithic code base, which is hard to understand and maintain.

You can re-use the CSS file you used for PA01. CSS stylesheets should be placed in the `stylesheets` directory and imported in `App.js` as needed.

The `models` directory defines the underlying data structure that the application uses. Currently, it only has `model.js` and is identical to the one you used in PA01. You should add your own methods to the class in this file, similar to PA01.

In the repository, you will find two files called `package.json` and `package-lock.json`. **Do not change these files**. If you have added dependencies to these files, then you will need to commit these files to the repository so we can run your application with the same dependencies. However, there should be no reason for you to change these files. These files list the external dependencies which we will need for the react application to run successfully on any machine.

When you clone the repository run the command **`npm install`**. Running the command will create a directory called `node_modules` which will contain all the required packages. You can then start your application using the command **`npm start`**. *Running this command will open a local host server at port 3000*. The application will automatically open in a browser. If it does not then manually open *`http://localhost:3000/`* in a browser. In the browser, you should see the message **Replace with relevant content** displayed. You can now proceed and make changes to the code base. If you change code and save it, the server will automatically reflect the changes.

Note: when you commit your code to the repository, the `node_modules` directory will not be committed. This is expected. We will run *`npm install`* on our end and recreate the required packages. **The `node_modules` directory is very large; do not push it to GitHub.**

Git Workflow (do not skip, must read)

For this assignment, you and your teammate are **required to work on separate branches in your git repository**. The repository already has a **`main`** branch. This branch should be used for your final code. Create a separate branch called **`testdev`.** This branch should be used to merge code between you and your team mate and test the merged code. Having a separate branch from **`main`** for merging purposes will ensure that existing code on the **`main`** branch is preserved and is changed only when merging is successful. Suppose you have a branch named `B1` and your teammate has a branch named `B2`. Here is the workflow you must follow:

1. Create **`testdev`**, `B1`, and `B2` from **`main`** (see *Figure 0.1*).
2. Before starting work on `B1`, pull changes from the **`testdev`** branch.

3. Switch to branch `B1` and merge the **`testdev`** branch so the `B1` has all recent changes made to **`testdev`**. Resolve merge conflicts if any (see later section on resolving merge conflicts).
4. Make your changes in branch `B1`.
5. Commit and push changes made in branch `B1`.
6. Create and submit a pull request (in GitHub) to merge `B1` into the **`testdev`** branch.
7. When you submit a pull request, one of two things will happen – either merge conflicts will be detected or no merge conflicts will be detected. If there are no conflicts, you can proceed to merge the branches. A merge conflict will happen if you have worked on your branch without first merging the changes in the **`testdev`** branch. Basically, if you ignored steps 1 and 2, then you are most likely to see merge conflicts.
8. After merging the pull request, the **`testdev`** branch now has all your changes along with previous changes. Close the pull request.
9. At the end of the exercise, branches `B1`, `B2`, and **`testdev`** should have the same code.
10. Finally, when you have tested the **`testdev`** branch merge it with **`main`**, again using a pull request or PR.

Additional notes:

- Do not close any of the 4 branches listed above. The **`testdev`** branch must be named as such. The `B1` and `B2` branches can have any name.
- The `B1` and `B2` branches must only have commits from the respective team member. For example, if Alice owns branch `B1` then `B1` must have commits from only Alice. Similarly, if Bob owns branch `B2` then `B2` must have commits from only Bob.
- The branches `B1` and `B2` along with the main branch will be used for grading.

Figure 0.1 shows a visual representation of the git workflow described above:

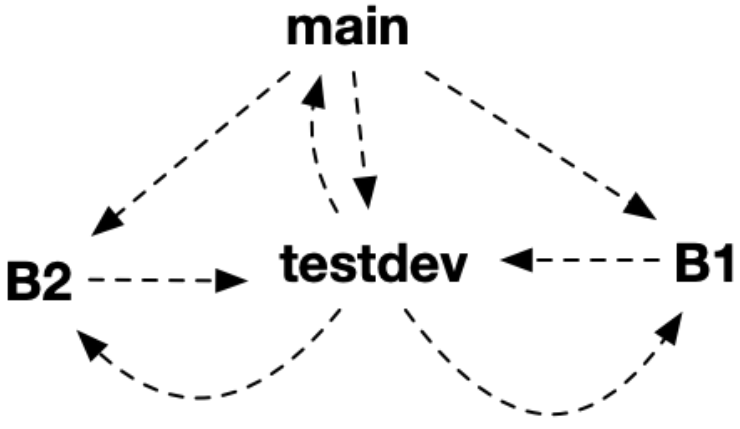

Figure 0.1

Creating Branches

There are two ways to create a branch on your repository.

Method 1 (on github.com):

Visit your remote repository on *github.com* and click the icon that says **"n branches"** (for some number `n`. Then click the "New branch" button.

Method 2 (local machine using the command line terminal):

Open the *terminal* on MAC OSX or Linux. On Windows, open *git bash*.

1. Navigate to your local repository clone path using `cd </path/to/repo>`.
2. Type `git checkout -b <branch-name>`. Replace `<branch-name>` with your branch name. This will create a local branch and checkout to it.
3. Making necessary changes push. This will automatically create the branch (in 2) on the remote repository. NOTE: your push might be rejected with a warning because you haven't set an upstream tracking branch. If this happens, type the command: `git push -u origin <branch name>` or alternatively, `git push --set-upstream origin <branch-name>`.
4. Checkout any existing branch (not a new branch) by using `git checkout <branch_name>`.

Creating pull requests (Step 5)

Let's say you made some changes on your local branch and have pushed a commit onto your remote branch. To merge that commit into your main branch, go to github on the browser and you should see a banner saying "**X branch has recent changes… compare and pull request**". If not, navigate to the pull requests tab, click create pull request, and set your own branch as **compare** and the main branch as **base**. Create the pull request, and then click commit merge. Your commits on the remote branch should now be merged into the main branch unless there is a merge conflict. See images below for reference.

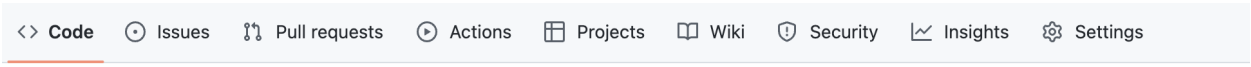

Figure 0.2

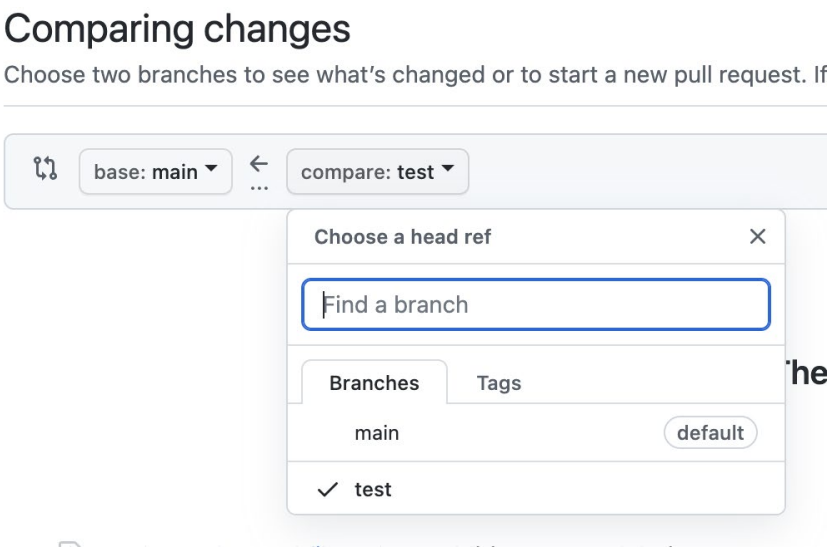

Figure 0.3

Resolving merge conflicts (Step 6)

Merge conflicts occur when two different branches have made edits to the same line in a file, and Git cannot decide which of these edits to accept as the final edit. The developers themselves must decide which of these edits should be accepted in order to resolve a merge conflict. *Figure 0.4* shows a merge conflict that happened when a pull request was created. The conflict needs to be resolved manually and then merged.

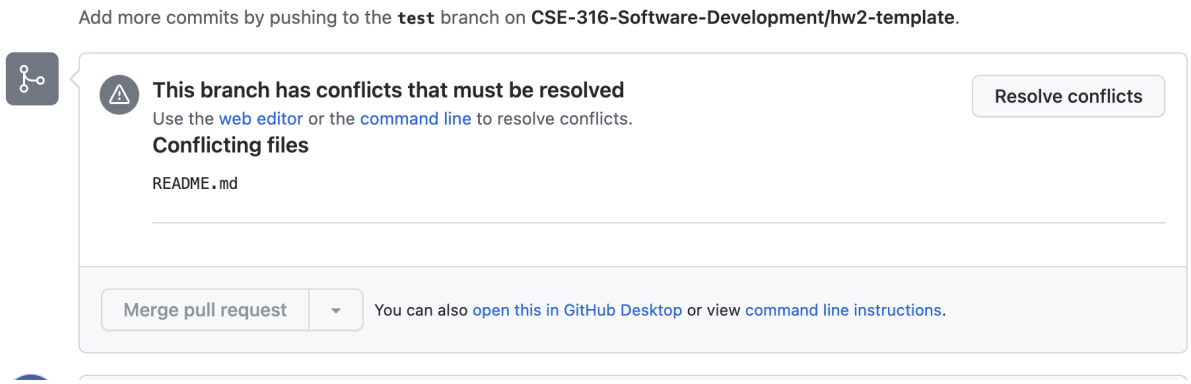

Figure 0.4

When this happens:

1. Manually resolve the conflict by first finding the *resolve conflict* notification in GitHub (as shown in *Figure 0.4*).
2. After discussing with your teammate about which edits should be in the final result, make the proper changes to the file(s). You can edit the files directly on GitHub if you wish using its 'web editor' feature. An example of how conflicts are shown in the code is shown below:

<<<<<<< dev_1_branch
code from dev 1 here
=======
<<<<<<< dev_2_branch
conflicting code from dev 2 here

3. After the edits have been chosen and the files have been finalized, confirm the edits to resolve the merge conflict and merge.
4. Finally, checkout the merged branch and test the changes you have merged. This step is crucial.

Keeping your branch updated (steps 1 & 2)
Before you begin working on your branch, update it using the commands shown below. The commands assume that you are pulling changes from **`testdev`** to your own branch. The commands are the same if you want to merge two other branches, just replace the branch names appropriately.

```
# Switches to testdev branch.
$ git checkout testdev
# Pulls current code in testdev branch.
$ git pull
# Switches to own_branch
$ git checkout own_branch
# merges own_branch with main branch.
# Merge conflicts could happen.
$ git merge testdev
```

Now your local branch will have the updated commits from the **`testdev`** remote branch.

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.

9. Modularity: 5 pts.
   Modularity means that your code should be divided into components defined in different files. Related components should be in one file. If a component needs functionality that belongs to other files, then they must be imported and exported appropriately.
10. Code Quality: 5 pts.
    Your code should comply with standard best practices followed in the React and JavaScript communities.

Total points: 110 pts.

IMPORTANT NOTES:

- In the `README.md` file of your repository each team member should list their contribution. The description should list the features you implemented. The features do not have to map specifically to the grading rubric. Without this description neither of you will get any points.
- You must maintain four branches, **`main`**, **`testdev`,** your branch, and your team member's branch.
- At the end of the project, all four branches must be identical.
- We will test all branches. Your grade will be based on the **main** branch and your branch.
- We will use the rules defined in ESLint https://eslint.org/docs/latest/rules/ to test code quality. To run ESLint on your code base type the following commands:

```
$ cd </path/to/your/repo>
# run eslint on all files in src/
$ npx eslint src
# run esling on individual files.
$ npx eslint src/<file>.js
```

Introduction

In this assignment, we will continue to develop fake stackoverflow.com web application using HTML, CSS, and JavaScript. However, this time we will use the React library to create the UI of the application. **The features of the application are exactly the same as last time, except for one change. See new question, new answer, and answers page.** The new requirement is written in blue. *For convenience, the description of the older features and the data model have been included in this document.*

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 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, 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, 2023, 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, 2023, 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, 2023, *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, 2023, *09:20:22* and viewed on Mar 31, 2023, 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.

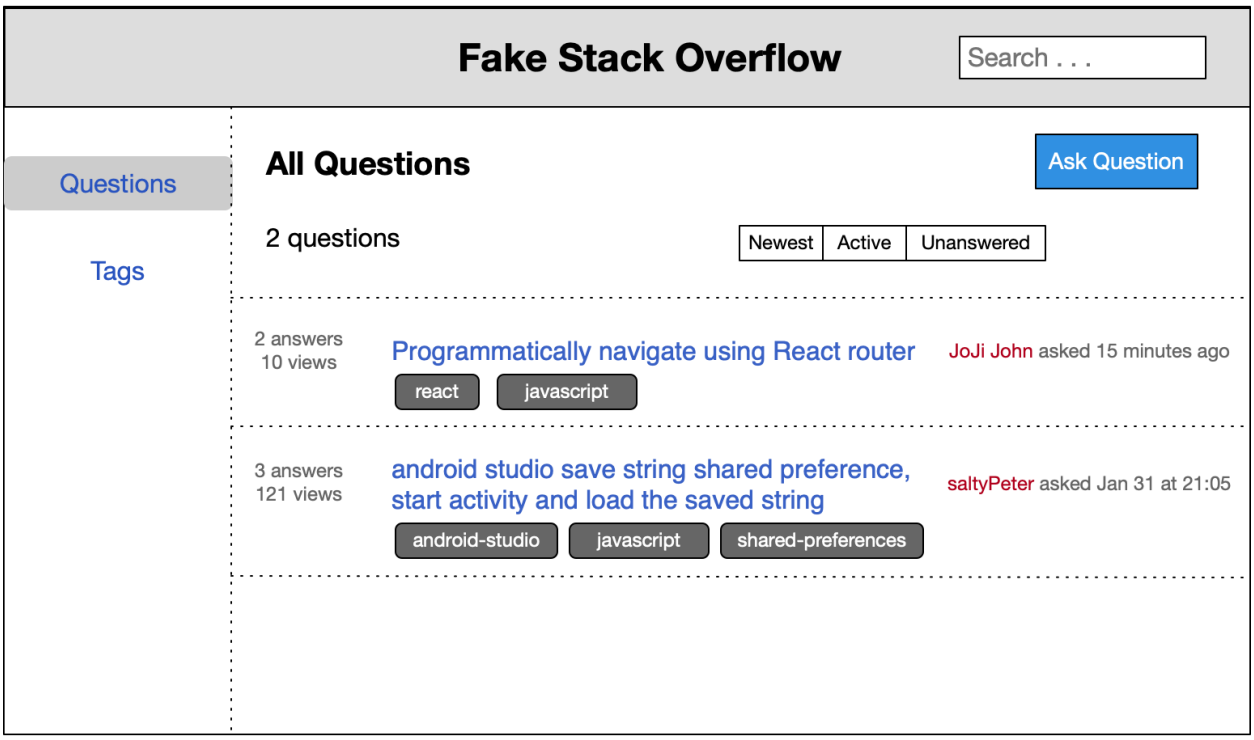

Figure 1

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 user is allowed to add hyperlinks in question text. Hyperlinks are identified in the text if a user encloses the name of the hyperlink in [] and follows it up with the actual link in (). For example, in Figure 2, the question text has a hyperlink "web scripting". The target of a hyperlink, that is, the stuff within () cannot be empty and must begin with "https://" or "http://". Warn the user with an error message if this constraint is violated.
- 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 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. Display an appropriate error message for invalid user input below the text box.
- 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.

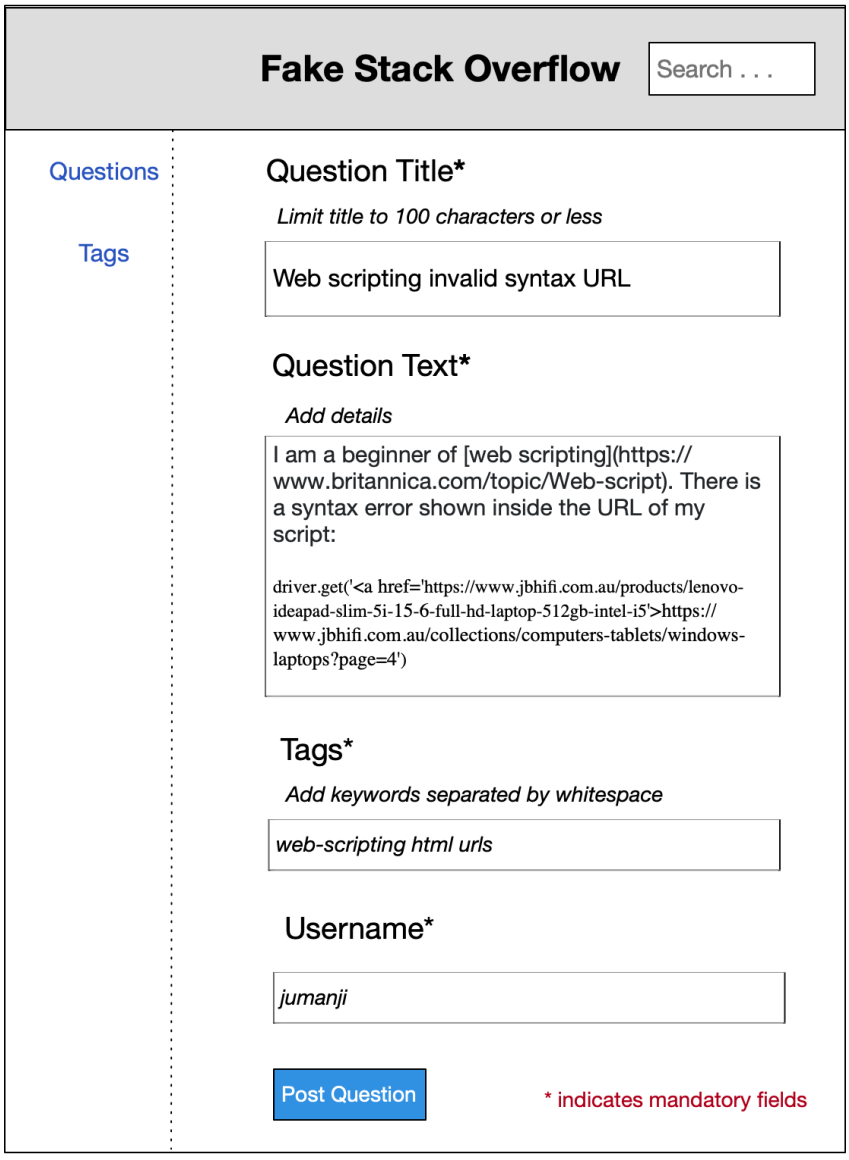

Figure 2

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 currently added. Further, 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 *jumanji* using the inputs on the new question form.

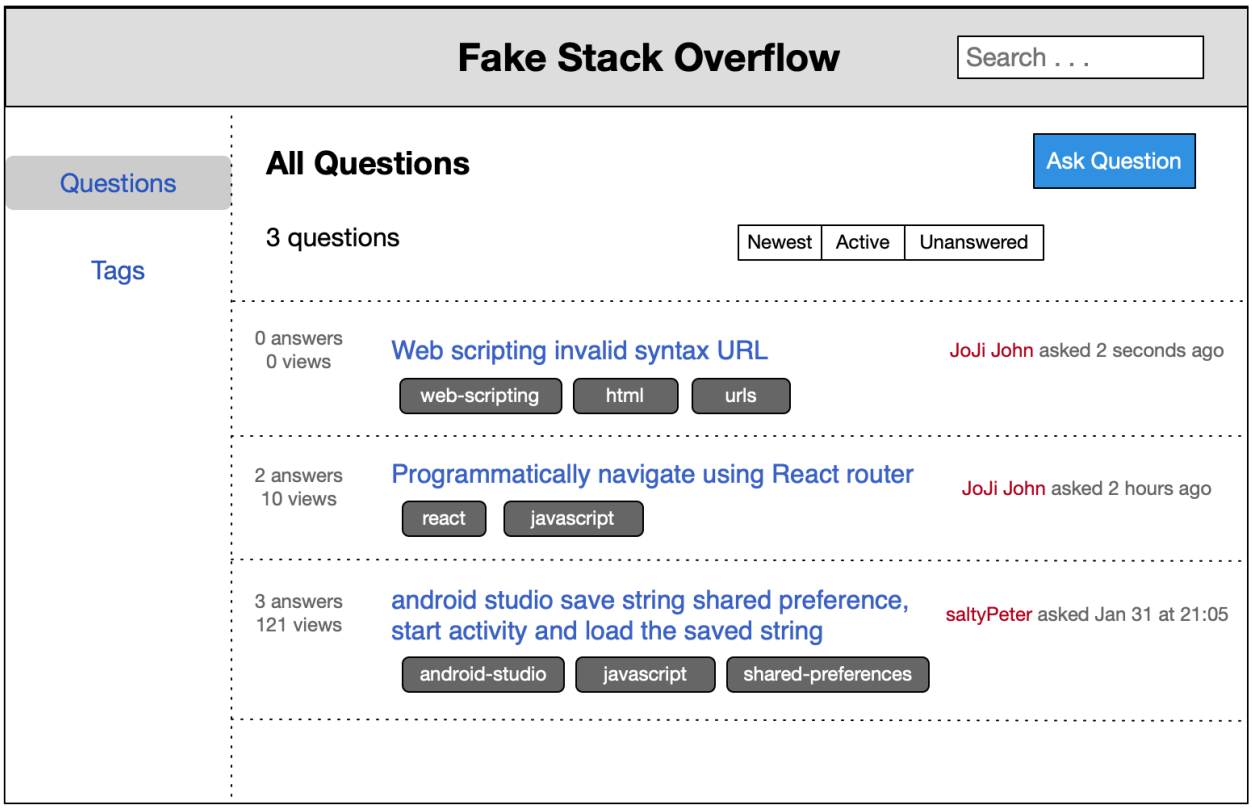

Figure 3

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 in the default (newest) order 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'.

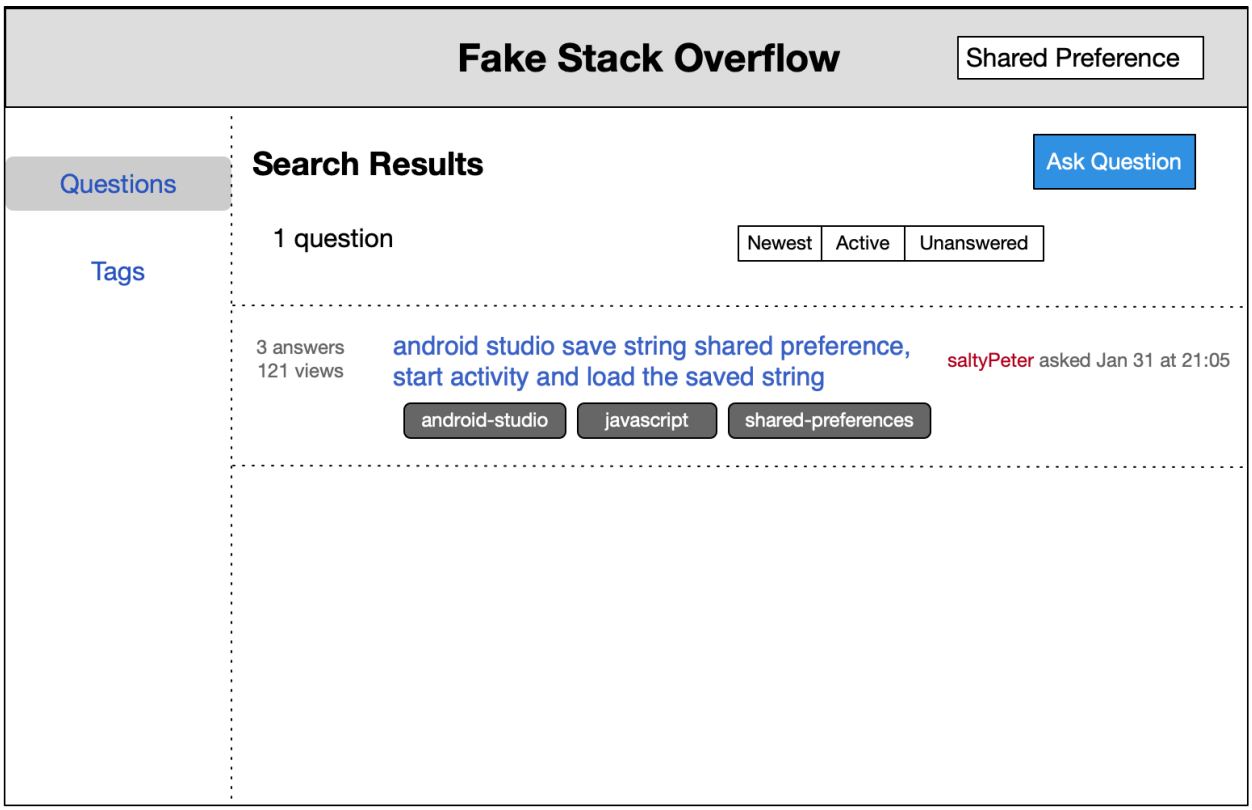

Figure 4

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.

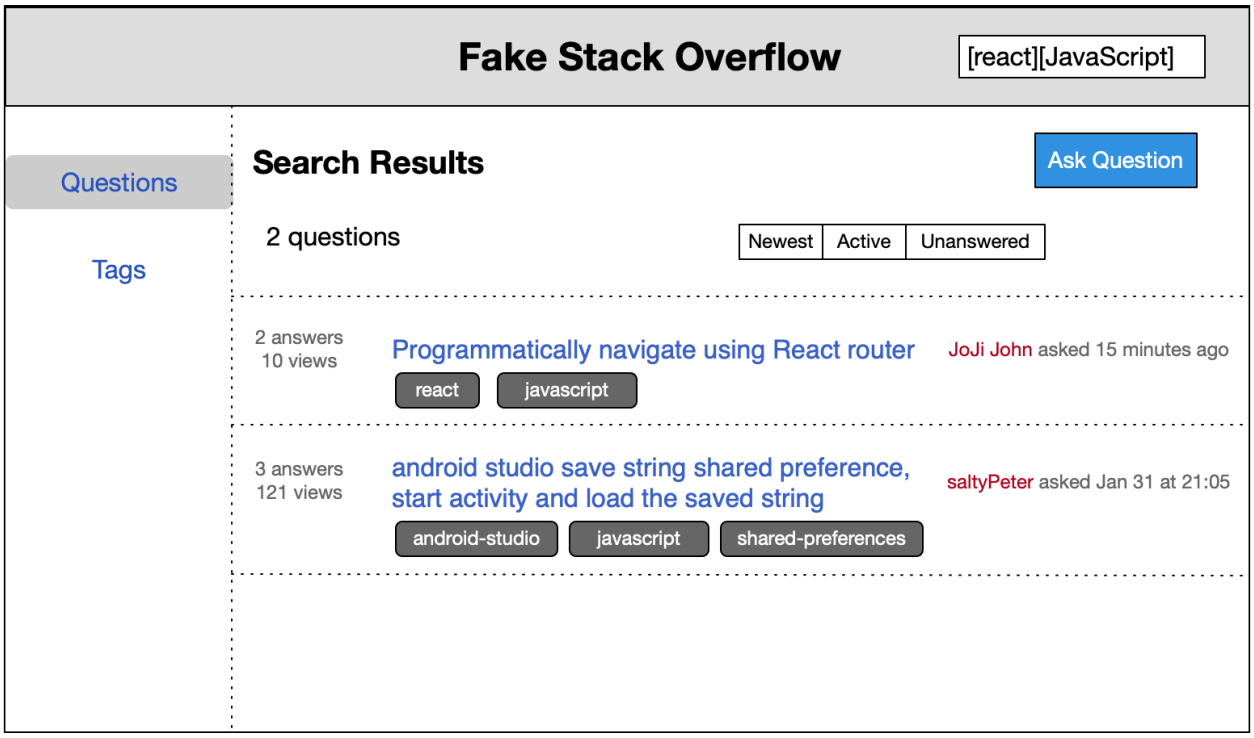

Figure 5

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.

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 do not fit on the page, then add a scroll bar.
- The answers should be displayed in ascending order of the day and time they were posted. In other words, display the answers that were posted most recently first.
- If the question text or an answer has hyperlinks, they should be displayed as such. Clicking on a hyperlink should open the target in a new tab or window. For example, in Figure 6, the first answer has two hyperlinks.
- 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.

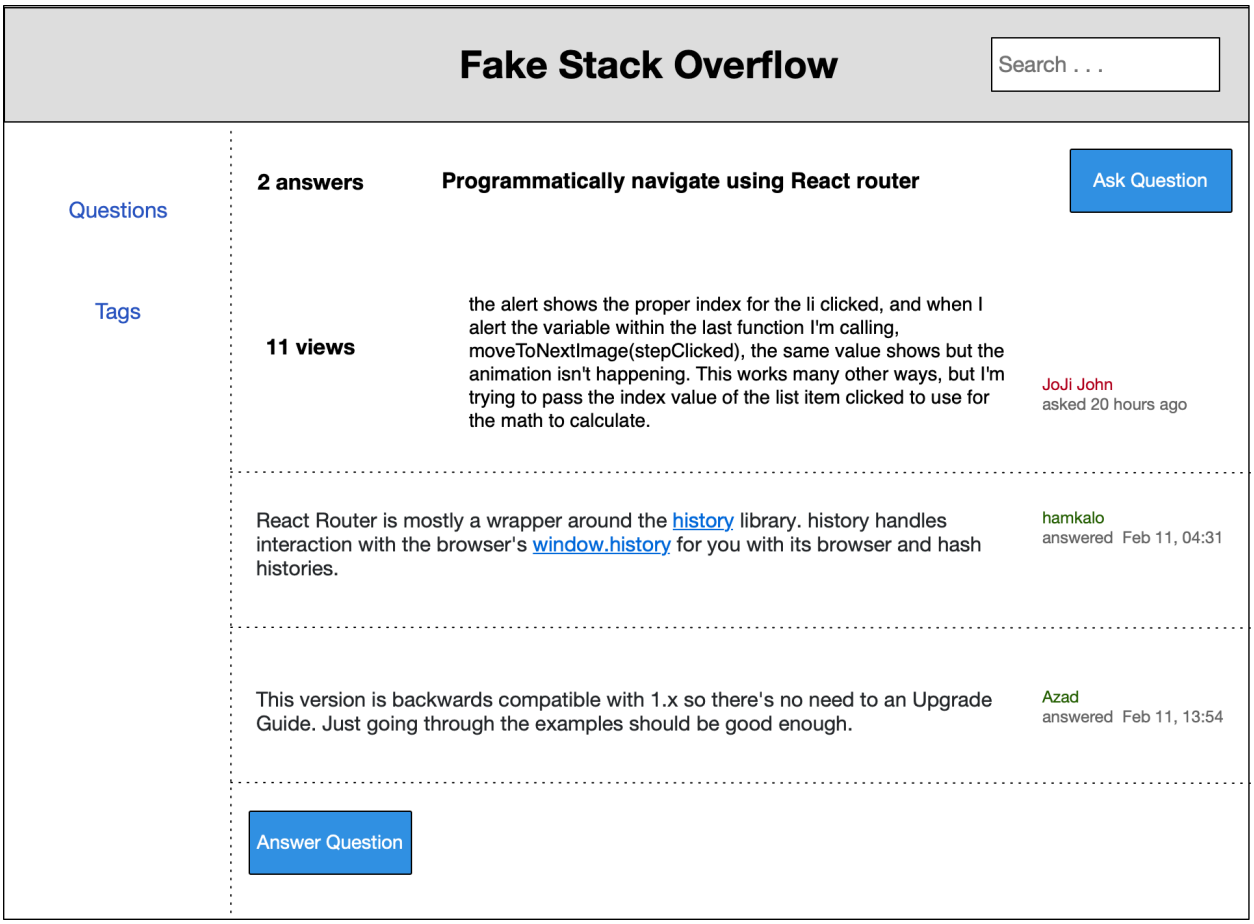

Figure 6

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.

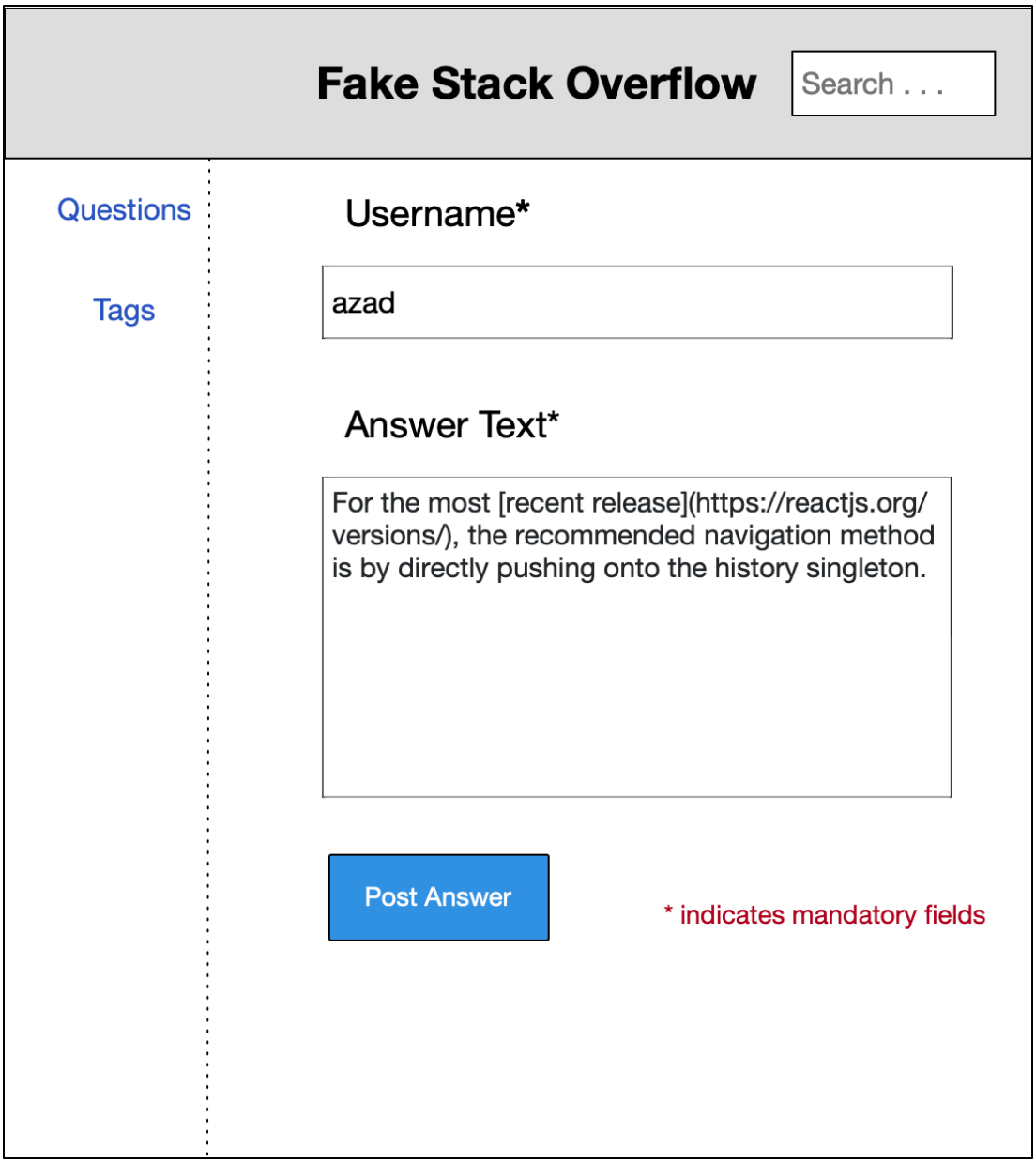

Figure 7

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.

A user is allowed to add hyperlinks in answer text. Hyperlinks are identified in the text if a user encloses the name of the hyperlink in [] and follows it up with the actual link in (). For example, in Figure 7, the answer text has a hyperlink “recent release”. The target of a hyperlink, that is, the stuff within () cannot be empty and must begin with “https://” or “http://”. Warn the user with an error message if this constraint is violated.

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.

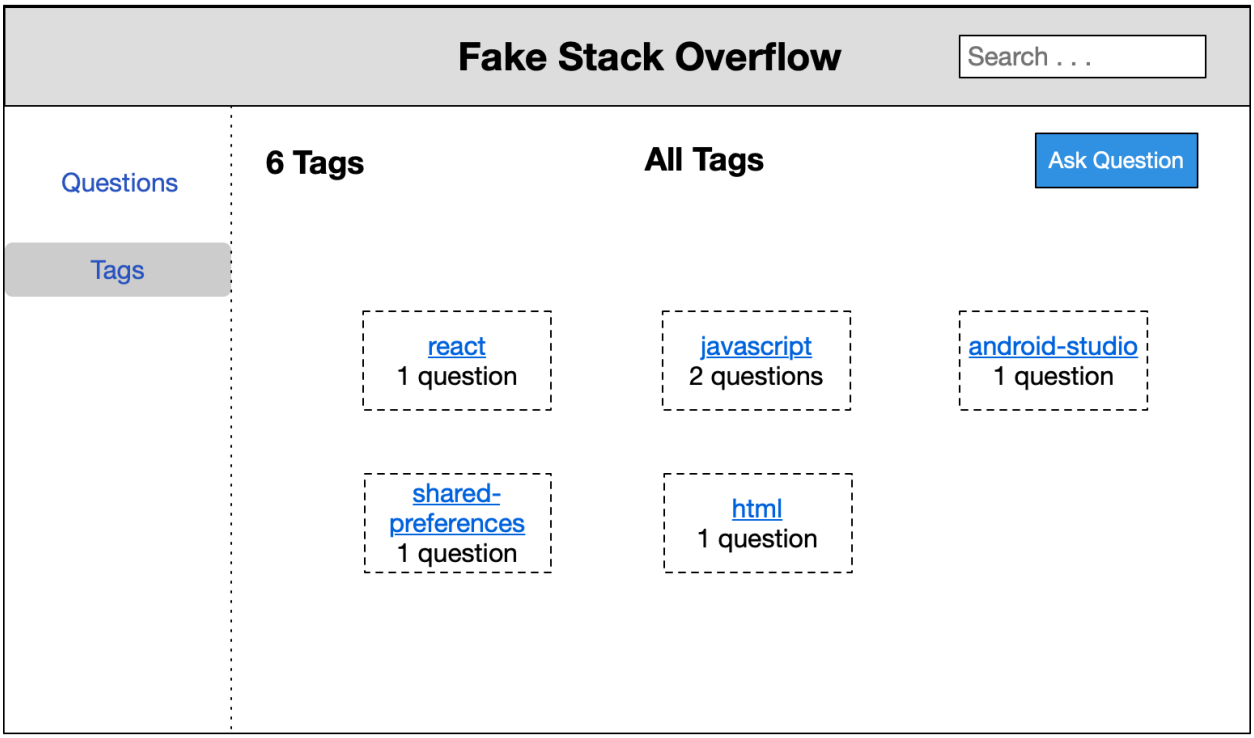
Fake Stack Overflow
Search . . .
Questions
Tags
6 Tags
All Tags
Ask Question
react
1 question
javascript
2 questions
android-studio
1 question
shared-preferences
1 question
html
1 question

Figure 8