Project 2: Tomcat-Based Markdown Editor

Overview

In Project 2, we will develop an online markdown editor using Apache Tomcat. Through this process we will learn how to develop a Web application using a "traditional" stack, in particular, MySQL and Apache Tomcat (Java servlet).
In later projects, we will use more "modern" Web development stack, such as MongoDB, Node.js, and Angular, but we want to make sure that everyone has an experience with developing Web applications on a more "traditional" stack. This will help students understand and appreciate why modern frameworks are structured as they are. All development for Project 2 will be done on the "tomcat" container that you created from "junghoo/tomcat" image in Project 1. Make sure that the container still starts and works fine by issuing the following command in a terminal window:

```
$ docker start -i tomcat
```

Part A: Learn Apache Tomcat

Before starting to code, first learn how we develop a Web application using Apache Tomcat by going over our tutorial:

- Developing a Web Application on Tomcat.

The tutorial teaches how you can develop a Web application using Java Servlet and JSP, and how you can package the set of files needed for your app into a single Web Application Archive (WAR) file for easy deployment on Tomcat.

Part B: Learn JDBC

In order to access MySQL data from a Java program, you will need to use JDBC (Java DataBase Connectivity) API. Go over the following tutorial to learn how to use JDBC to access MySQL in a Java program.

- A short tutorial on JDBC

Part C: Implement Markdown Editor and Previewer

Now that you have learned how to develop applications on Tomcat and MySQL, it is time to get your real work done. In Part C, you will need to implement an "online markdown editor" that allows users to save and edit blog posts written in markdown.

Markdown is a lightweight and intuitive markup language originally proposed by John Gruber in 2004 to help people "write using an easy-to-read, easy-to-write plain text format, and optionally convert it to structurally valid XHTML (or HTML)". Due to its simplicity, markdown has become the de-facto standard for writing readme files on many web sites (e.g., GitHub and BitBucket). In fact, this page was originally written in markdown and has been converted to HTML using Pandoc! Since we will use an "off-the-shelf" markdown parsing library in Java for this project, you don't have to learn markdown syntax, but in case you are interested, here is a 2-minute introduction to markdown syntax. For your reference, John Gruber's description of markdown syntax is available here (highly recommended if you want to learn more precise syntax) and the result of recent standardization efforts, referred to as CommonMark, is available here (extremely detailed and precise, yet boring to read).

"Pages" Within Your Application

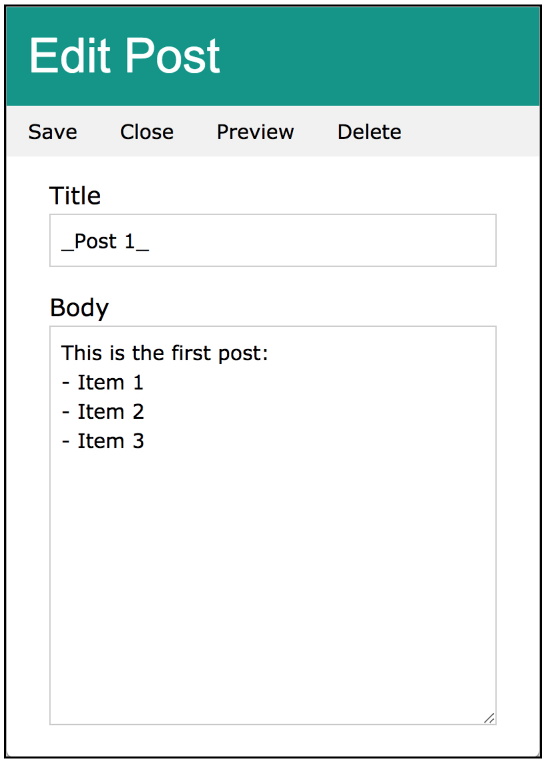
Your Web site should allows users to create a new post (written in markdown), preview the post (rendered in HTML), and manage existing posts. These tasks are supported through three main pages on your Web site: *edit, preview,* and *list* pages

Edit page

The "edit page" allows editing the title and body of a post.

- The page should contain two input boxes
  1. a *title* `<input>` box of `text` type. This text input element **must** have the ID attribute with value "title".
  2. a *body* `<textarea>`. This textarea element **must** have the ID attribute with value "body".
- The page should contain four buttons: *save, close, preview,* and *delete*. Once pressed,

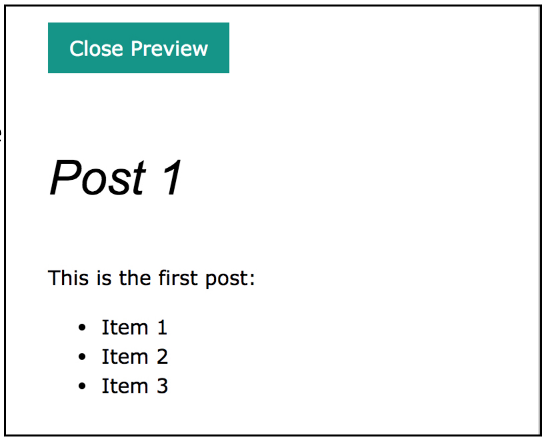
1. "save" button saves the content of the post to the database and goes to the "list page".
2. "close" button goes to the "list page" without saving the current content.
3. "preview" button goes to the "preview page" (without saving the current content).
4. "delete" button deletes the post from the database and goes to the "list page".



This is the first post:
- Item 1
- Item 2
- Item 3

Preview page

The "preview page" shows the HTML rendering of a post written in markdown. The page must have a "close" button. Once pressed, close button goes back to the "edit page" of the post.



This is the first post:

- Item 1
- Item 2
- Item 3

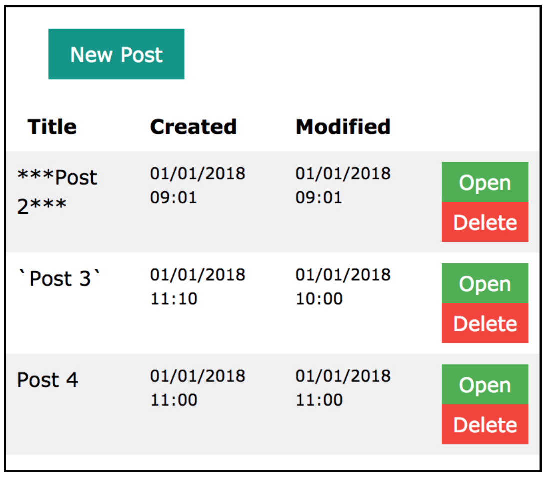
List page

The "list page" shows the list of all blog posts saved by the user. The posts in the list should be sorted by their "postid" (a unique integer assigned to a post) in the ascending order. Each item in the list must show:

1. title, creation, and modification dates of the post, and
2. two buttons: *open* and *delete*. Once pressed,

- “open” button goes to the “edit page” for the post.
- “delete” button deletes the post from the database and comes back to the list page.

The list page should also contain a “new post” button to allow users to create a new post. Once pressed, the button should lead to the “edit page” for a new post.



Title	Created	Modified	
Post 2	01/01/2018 09:01	01/01/2018 09:01	Open Delete
\`Post 3\`	01/01/2018 11:10	01/01/2018 10:00	Open Delete
Post 4	01/01/2018 11:00	01/01/2018 11:00	Open Delete

To help you better understand the function of the three pages on your site and their interaction model, we made an example demo site available at http://oak.cs.ucla.edu/classes/cs144/demos/project2/post?action=list&username=junghoo. Note that the demo is just for illustration purposes and may not follow our spec completely. When the behavior of the demo is different from our spec, you need to follow the spec. For security concerns, for example, we do not perform any update operation to our database, so even if you “delete” a post, it won’t disappear from the list. We also removed all CSS styling tags in the demo, so it looks ugly. Despite these, you can still get a basic idea of how the three pages should interact and function.

Note: The three “pages” that we described above may not necessarily be three separate HTML pages. Depending on your implementation, they may be implemented as one or more HTML pages, Java servlets and/or JSP pages. You may consider “three pages” as *three classes of page layouts* that your application should use.

Server-Side API

In developing your application, you should follow the following REST API. First, you should make your application available at the following path:

```
/editor/post
```

Depending on the HTTP method, additional parameters of the form `action=type&username=name&postid=num&title=title&body=body` may be included either in the path or in the body of the request. The path and the parameters are all case sensitive. The parameter “action” specifies one of five “actions” that your site has to take: *open*, *save*, *delete*, *preview,* and *list.* The other four parameters, username, postid, title, and body are (optional) parameters that the actions may need. Here is detailed descriptions of the five actions – their functions and required parameters:

open

- required parameters: username and postid
- function: return the "edit page" for the post with the given postid by the user
    - If postid == 0,
        - if title and body parameters have been passed, use the passed parameter values as the initial title and body values and return with status code 200 (OK)
        - otherwise, set missing title and/or body to empty string and return with status code 200 (OK)
    - If postid > 0,
        - if both title and body parameters have been passed, use the passed parameter values as the initial title and body values and return with status code 200 (OK)
        - otherwise
            - if (username, postid) row exists in the database, retrieve the title and body from the database and return with status code 200 (OK).
            - otherwise, return with status code 404 (Not found).

save

- required parameters: username, postid, title, and body
- function: save the post into the database and go to the "list page" for the user
    - if postid == 0
        - assign a new postid, save the content as a "new post", and return with status code 200 (OK)
    - if postid > 0
        - if (username, postid) row exists in the database, update the row with the new title, body, and modification date and return with status code 200 (OK).
        - if (username, postid) row does not exist, do not modify the database and return with status code 404 (Not Found).

delete

- required parameters: username and postid
- function: delete the corresponding post and "redirect" to the "list page"
    - if (username, postid) row exists in the database, delete the row, and return with status code 200 (OK). In the response body, include the HTML redirection meta tag similar to the following, so that the browser will automatically redirect to the "list page".

    ```
    <head>
      <meta http-equiv="Refresh" content="0; URL=https://redirected-url.com/">
    </head>
    ```

    - if (username, postid) row does not exist in the database, return with status code 404 (Not Found).

To learn different mechanisms to redirect the browser automatically to a different page, read the MDN document on HTTP redirections.

preview

- required parameters: username, postid, title, and body
- function: return the "preview page" with the html rendering of the given title and body
    - return with status code 200 (OK).

list

- required parameters: username
- function: return the "list page" for the user
    - if username exists in the database, return with status code 200 (OK).
    - if username does not exist in the database, return with status code 404 (Not Found).

NOTE:

1. Among the five actions, you should take requests via both GET and POST methods for *open*, *preview*, and *list*, while you should allow **only POST method for *save* and *delete*.** This is because *save* and *delete* leave significant side effects on the server.

2. When the server cannot handle a request because the request is “invalid” – for example, the request does not include a required parameter – the server must return with a 4XX response status code, such as “400 (Bad request)” or “404 (Not found)”. **The server should never return a 5XX response status code because of an invalid request.** Look up this page to see the list of HTTP response status codes and their meaning. Refer to the `HttpServletResponse` API to learn how to set the HTTP status code in Servlet.
3. To make the scope of this project manageable, you may assume that the title and body parameters will not contain any HTML tags or JavaScript. You do not need to worry about problems arising from them. In real projects, though, you will have to worry about these issues since they may lead to serious security vulnerabilities.

NOTE: During our grading, we won’t be doing extensive check for security related issues, but there are two things that we may test:

1. Your implementation should do the basic input validation. For example, we do expect that students make sure that postid is integer.
2. Your implementation should use `preparedStatement` in your SQL queries as a basic protection against SQL injection attack. As we mentioned in our JDBC tutorial, we expect students to follow this industry-wise convention, which cannot be overemphasized as a minimal security measure.

Whatever the user input, it will be important that your implementation won’t flake out (like throwing Java exception messages as the output). All requests need to be handled gracefully, however malformed they may be.

Note on Multi-Threading:

Tomcat creates one thread per each request and concurrent data manipulation can be taken care of by MySQL. This makes server-side programming easy since the programmer doesn’t have to worry about concurrency. But in order to do things this way, you need to keep two things in mind:

1. JDBC API is thread-safe only if each thread uses its own connection. If multiple threads share the same connection, the behavior is unpredictable. Create one DB connection in each servlet request handler, such as `doGet()` and close it before you return from the handler.
2. Any servlet request handler **must not use global states**. Any global state that needs to be shared between multiple requests **must be stored and managed by MySQL**. In particular,

any servlet request handler, such as `doGet()` must not reference a class instance variable. It must reference only local variables.

Note on Connection Pooling:

You may think that creating one DB connection per request will lead to too much overhead and wonder whether there is any way to "share" the same DB connection among multiple requests. The proper way to do it is to use *connection pooling*, but this is beyond the scope of this project. Connection pooling is something that must be used in a production environment, but you MUST NOT use connection pooling in this project since it won't work on the grader's machine.

Database Schema

All blog posts that are saved by the users must be stored as a row a MySQL table with the following schema:

```
Posts(
    username      VARCHAR(40),
    postid        INTEGER,
    title         VARCHAR(100),
    body          TEXT,
    modified      TIMESTAMP DEFAULT '2000-01-01 00:00:00',
    created       TIMESTAMP DEFAULT '2000-01-01 00:00:00',
    PRIMARY KEY(username, postid)
)
```

Hopefully, the meaning of each column would be clear from our earlier discussion. MySQL schema definition is case sensitive, so be careful with the case of your schema definition.

Create the table within "CS144" database as the user "cs144". The first blog post of every user must start with postid=1 and their subsequent posts must be assigned to a linearly increasing postid. Therefore, blog posts by different users may share the same postid, but postid is unique within a single user. Populate the `Posts` table with the following six initial tuples:

```
('user_XYRSAF', 1, 'Post 1 by XYRSAF', 'Article1 written by XYRSAF', '2020-01-03 1
('user_XYRSAF', 2, 'Post 2 by XYRSAF', 'Article2 written by XYRSAF', '2020-01-04 1
('user_XYRSAF', 3, 'Post 3 by XYRSAF', 'Article3 written by XYRSAF', '2020-01-05 1
('user_XYRSAF', 4, 'Post 4 by XYRSAF', 'Article4 written by XYRSAF', '2020-01-06 1
('user_ACHERW', 1, 'Post 1 by ACHERW', 'Article1 written by ACHERW', '2020-01-03 1
('user_ACHERW', 2, 'Post 2 by ACHERW', 'Article2 written by ACHERW', '2020-01-04 1
```

In case you find it helpful, here is a text file that contains the six tuples in a format that you can simply "cut-and-paste" into a SQL INSERT statement.

In implementing your app, you may need to create tables other than the "Posts" table to store other information needed for your app persistently.

Skeleton Code

To help you get started, we provide skeleton sample code for this project in project2.zip, which has the following set of files:

```
project2.zip
 +- build.gradle
 +- create.sql
 +- deploy.sh
 +- src
    +- main
        +- java
        |   +- Editor.java
        +- webapp
            +- edit.jsp
            +- WEB-INF
                +- web.xml
```

1. *create.sql*: This file has has a sequence of SQL commands that create "Posts" table in the CS144 database and load the six initial tuples into the table. If you need to create any additional tables for your app, you will need to add the SQL commands to create them in this file.
2. *src/main/java/Editor.java*: This is a skeleton Java code for servlet implementation. The provided code simply dispatches the request to *edit.jsp* page, so you will need to implement your application logic as part of the `Editor` class, such as connecting to, retrieving from, and updating MySQL database server, and dispatching the user's request to an appropriate JSP page.
3. *src/main/webapp/edit.jsp*: This is a simple JSP page that contains HTML elements for implementing the "edit page". You will need to update this page according to our earlier specification.
4. *src/main/webapp/WEB-INF/web.xml*: This is the Tomcat deployment descriptor file prepared for this project. Currently, it sends any request to `/editor` to the Java class `Editor`.
5. *build.gradle* and *deploy.sh*: These are gradle "build script" and the app deployment script

that you can use to build your WAR file and deploy it to Tomcat. We will explain these scripts in more detail in the next section.

Gradle build script

As you learned from our tutorial on Tomcat, a Web application needs many files that should be carefully prepared and packaged. To help you focus on coding, not packaging, we included a "gradle build script" that takes care of compilation and packaging. To see how it works, unzip project2.zip into a folder, `cd` to the folder and run the following command:

```
$ gradle assemble
```

This will compile all your Java source files in the `src/main/java/` directory (currently, just "Editor.java"), package them together with everything under `src/main/webapp/` directory according to the spec, and create a war file at `build/libs/editor.war`. Once created, you can copy the war file to `$CATALINA_BASE/webapps`:

```
$ cp build/libs/editor.war $CATALINA_BASE/webapps
```

to deploy it to the Tomcat server. Copy the file, wait for a few seconds (so that Tomcat detects the new war file and sets it up), and access http://localhost:8888/editor/post?action=list&username=test from a browser on your host. You will get a (non-functional) version of the "edit page" that has been generated from the `edit.jsp` page.

As long as you can use our provided gradle build script to compile your code and create the war file, it is okay for you to know nothing about how the gradle build script works. But if you want to learn more, read one of many online tutorials on Gradle.

Your job now is to write Java code in `Editor.java` and (optionally) add new JSP, HTML pages to `src/main/webapp` in order to implement Online Markdown Editor. If you are not sure how the code in `Editor.java` works or how to write a JSP page, go over our Tomcat Application Development Tutorial. If you are not sure how to access a MySQL database with a Java code, go over our JDBC tutorial.

Note: You may find it helpful that what is written to `System.out/err` in Tomcat servlet is (1) printed on console and (2) written to the `catalina.out` log filelocated in `$CATALINA_BASE/logs/`.

Commonmark Java Library

To implement the “preview page”, you need to “compile” a markdown-formatted input into an HTML-formatted output. For this, you can use the commonmark Java library. We have already downloaded the library and made it available in our “tomcat” container. Go over the the README.me file on the library page to learn how to use the library. Roughly, the following Java code

```
import org.commonmark.node.*;
import org.commonmark.parser.Parser;
import org.commonmark.renderer.html.HtmlRenderer;
...
Parser parser = Parser.builder().build();
HtmlRenderer renderer = HtmlRenderer.builder().build();

String markdown = "This is *CS144*";
String html = renderer.render(parser.parse(markdown));
/* html has the string "<p>This is <em>CS144</em></p>\n" */
```

will “compile” the markdown text `This is *CS144*` into HTML text `<p>This is <em>CS144</em></p>`.

Deployment Script `deploy.sh`

In project2.zip, we also included our “deployment bash script”, deployment.sh, that automates the entire application deployment process. In particular, it (1) creates tables for our application in MySQL (2) build the “editor.war” file and (3) deploy it to the Tomcat server. Once you finish developing your code, make sure that simply running

```
$ ./deploy.sh
```

properly sets up your application. This script will be used to deploy your application during our grading, so it is ***extremely important*** to make sure that this script works without any error.

Your Final Submission

Your project must be submitted electronically before the deadline via GradeScope.

What to Submit

You should submit a single file named project2.zip that roughly has the following contents:

```
project2.zip
 +- README.txt (optional)
 +- deploy.sh
 +- build.gradle
 +- create.sql
 +- src
    +- main
        +- java
        |   +- Editor.java (and other java files that you wrote)
        +- webapp
            +- edit.jsp (and other jsp, css, and html files that you added)
            +- WEB-INF
                +- web.xml
```

We have already explained what most of the above files are earlier, but your submission should include the following additional files:

1. README.txt: This optional file should include any information that you need to communicate to the grader.
2. Please ensure that your `create.sql` file inserts the six initial tuples into `Posts` table.

Please ensure that your submission is packaged correctly with all required files, but not compiled Java class or WAR files. Make sure that each file is correctly named (including its case). Perhaps, the most important requirement of your submission is that ***the grader should be able to deploy a fully-functional version of your Web site just by running `./deploy.sh`*** after unzipping your submission. **You may get as small as zero points if the grader encounters an error due to incorrect packaging, missing files, and failure to follow our exact spec.**

To help you package your submission zip file, we prepared "a packaging script" p2_package. After downloading the script in the directory that has your Project 2 file, set its executable permission and run it as follows:

```
$ wget http://oak.cs.ucla.edu/classes/cs144/project1/p2_package
$ chmod +x ./p2_package
$ ./p2_package
  adding: deploy.sh (deflated 36%)
  adding: create.sql (deflated 69%)
  adding: readme.me (deflated 43%)
  adding: group.txt (stored 0%)
  adding: src/ (stored 0%)
  adding: src/main/ (stored 0%)
  adding: src/main/webapp/ (stored 0%)
  adding: src/main/webapp/edit.jsp (deflated 62%)
  adding: src/main/webapp/preview.jsp (deflated 61%)
  adding: src/main/webapp/list.jsp (deflated 65%)
  adding: src/main/webapp/error.jsp (deflated 60%)
  adding: src/main/webapp/WEB-INF/ (stored 0%)
  adding: src/main/webapp/WEB-INF/web.xml (deflated 56%)
  adding: src/main/java/ (stored 0%)
  adding: src/main/java/Editor.java (deflated 79%)
  adding: build.gradle (deflated 8%)
[SUCCESS] Created '/home/cs144/p2/project2.zip'
```

When executed, our packaging script will collect all necessary (and optional) files located in the current directory and create the `project2.zip` file according to our specification that can be submitted to GradeScope.