

Project 3: reCAPTCHA, HTTPS, PreparedStatement, Stored Procedure, XML Parsing

Due May 14 by 11:45pm **Points** 100

▼ Overview

Improve the security of the Fabflix website

A real business needs to treat security issues seriously. In the tasks, you will improve the security of your website in Project 2 in various ways:

- reCAPTCHA enhances security by preventing bots and only allowing real human users.
 - HTTPS enhances security by securing the process when the client sends sensitive information to the server.
 - Using prepared statements can help prevent a widespread SQL injection attack.
 - Encrypting the password improves security in case your database gets attacked. The hacker won't be able to get compassionate information, such as a password.
-



Expand the movie dataset

In Project 1, we provided some movie data to get you started. However, new movies can be published every day. Fabflix has to be able to expand dynamically.

- Administrators should be able to add movies and stars from the web app without interacting with the database directly.
 - A larger dataset from a third party could be parsed and imported into the database on the backend.
-

▼ Task 1: Adding reCAPTCHA

Improve the user-authentication process of Project 2 by adding a module that verifies real users and blocks bots.

1. Go to [recaptcha-example](https://github.com/UCI-Chenli-teaching/cs122b-fall21-project3-recaptcha-example) , and follow the README instructions on the Github Page for the code.
2. Get a [reCAPTCHA](https://www.google.com/recaptcha/intro/v3.html)  from Google. Add the `public` IP of your AWS instance and `localhost` to the domain. You will get a *secret key* and a *site key*.
3. Test the page <http://localhost:8080/cs122b-project3-recaptcha-example/> to see if reCAPTCHA works.
4. Apply the same logic to your project 2 by making the necessary changes on the login frontend and backend logic.

Note that if your AWS instance is restarted, the public IP will change, and you must also add the new IP to the Google ReCaptcha console.

▼ Task 2: Adding HTTPS

HTTPS makes your website substantially more secure from hacking and other security breaches. HTTPS ensures the client can safely send sensitive data to your server without the risk of leaking the data in the transfer process.

Step 1: On your AWS instance, you can use the following command to create a keystore to be used by Tomcat. Remember what password you use for the keystore. Our example uses "changeit".

In the following steps, "/etc/tomcat10" is an example of your tomcat installation directory on AWS. You need to replace it with your own directory.

```
sudo keytool -genkey -alias fabflix -keyalg RSA -keystore /var/lib/tomcat10/keystore
```

Step 2: Uncomment the `Connector` tag in `/etc/tomcat10/server.xml`, which has `port="8443"` and defines an SSL HTTP/1.1 connector. Please modify and use the following configuration:

```
<Connector port="8443" protocol="org.apache.coyote.http11.Http11NioProtocol" maxThreads="150" SSLEnabled="true">
  <UpgradeProtocol className="org.apache.coyote.http2.Http2Protocol" />
  <SSLHostConfig>
    <Certificate certificateKeystoreFile="/var/lib/tomcat10/keystore"
      certificateKeystorePassword="changeit"
      type="RSA" />
  </SSLHostConfig>
</Connector>
```

Step 3: Restart Tomcat:

Go to your Tomcat directory. Do `sudo systemctl restart tomcat10`

At this point, [https://\[YOUR_IP\]:8443/manager/html](https://[YOUR_IP]:8443/manager/html) should also be an entry point to Tomcat. It uses secure HTTP (HTTPS) and a different port, 8443, by default. Make sure to open port 8443 on your AWS instance.

You may get a warning in your browser when you go to the [https://\[YOUR_IP\]:8443/manager/html](https://[YOUR_IP]:8443/manager/html), because we use a self-generated certificate (generated in step 1), and self-generated certificates, are not considered trusted. In most cases, you should be able to bypass this warning and visit your website. You may still see a warning that says the website is not secure next to the address bar in the browser, which you can ignore.

If you cannot bypass the warning and cannot visit your page (most likely on Mac), you have to allow your browser to trust the self-signed certificate. Follow the steps under the 'Add Certificate to Trusted Root Authority' section on this link: [Trust self-signed certificate on MAC](https://www.andrewconnell.com/blog/updated-creating-and-trusting-self-signed-certs-on-macos-and-chrome/). [➡ \(https://www.andrewconnell.com/blog/updated-creating-and-trusting-self-signed-certs-on-macos-and-chrome/\)](https://www.andrewconnell.com/blog/updated-creating-and-trusting-self-signed-certs-on-macos-and-chrome/)


Step 4: Limit a Tomcat application to HTTPS only. Currently, the http://YOUR_IP:8080/manager/html entry (which uses HTTP) is still open. To disable HTTP and enable HTTPS only, change your cs122b-project3-recaptcha-example application's web.xml by adding the following code before the closing "</web-app>" tag:

```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>HTTPSOnly</web-resource-name>
    <url-pattern>/*</url-pattern>
  </web-resource-collection>
  <user-data-constraint>
    <transport-guarantee>CONFIDENTIAL</transport-guarantee>
  </user-data-constraint>
</security-constraint>
```

This setting will force all HTTP requests to this application to get redirected to the corresponding HTTPS URL. Reload the application to see if the redirection works for the URL http://YOUR_IP:8080/cs122b-project3-recaptcha-example/.

Step 5: Make necessary changes to your project 2 to use HTTPS always with HTTP disabled.

▼ Task 3: Use PreparedStatement


Until now, we generated the SQL query by concatenating the parameter value with a template string so that it is passed to the database to be compiled and executed. This string manipulation approach leaves us very vulnerable to [SQL Injection Attack](https://en.wikipedia.org/wiki/SQL_injection)  (https://en.wikipedia.org/wiki/SQL_injection). We can prevent this by using `PreparedStatement`. `PreparedStatement` pre-compiles the SQL template, and when the database executes the query, it treats the parameters as values without messing with the SQL query itself.


You can learn `PreparedStatement` at [this reference](https://docs.oracle.com/javase/tutorial/jdbc/basics/prepared.html)  (<https://docs.oracle.com/javase/tutorial/jdbc/basics/prepared.html>).

Change all the queries in your project 2 involving specifying any parameter from user input to use `PreparedStatement`.

▼ Task 4: Use Encrypted Password

This task will show you how to update the plain text password to encrypted password and store it in the database. The client (frontend) will still pass the plain text password to the server, (this is okay because we have HTTPS), the server will then compare the plain text password with the encrypted password stored in the database.

The current `moviedb.customers` table stores plain text password. We have provided you a Java program you can directly run to update them with the encrypted password. We use [Java Simplified Encryption](http://www.jasypt.org/)  (<http://www.jasypt.org/>) library to do the encryption.

1. Go to [project3-encryption-example](https://github.com/UCI-Chenli-teaching/cs122b-project3-encryption-example)  (<https://github.com/UCI-Chenli-teaching/cs122b-project3-encryption-example>) to download and deploy the project.
 2. Import the project into IntelliJ on your local machine, and run `UpdateSecurePassword.java`. This program will read all the passwords from the existing `moviedb.customers` table, encrypt them, and update the table with the encrypted passwords.
 3. On AWS, also run `UpdateSecurePassword.java` to update the database. Note that you should only run this program **once**. If you need to re-run it, you need to re-populate the customers table with the plain text password.
 4. `VerifyPassword.java` shows you how to verify the user's email and password against the encrypted password stored in the database.
 5. Change your login modules in project 2 to verify user email/password based on the encrypted password.
-

▼ Task 5: Implementing a Dashboard using Stored Procedure

Step 1: To store information about Fabflix's employees, create a new table called **employees** with the following attributes:

```
email varchar(50) primary key
password varchar(20) not null
fullname varchar(100)
```

Step 2: Insert an employee with `classta@email.edu`, `classta`, and `TA CS122B` as its email, password, and fullname respectively. Use the same technique in the encrypted password section to encrypt the password here.

Step 3: Setup an entry point https://YOUR_IP:8443/fabflix/_dashboard (note the underscore!) that uses HTTPS. It allows employees to login with a valid email and password to access a dashboard.

Step 4: Allow a logged-in employee to do the following operations using this dashboard.

- Inserting a new star into the database. Star name is required, birth year is optional.
- Providing the metadata of the database; in particular, show the name of each table and, for each table, each attribute and its type.

Step 5: Write a stored procedure called `add_movie`. Its arguments include all the required fields of the movie, a *single* star (star name) and a *single* genre (genre name). (The reason we don't want to pass multiple stars and genres is that the current MySQL implementation doesn't allow stored procedures to take an array argument.)

Step 6: In the dashboard implemented in Step 4, add one more feature that allows an employee to add a **new** movie, including a single star (new or existing) and a single genre (new or existing). The UI should allow the employee to provide the information, and the backend should call the stored procedure `add_movie` created above. Duplicate movies should not be added. It is not required to update an existing movie.

Step 7: Include your stored procedure in a sql file, `'stored-procedure.sql'`, in the submission.

Remarks:

1. The added new stars and movies should be accessible and searchable from the project 2 webpage.
2. When adding a new star, if an existing star has the same name and birth year, you can treat them as a new person and create a new record.
3. When adding a new movie:

- If the star or genre already exists, you should link them to the movie. In the case of two star records with the same name, you can link any of them.
 - If the star or genre doesn't exist, you should create it and then link it to the movie. (star birth year is optional and can be default null)
 - All necessary `stars_in_movies` and `genres_in_movies` records should also be created. The procedure should output informative status messages to the user as it performs the task (see demo instruction for details.)
 - A movie is identified by (title, year, director). For example, a movie with same title and year, but not same director is considered as a different movie.
 - If the movie already exists, you should show a corresponding message, and no changes to the database are made.
4. The user does not provide an ID for star, movie, or genre. Those ids should be managed by your stored procedure. To implement this in your stored procedure, you could:
- Use `select max(id) from TABLE` to get the "largest id" according to the alphabetic order, then "parse" it and use it to generate a new unique id;
 - Use a new "helper table" to store the next available integer and increment it each time for the next unique id; Changing the our existing table schema to implement this feature is not allowed.

▼ Task 6: Importing large XML data files into the Fabflix database

The goal of this task is to use a new source of data to increase the content size of the Fabflix movie database.

- You can find the raw version of this new data source in [this attachment](https://canvas.eee.uci.edu/courses/54347/files/22162681/download) (<https://canvas.eee.uci.edu/courses/54347/files/22162681/download>). The schemas of these files are explained on [this page](http://infolab.stanford.edu/pub/movies/dtd.html) (<http://infolab.stanford.edu/pub/movies/dtd.html>).
- For example a tag `<dirid />` will be defined as `<!ELEMENT dirid >` in the DTD file. The comments, along with the DTD declarations, are beneficial to identify what field is used for. Try to find relationships between the XML files and correlate the entities together with the help of the DTD files.
- Particularly, you are required to parse the files `mains243.xml` and `casts124.xml` to add new `movies`, `stars`, and `genres` to the Fabflix database (i.e., on top of what you already have in the database). If needed, the `stars_in_movies` and `genres_in_movies` tables should also be updated accordingly. Also, for each XML file in this package, there is a DTD file to be used to understand/validate the structure of that XML file.

- Please note that in this project, it is expected that, if needed, you make reasonable decisions on how to use this new data source. The files are not clean, and you may find inconsistencies in them. In such cases, do not be surprised. Make reasonable decisions. Such issues are very common in many real applications. You can use the following high-level description regarding mapping the XML tag attributes to the Fabflix database schema given in Project 1.
- As shown in Figure 1, the root tag in the file `mains243.xml` is `<movies>`, which consists of several instances of the tag `<directorfilms>`. Each `<directorfilms>` contains the information of a director, along with his/her movies, each movie being an instance of the tag `<film>`. It is clear where to find the values of all the columns of the movies table. Furthermore, each `<film>` has a list of assigned categories in a `<cats>` tag (i.e., each category as a tag). The value of the `<cat>` tag corresponds to the genre name in the `genres` table of the Fabflix database. If a `<cat>` does not exist in the `genres` table, a new genre must be added, and the `genres_in_movies` table must be updated accordingly.

```

<movies>
  <directorfilms>
    <director> ... </director>
    <films>
      <film> ... </film>
      <film> ... </film>
      ...
      <film> ... </film>
    </films>
  </directorfilms>
  <directorfilms> ... </directorfilms>
  ...
  <directorfilms> ... </directorfilms>
</movies>

```

Figure 1

- For updating the `stars` and `stars_in_movies` tables, you should use the content of the file `casts124.xml`, which is shown in a high-level view in Figure 2.

```

<casts>
  <dirfilms>
    ...
    <filmc>
      <m> ... </m>
      <m> ... </m>
      ...
      <m> ... </m>
    </filmc>
    <filmc> ... </filmc>
    ...
    <filmc> ... </filmc>
  </dirfilms>
  ...
  <dirfilms> ... </dirfilms>
</casts>

```

Figure 2

- Similar to the `<movies>` tag, the content of the `<casts>` tag, which is the root of the file `casts124.xml`, is also grouped by the movie directors. You can find the information on the star-movie connections in the `<m>` tag instances. In this tag, the film is identified using the `<f>` tag, which links to the `<fid>` tag of the `<film>` tag in the main file. Tag `<m>` also gives you the `stagename` of the actor of this relation in a tag called `<a>`. The `stagename` helps you connect this `stars_in_movies` record to the right star record. The stars to be added to the `stars` table are parsed from the file `actors63.xml`.
- Although there are many libraries for parsing XML, there are two main approaches to this problem: (1) SAX: event-based XML parsing, and (2) DOM: using a tree-based object model for parsing XML. In SAX, the library **moves** through the XML tags and makes calls to *event-handler functions* that the developer implements. In DOM, the developer can move on a tree of objects made available by the library, each corresponding to an *entity* in the XML tree (e.g., a tag). You can choose between either SAX or DOM approaches to parse the document. For examples, look at <https://github.com/UCI-Chenli-teaching/cs122b-spring20-project3-SAXParser-example>, <https://github.com/UCI-Chenli-teaching/cs122b-spring20-project3-recaptcha-example>, [SAXParser-example](https://github.com/UCI-Chenli-teaching/cs122b-project3-SAXParser-example), <https://github.com/UCI-Chenli-teaching/cs122b-project3-SAXParser-example>, and [DomParser-example](https://github.com/UCI-Chenli-teaching/cs122b-project3-DomParser-example), <https://github.com/UCI-Chenli-teaching/cs122b-project3-DomParser-example>.

Requirements:


- **XML parsing:** You should write a Java program to parse the XML files and insert the new data into the existing moviedb database. The new movie data should be searchable on the Project 2 search page.
- **Performance Tuning:** A naive implementation of your program is expected to be slow. Come up with at least two optimization techniques to reduce the running time of your program. (Note that `setting auto-commit off` and using `PreparedStatement` can **NOT** be counted as optimization approaches for this task.) Write an itemized, brief report accessible to describe your optimizations and the corresponding time reductions. Put this optimization report in your README file.

Remarks:

- In case of inconsistencies between the provided data with respect to the relational schema, make sure your program does not crash. The inconsistent data must be reported to the user (print the element name and node value) and your program should resume processing.
- Use NULL for missing values when inserting records into the database.
- For inconsistent values with the schema (like non-integer values for volume or roman numbers) you can treat them as NULL.

- For entries with multiple ISBN/publishers/etc, only use one of them.
 - Keep track of duplicate entries like multiple occurrences of same movies/actors and insert them only once.
 - Make sure the encoding of your parser is ISO-8859-1. You can ignore any special character that this encoding cannot handle.
-

▼ Extra Credit: Domain Registration

If you are tired of using an IP address to visit your AWS instance and want to make it "real," you can register a domain name from a registrar company such as [Godaddy](https://www.godaddy.com/)  [\(https://www.godaddy.com/\)](https://www.godaddy.com/). Then use the AWS instance to do the hosting. Now we are open for real business!

We suggest that you assign an Elastic IP before you connect your domain.

We will not answer domain or Elastic IP questions since it's extra credit.

▼ Different Domain

- Task 6 involves creating a dashboard to display metadata, insert movie/star/genre, etc. Implementing this with your custom domain is worth 2 points out of 5 extra credits. You must ensure you insert at least one 1-to-1 and 1-to-many relationship.
 - Task 7 is to insert large XML files into the Fabflix movie database. To get the remaining 3 extra credit points for this project, you must generate or create XML files containing your custom domain's data. If you use the movie data we provided for part 5, you won't get the 3 points.
-

▼ Rubric

General requirements: Same as project 2.

Feature	Item	Points
reCAPTCHA	reCAPTCHA works properly, with appropriate error messages.	5
HTTPS	HTTPS works properly.	5
	Redirect all HTTP traffic to HTTPS (force use HTTPS).	5
Password Encryption	Encrypt password in DB, and login should verify user input with encrypted password	5

Prepared Statement	Use prepared statements for all queries in the codebase and included in README	5
Employee Dashboard	Login to dashboard page using employee credentials	4
	reCAPTCHA works properly, with appropriate error messages.	4
	Show database metadata	4
	Add a new star successfully. Stars linked to movie should be searchable/accessible on the webpage	4
	Add a new movie successfully with a new star, a new genre. Should be searchable/accessible on the webpage	4
	Add a new movie successfully with existing star, existing genre. Should be searchable/accessible on the webpage	4
	Show error message when adding an existing movie (same title, year and director)	4
	Show all proper success messages and error messages	4
XML Parsing	Parsing main.xml successfully, movie title and director information shown correctly. Should be searchable/accessible on the webpage	8
	Parsing actors.xml successfully, birthYear of actors can be shown correctly. Stars linked to movie should be searchable/accessible on the webpage	8
	Parsing casts.xml successfully. Actors of movies should be searchable/accessible on the webpage	12
	Report inconsistency data to the user and included in README.	5
	Two parsing time optimization strategies, compared with the naive approach, included in README	10

	("set auto-commit off" and "using PreparedStatement" can NOT be counted as optimization strategies for this task.)	
Extra Credit	Register a domain name for Fabflix	5
Total		100 + 5

▼ Demonstration

In summary, we require 2 things for the demo:

1. A screen recording demo video showing your web application working on AWS.
2. Leave an AWS instance running for a week.

Here are the details:

Screen recording demo video

- The entire demo should be on AWS. **NO** local demo allowed.
- You need a terminal connects to AWS, and a web browser open to tomcat manager page on AWS (<https://<AWS public IP>:8443/manager/html>).
- Use screen recording tools (e.g, QuickTime Player for macOS and Open Broadcaster Software for Windows, or other equivalents)
- Please keep the length of the video within 20 minutes. **Your XML parsing cannot exceed 15 minutes.** However, do **NOT** edit the video in any way. Otherwise it could be treated as cheating.
- Upload the video to Youtube or other public media hosting website, then include your video URL (publicly accessible) in README.md, submit on GitHub.
- Preparation before the demo:
 - ssh onto AWS instance.
 - on AWS instance, prepare the MySQL database `moviedb`, load data (movie-data.sql) properly, **remove any data you added before (from XML or from dashboard).**
 - remove your git repo from AWS instance, demo should start with a fresh cloned repo.
 - **prepare the xml files in a separate folder.**
 - make sure MySQL and Tomcat are running on AWS instance.

- Demo on AWS instance:
 - **Commit check:**
 1. git clone your git repo to AWS instance: ``git clone <repo url>``. This step is to make sure you have a clean repo.
 2. cd into your repo and run ``git log``, show your latest commit.
 3. fail to show commit check will result in an instant 0 of project.
 - **Show database and data on AWS MySQL:**
 1. show your data counts:

```
mysql -u mytestuser -p -e "use moviedb;select count(*) from stars;select count(*) from movies;"
```

Should not contain the data from XML yet.
 2. show your encrypted passwords:

```
mysql -u mytestuser -p -e "use moviedb;select * from customers;"
```
 3. show you do not have movies that will be entered:

```
mysql -u mytestuser -p -e "use moviedb;select * from movies where title like 'movie%';"
```
 - **Deploy your web application on AWS instance:**
 1. inside your repo, where the pom.xml file locates, build the war file:

```
mvn package
```
 2. show tomcat web apps, it should NOT have the war file yet:

```
ls -lah /var/lib/tomcat9/webapps/
```
 3. copy your newly built war file:

```
cp ./target/*.war /var/lib/tomcat9/webapps/
```
 4. show tomcat web apps, it should now have the new war file:

```
ls -lah /var/lib/tomcat9/webapps/
```
 - **Show your XML parsing on your terminal (ssh to AWS).**
 - Move your XML data files to a location your XML parser will need.
 - Build and execute your parser.
 - An inconsistency report should be output to the console or a file and will be checked later.
 - The entire parsing time cannot exceed **15** minutes on the AWS machine.
 - **Show your website in your web browser:**
 - use HTTPS to go to your website. Manually change it to HTTP with port 8080, and it should be redirected to HTTPS correctly.
 - Login with a customer credential.
 - First, without using reCAPTCHA, an error message should be displayed.

- Then use reCAPTCHA and login, and it should succeed.
- Show that there is no access to the employee dashboard with the customer's account.
- Login with the employee credentials email "classta@email.edu" and password "classta".
 - Go to the employee dashboard.
 - Skim through the metadata of your database.
 - Go to add a star:
 - enter star name "Star1" and birth year "2021", add it, a confirmation message should be displayed with the generated star ID.
 - Go to add a movie:
 - enter movie name "**Movie1**", director "**Director1**", year 2021, genre "**Genre1** (new)", star "**Star2** (new)", add it, a confirmation message should be displayed with the generated Movie ID, Genre ID, and the star ID.
 - enter movie name "**Movie2**", director "**Director1**", year 2021, genre "**Genre1** (existing)", star "**Star1** (existing)", add it, a confirmation message should display with the generated Movie ID, and the existing star ID, Genre ID, that was found.
 - enter movie name "**Movie1**", director "**Director1**", year 2021, genre "**Genre2** (new)", star "**Star2** (existing)", add it, an error message should be displayed due to duplicated movie.
- **After XML parsing is finished, examine the data you entered:**
 - Browse movies by genre "Genre1". Two movies ("Movie1" and "Movie2") should show up. Click each movie to go to Single Movie Page, and show star and genre information.
 - Search movie by title "Movie", the same two movies should show up.
 - Choose any **two** movies from the following and display the Single Movie Page of each (those are from the XML file):
 - Mission Impossible
 - Forrest Gump
 - The Godfather
 - Star Wars
 - The Silence of the Lambs
 - The Matrix
- **Show your inconsistency report from the XML parser.**
- **Done.**
- Here is a [sample \(https://youtu.be/ZEyUdp5jVrg\)](https://youtu.be/ZEyUdp5jVrg) demo video.

▼ Submission

Same as Project 2, but please update README with the Project 3 content.

In addition:

- Include a stored-procedure.sql for the stored procedure!
- README:
 - List filenames with Prepared Statements.
 - Two parsing time optimization strategies compared with the naive approach.
 - Inconsistent data reports from parsing. It also can be referred to from another separate report file generated by your code.