

GROUP-6 PS4: Documentation

Url: <https://web06.cs.ait.ac.th/>

Gitlab repo: <https://gitlab.com/ait-fsad-2022/web6/PS1>

Implementation of issue tracking

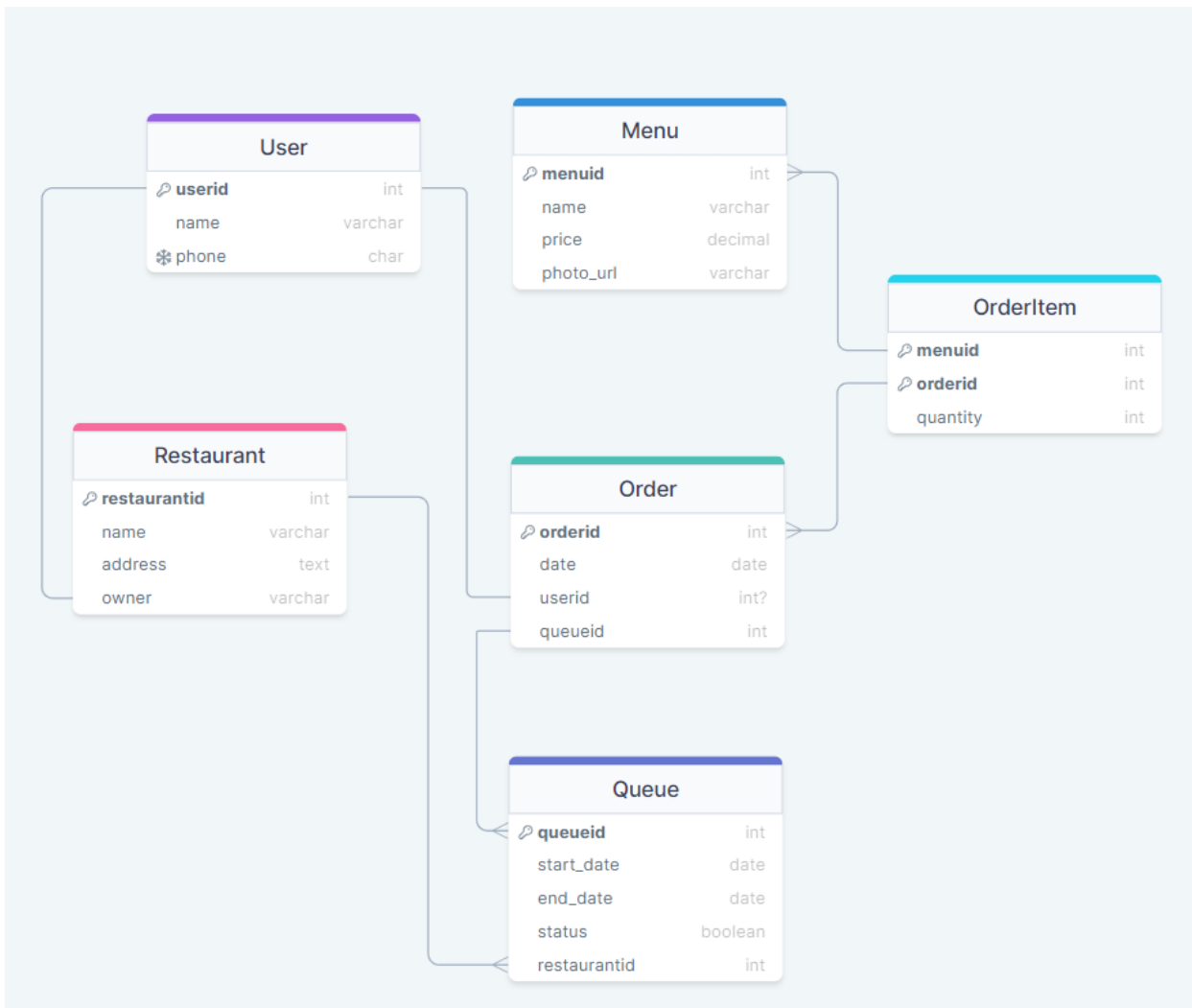


We implemented issue tracking to keep track of to-dos. Issues can be automatically closed by including the keywords in the commit messages such as 'Closes #1'.

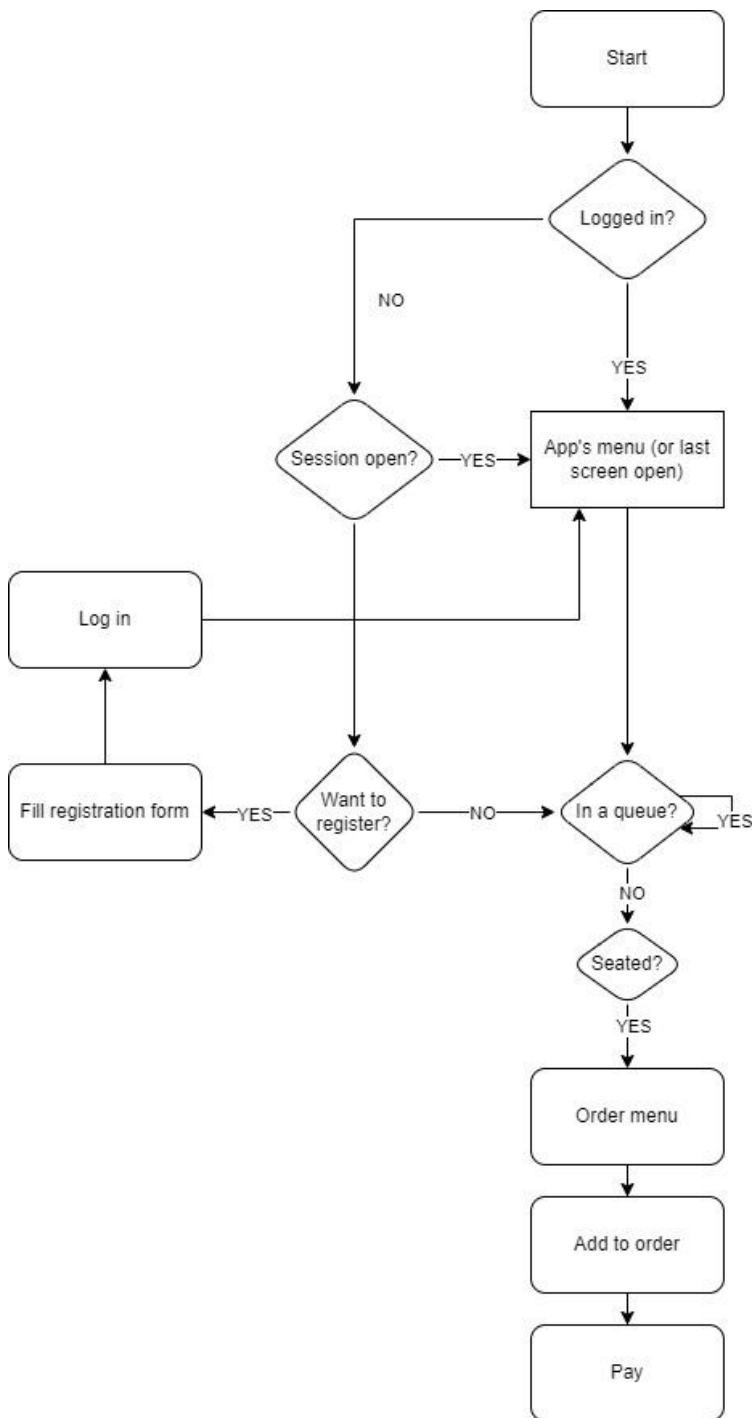
Implementation of administrator page

SQL Tables

<https://drawsql.app/teams/jekams/diagrams/yumq>



User registration and usage flow diagram (customer)



Preventing SQL injection and XSS attacks

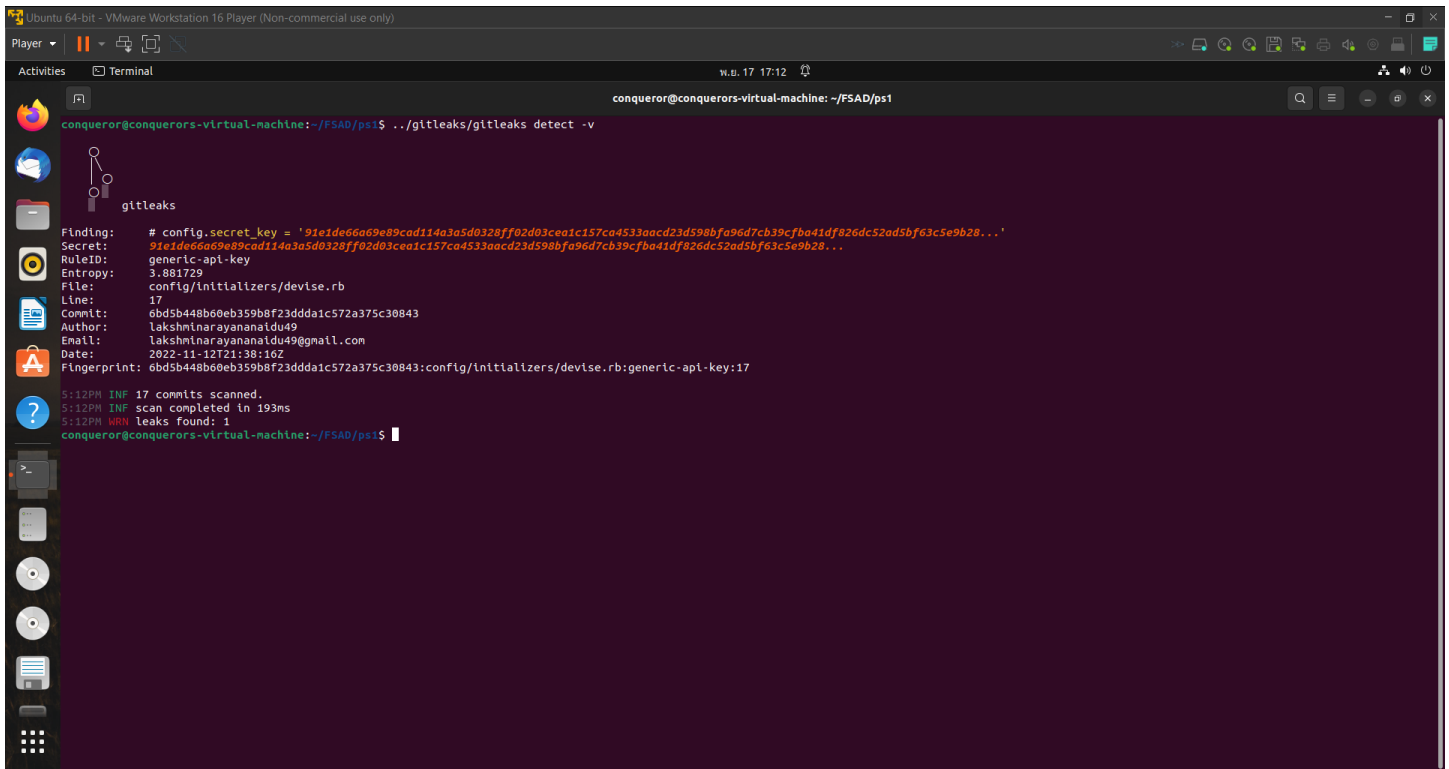
In order to prevent SQL injection attacks, we need to do the following steps while developing our project. They are as follows

1. Validating Input
2. Auditing database
3. Changing the default settings to custom settings
4. Securing the system

1. Validating Inputs:
 - a. Inputs from the SQL statements should be validated or sanitized before being sent to the database. This can be done by making use of the parameterized SQL statements, which separate the query statements from the data.
 - b. Parameterized statements use stored queries that have markers, known as parameters, to represent the input data. Instead of parsing the query and the data as a single string, the database reads only the stored query as query language, allowing user inputs to be sent as a list of parameters that the database can treat solely as data.
2. Auditing database:
 - a. We can make use of the extensions like `pg_audit` which provides a deeper level of auditing such as detailed session and object logging than the standard logging found in the PostgreSQL. This level of detail can help pinpoint any unusual or irregular queries.
 - b. System tables should be treated with suspicion since they won't be regularly accessed by the users.
 - c. Removing the tables, views, procedures and functions which are no longer in use reduces the chances of an attacker injecting them with the malicious code.
3. Changing the default settings to custom settings:
 - a. We have to change some of the default settings in order to prevent any kind of security vulnerabilities.
 - b. We can also maximize the information captured in our log files. This can be done by changing the settings in the **`postgresql.conf`** file.
4. Securing the system:
 - a. Turning off the database errors on production sites to avoid revealing information about the system structure.
 - b. Keeping the database and the OS up to date with the latest security patches.
 - c. Making use of vulnerability scanners and Web Application Firewalls (WAF) to make sure that applications are secure before releasing them to production servers.

Auditing software for sensitive data

In order to find out any sensitive information leak in the software, we used the **GITLEAKS** tool. Gitleaks is officially used by the github and gitlab to find out any sensitive information leak. After running the gitleaks tool on our repository in our local machine, we have found out that there is a secret exposed in the `devise.rb` file.



```
conqueror@conquerors-virtual-machine: ~/FSAD/ps1
conqueror@conquerors-virtual-machine: ~/FSAD/ps1$ ../gitleaks/gitleaks detect -v

gitleaks

Finding: # config.secret_key = '91e1de66a69e89cad114a3a5d0328ff02d03cea1c157ca4533aacd23d598bfa96d7cb39cfba41df826dc52ad5bf63c5e9b28...'
Secret: 91e1de66a69e89cad114a3a5d0328ff02d03cea1c157ca4533aacd23d598bfa96d7cb39cfba41df826dc52ad5bf63c5e9b28...
RuleID: generic-apl-key
Entropy: 3.881729
File: config/initializers/devise.rb
Line: 17
Commit: 6bd5b448b0eb359b8f23ddda1c572a375c30843
Author: lakshminarayananaidu49
Email: lakshminarayananaidu49@gmail.com
Date: 2022-11-12T21:38:16Z
Fingerprint: 6bd5b448b0eb359b8f23ddda1c572a375c30843:config/initializers/devise.rb:generic-apl-key:17

5:12PM INF 17 commits scanned.
5:12PM INF scan completed in 193ms
5:12PM WRN leaks found: 1
conqueror@conquerors-virtual-machine: ~/FSAD/ps1$
```

This process of auditing can be automated by including the following in the Gitlab CI/CD configuration file.

```
include:
  - template: Jobs/Secret-Detection.gitlab-ci.yml
```

Implementation of user login/registration pages

For the user registration/login pages, we made use of devise gem.

Implementation of administrator page

For the implementation of the administrative pages, we have used the **activeadmin** gem. We have setup the activeadmin using the documentation that is referenced in references section. The setup process is fairly straightforward.

In order to ban the users, we have added a new column banned to the table which is of type boolean. Devise checks if the model is active by calling the active_for_authentication method before authenticating an user. We have overwritten the active_for_authentication method in the users model with the following so the user will only be authenticated if he is not banned by the admin.

```
def active_for_authentication?
  super && special_condition_is_valid?
end
```

```
def active_for_authentication?  
  super && !self.banned  
end
```

References:

<https://www.crunchydata.com/blog/preventing-sql-injection-attacks-in-postgresql>

<https://stackoverflow.com/questions/5629480/rails-devise-is-there-a-way-to-ban-a-user-so-they-cant-login-or-re-set-their>

<https://activeadmin.info/documentation>

<https://rubyonrailssolutions.wordpress.com/2017/02/06/how-to-remove-comments-option-in-activeadmin-menu-in-rails-4-0-3-2/>

<https://github.com/zricethezav/gitleaks>