Group 5 Portfolio

ESOF 423

Jacob Tanner, Mike Kadoshnikov, Boone Schmaltz, Gunnar Rasmussen

Section 1: Program

Link to GitHub page: https://github.com/423s24/Group_5
Database Management System: PostgreSQL
Web Framework: Django

Section 2: Teamwork

Team member 1 (20%):

Focused on maintenance form and support page. Set up emails for testing and got a gmail account setup and authorized for use in sending email notifications. Created html notifications with plaintext backup. Fixed maintenance form bugs after outside testers provided feedback.

Team member 2(10%):

Focused on housekeeping and keeping the GitHub IO page and documentation up to date, contributing to website styling, managing issues on GitHub, managing the support page on the website, and making sure the website maintains an intuitive user experience.

Team member 3 (37%):

Got user authentication to work (login, sign up, logout, password reset). Got account types set up, separated dashboards for each account type, and allowed admin users to change user account types. Hosted the production version on a public domain with a personal server for testing and client demos. Made the site mobile friendly.

Team member 4 (33%):

Initially setup the postgresql database and django project. Worked on several basic tables for buildings and maintenance requests. Got active search working for maintenance requests. Made the basic dashboard format. Created Maintenance Notes and functions for adding, editing, and deleting them. Added file upload to the project as well. Worked on exporting pages as pdf. Changed initial navbar and design into a sidenav. I managed to get django tests to work with GitHub continuous integration yaml file.

Section 3: Design Pattern

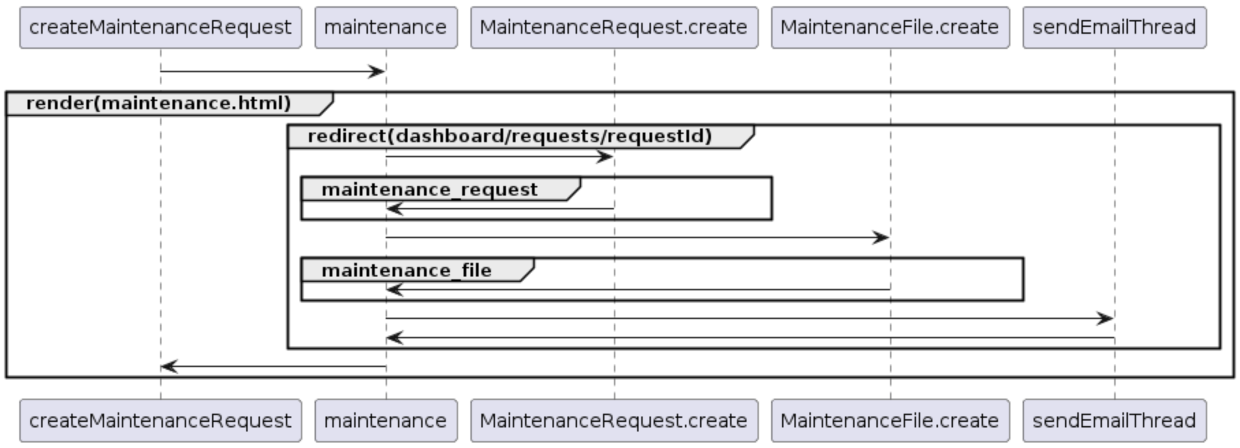
The project primarily inherits an active record pattern. An active record pattern is defined as such “The active record pattern is an approach to accessing data in a database. A database table or view is wrapped into a class. Thus, an object instance is tied to a single row in the table. After creation of an object, a new row is added to the table upon save. Any object loaded gets its information from the database. When an object is updated, the corresponding row in the table is also updated. The wrapper class implements accessor methods or properties for each column in the table or row.” How this relates to the project is as follows: The HRDC Maintenance Portal primarily uses a database to store all information regarding the buildings, maintenance forms, users, saved requests, and related objects. When a user, request, note, image, building, etc. are added to the project they are immediately added to the database and the tables reflect the change. We directly have a row of a table to an object, and the table is updated upon saving an object. Although this pattern is often considered an anti-pattern and can lead to greater issues it's also noted that “Another critique of the active record pattern is that, due to the strong coupling of database interaction and application logic, an active record object does not follow the single responsibility principle and separation of concerns. This is opposed to multi tier architecture, which properly addresses these practices. Because of this, the active record pattern is best and most often employed in simple applications that are all forms-over-data with CRUD functionality, or only as one part of an architecture. Typically that part is data access and why several ORMs implement the active record pattern.” As is typical with simpler applications and best to implement with them, we fall into form-over-data with CRUD functionality. Although considered an anti-pattern for this project it works perfectly. Although it will be hard to fully describe where exactly in the code it is, I believe 90% of the project contributes to this design pattern, and to highlight and contribute it to this document would be a rather tedious task. The project itself is meant to use CRUD directly with the database, there isn’t really any task that could be performed that doesn’t directly contribute towards the design pattern.

Section 4: Technical Writing

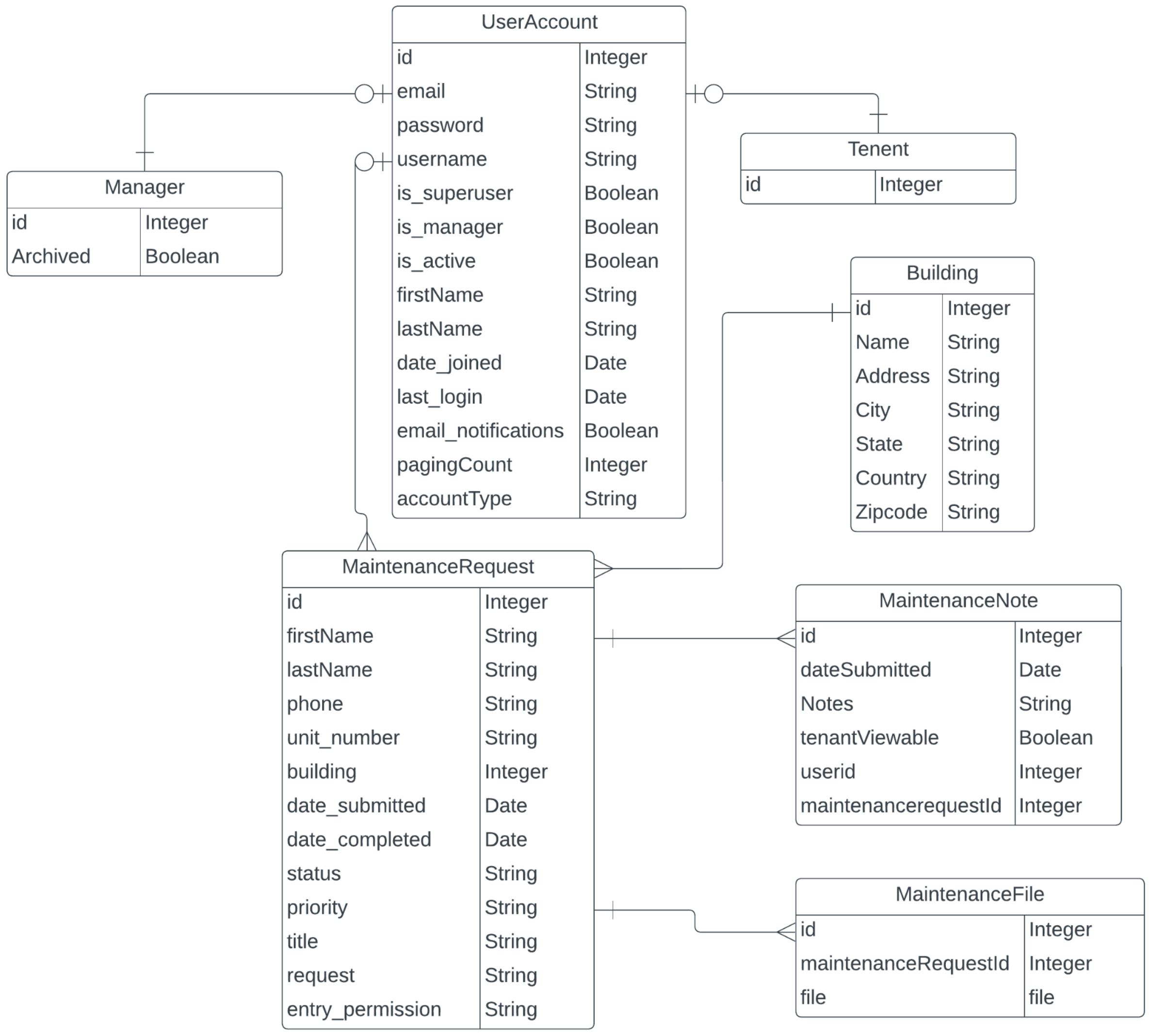
A link to the documentation for the maintenance portal can be found here: https://423s24.github.io/Group_5/

Section 5: UML

Here is a simple representation of how creating a maintenance request works:



Below we have our database diagrammed:



Section 6: Design Trade-Offs

Initially in the database design we were going to include a Unit field that would have tied a request to a building, and we would have the manager connected to the building, that way we could perform joins to get the manager from the unit. Eventually through testing of the software we came to an understanding that in order to minimize the amount of work the clients would need to do, we would just remove those connections and only tie the maintenance requests to their respective buildings. Not only does it reduce computation on the server-side but it also reduces the amount of items required to be added to the database. For example if the client has 15 buildings which all contain 20 units, then the client would have to add all 15 buildings and the 300 units associated to those buildings, then the user would have to choose their unit correctly on the maintenance form. Now we only require the client to add only the buildings totaling 15 additions, each request will be tied directly to the building and the unit/number will be input directly from the user submitting a request. With this trade off the client now has a reduced work load, the units field is removed freeing up more space on the database only at the expense that the user inputs the correct unit upon request completion. This design change reflects reducing the amount of computation with retrieving data, reducing the amount of space the database takes up, and also reducing the amount of work for the client, at the expense of the user verifying that their Unit number is correct when inputting it in the maintenance request form.

Section 7: Software Development Life Cycle Model

We used Agile project management to develop our project. This allowed our team to discuss who was going to work on what and for discussion of this every class period. Dividing the project into sprints gave our group a decent structure for what needed to be finished by when, so dividing up the work per day per teammate was simple. The problem with this type of development comes in whenever a team member is absent. Even if they check in over text, it doesn’t quite replace the value you get out of an actual scrum meeting and session of class, especially if it was a meeting at the start of a sprint or demo day.