Performance Analysis and Optimization Report

MainePad Finder

Introduction

MainePad Finder is a full stack web application designed to help Maine college students and other renters discover housing opportunities in Maine. The system allows users to create accounts, search and filter rental properties, view designed listings and landlord information, exchange messages with other users, and manage their own properties as landlords. This application is built on a React frontend, a Flask backend, a MySQL relational database, and is deployed in a local development environment using http://localhost:5173 for the frontend and http://localhost:5000 for the backend.

As the number of properties, users, messages, and reviews grow, the performance becomes a critical part of making the design of our system. Having slow page loads, inefficient database access, or unnecessary network traffic would severely impact the usability of our application. For this reason, we tried to prioritize performance and optimization while implementing our many features we have in *MainePad Finder*.

The goal of this report is to document how we analyzed and improved the performance of *MainePad Finder* across the entire application.

On User Accounts

The signups feature allows users to sign up for an account in the Maine Pad Finder application. For the signup page we worked on reducing unnecessary work on both the client and server side. On the frontend we added simple client side validation (e.g., checking if password and confirm password match) so invalid submissions are rejected before a network request is even made. This avoids wasted backend and database processing. On the backend, the /api/signup endpoint now performs a single existence check using SELECT USER_ID FROM USERS WHERE USERNAME = ? OR EMAIL = ?, instead of separate queries for username and email and relies on indexes and unique constraints on those to keep this lookup efficient as the USERS table grows. Additionally the signup page now adds to renter or landlord tables within the same query that creates the entry into users. Previously it was one query to add to users, and then another query to add to either renter or landlord. This improves performance by reducing the network latency of sending several queries and leveraging the database's optimized conditional logic.

On Property-Related Queries

The Properties feature could easily become a bottleneck by loading all properties and filtering them in Python or the browser, issuing separate queries for PROPERTY and ADDRESS, and using SELECT * to return unnecessary columns, all of which would increase database load, network traffic, and rendering cost. We avoided these issues by pushing the work into MySQL with a single JOIN-based query (PROPERTY to ADDRESS) that applies filters directly in the WHERE clause and selects only the fields needed by the UI. On the frontend, Properties.jsx further improves performance by relying on server-side filtering and then rendering only 10 cards per page using client-side pagination, which keeps DOM size small and the interface responsive even when many rows match the search.

The listing feature is a component of properties. When you click on a specific property you are interested in listing will give you specifications of that property. For the Listing page, the main risks were making multiple round-trips to the database (separate queries for property, address, landlord, and rating) and using a slow "check then insert or update" pattern for reviews. We addressed these potential bottlenecks by calling a single stored procedure (GET_LISTING_WITH_LANDLORD) in GET /api/listing/<id>, which performs the necessary joins and aggregations inside MySQL and returns all relevant data in one row. For reviews, POST /api/listing/<id>/review uses INSERT ... ON DUPLICATE KEY UPDATE to handle both new and edited reviews in a single query, with validation of the star rating in Python before hitting the database. On the frontend, the Listing page only loads one listing at a time and updates just the review state rather than re-fetching large datasets, which keeps the page fast and predictable.

On Messaging

Messaging could suffer from performance problems such as N+1 queries to look up sender usernames, duplicated authentication checks in each endpoint, and excessive polling or refetching on the frontend. In GET /api/messages/thread, we avoid N+1 by using a single query that pulls all messages between the two users, joins MESSAGE with USERS once to get SENDER_USERNAME, and returns the full ordered thread in one round-trip. Both messaging endpoints are protected by the shared @login_required decorator, which validates the session token once and exposes g.user_id, eliminating repeated auth logic. On the frontend, Messages.jsx calls /api/me only once on mount, loads a thread only when the user explicitly requests it, and sends messages via a single POST /api/messages/send call, updating the UI without unnecessary extra requests—together, these choices keep the messaging feature efficient for both server and client.

On Data Analytics

As an advanced feature of the *MainePad Finder* database and backend, some simple schemas and triggers were designed to enable the analysis of a property's price over time. These implements were designed simply and tersely, both to ensure readability and to render them more functionally compact, with less reliance on complex and time-costly queries. The entire new schema for providing an updating list of property price changes is provided below

```
CREATE TABLE IF NOT EXISTS PROP_PRICE_HISTORY (
    ENTRY_ID INT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
    PROP_ID INT UNSIGNED NOT NULL,
    RENT_COST INT UNSIGNED NOT NULL,
    PRICE_START DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (PROP_ID)
    REFERENCES PROPERTY (PROP_ID)
);
```

PROP_PRICE_HISTORY.sql

and the trigger which populates this table is provided below

```
CREATE TRIGGER ADD_PROP_PRICE_HISTORY
BEFORE UPDATE
ON PROPERTY
FOR EACH ROW
BEGIN
    IF (NEW.RENT_COST <> OLD.RENT_COST) THEN
         INSERT INTO PROP_PRICE_HISTORY (PROP_ID, RENT_COST)
         VALUES (NEW.PROP_ID, NEW.RENT_COST);
    END IF;
END //

DELIMITER ;
```

ADD_PROP_PRICE_HISTORY.sql

The design of both scripts was intentionally kept simple and terse to enable the *MainePad Finder* database to comprehensively track how *all* properties' prices have changed over time, while enabling minimal complex queries or joins to hasten the collection of that price change data. This design also permits individual property price trends to be intuitively examined in closer detail through simple queries (as the `prop_price_trending` function in the `app.py` Flask backend illustrates). Although the `FOR EACH ROW` qualifier for this trigger could be

perceived as a deficiency within this system, given that many property price changes would realistically be performed on single, individual properties, it would still prove useful in circumstances where a landowner owns multiple individual units within a larger apartment complex that are all similar in size and structure, justifying its preservation here. Additionally, the `IF (NEW.RENT_COST <> OLD.RENT_COST)` conditional ensures that updates upon a property which do not change the cost of rent for that property do not perform any queries, removing additional deficiencies from the trigger.