## Introduction

This report details the design and architecture for the Sublime event ticketing platform - a web app that will connect customers with Kelowna music events and event organizers with local attendees.

The technology stack chosen for the project includes Python, Django web framework, Stripe Payments, SQL Express database (for local development), and AWS with DynamoDB (for production).

Subsequently, this report presents appropriate models that identify different entities in the system (class diagram) and how they interact with respect to time (sequence diagrams). Additionally, this report identifies the testing approaches and design patterns to be used for app development.

## Models: Sequence Diagrams

For this report, two primary use cases were modeled using sequence diagrams:

|  |  |
| --- | --- |
| 1 | Use Case: Buy a Ticket  Primary Actor: Customers |
| 2 | Use Case: Create an Event  Primary Actor: Event organizer |

##### Customer Buys a Ticket

The following sequence diagram shows the time-ordered interactions between the customer, system and database when buying a ticket.

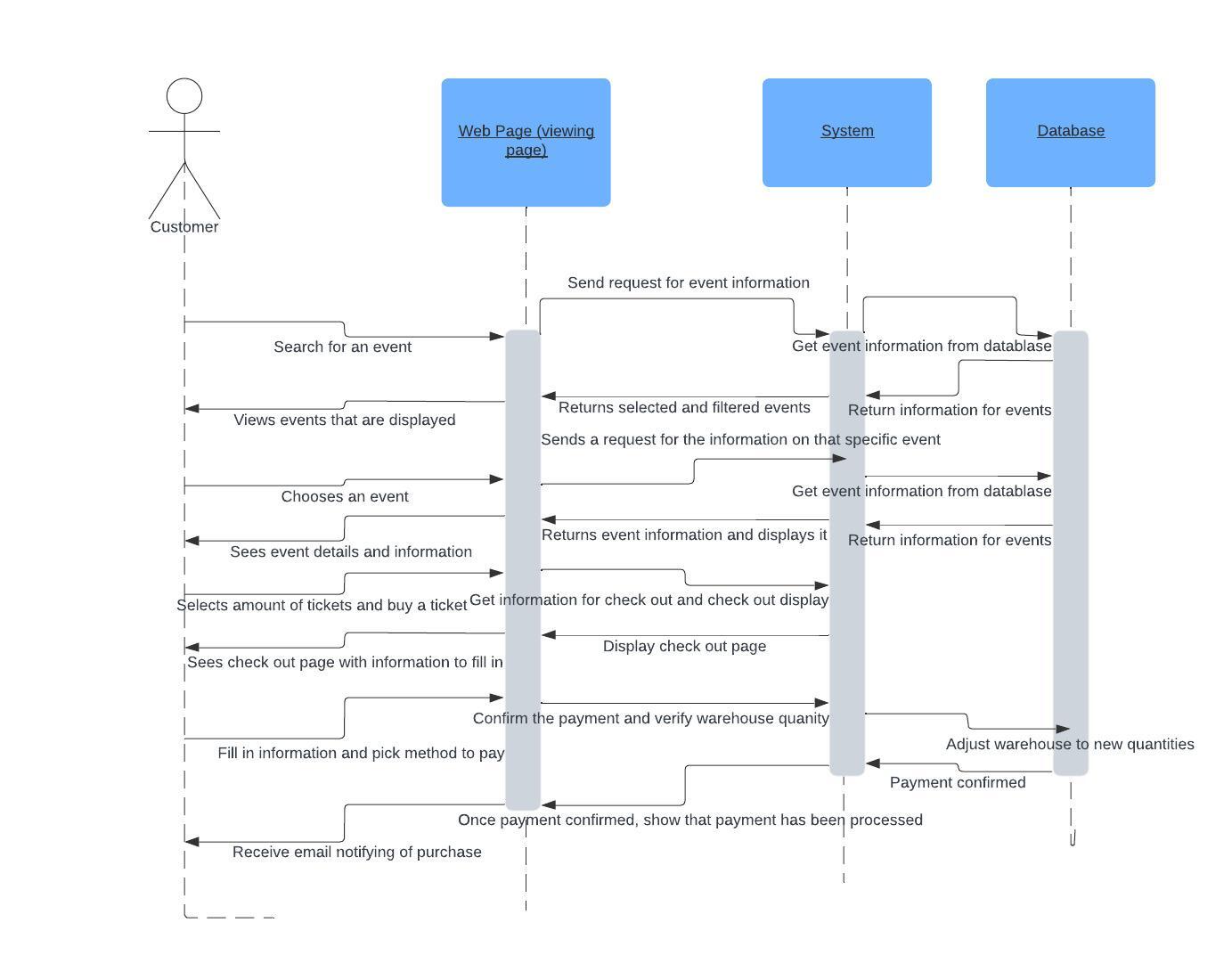


Figure 1: Sequence Diagram - Customer Buys an Event Ticket. [Lucid Chart Link](https://lucid.app/lucidchart/661b369f-c980-46fa-bf8e-40f5a0c4fb5f/edit?invitationId=inv_0f90c8ae-7bde-4339-a6c9-00acf5c8c34a&page=0_0)

##### Event Organizer Creates New Event

The below sequence diagram shows the time ordered interaction between event organizer, system view, controllers, and models in creating an event.

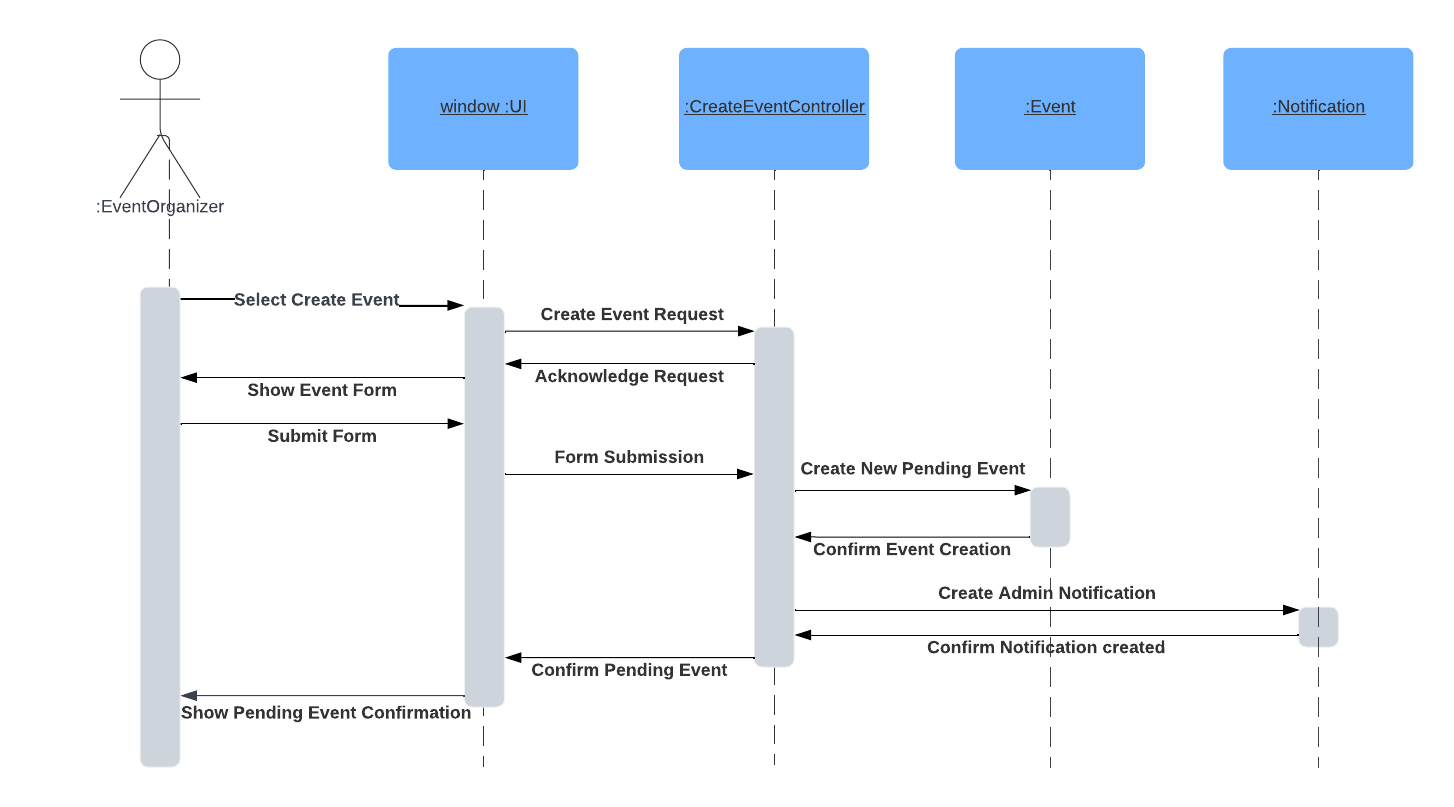


Figure 2: Sequence Diagram - Event Organizer Creates New Event. [Lucid Chart Link](https://lucid.app/lucidchart/6f82500b-77e4-42b4-8974-1a9c779c94c4/edit?invitationId=inv_09ab77ac-3327-429c-a9e6-b2e96cd665a2&page=0_0)

## Models: Class Diagram

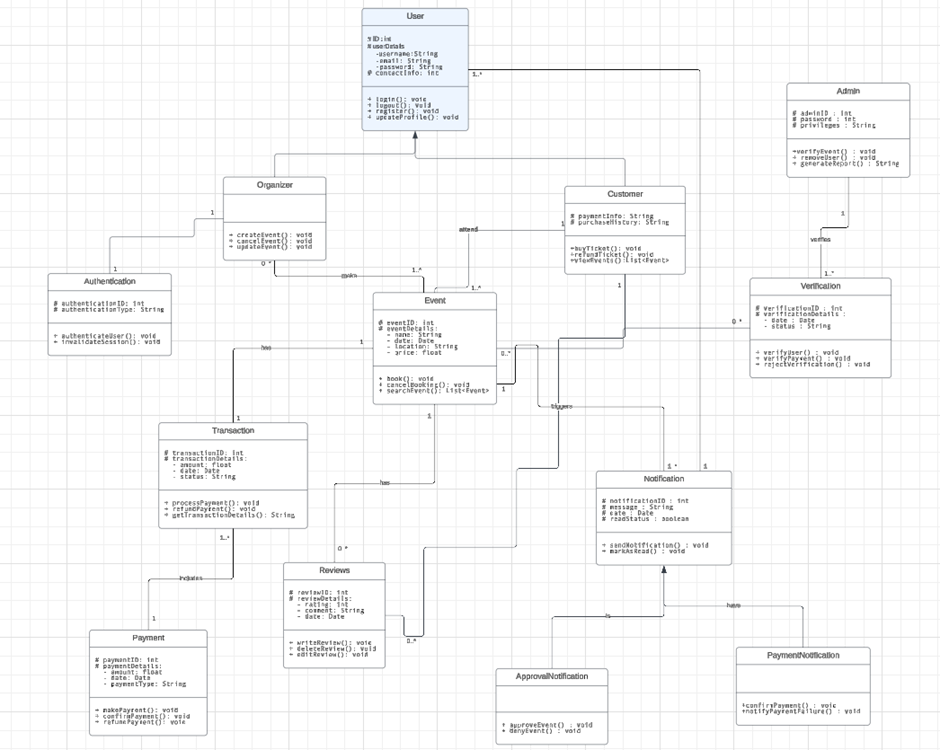


Figure 3: Class Diagram. [Lucid Chart Link.](https://lucid.app/lucidchart/2fdbb0ef-3602-481d-8183-c5e6c2d6b2ac/edit?invitationId=inv_033dd7b3-1967-48b3-946c-751f265b24ae&page=0_0)

## Testing Plan

Below, the testing approaches (both current and future) are described for the Sublime event ticketing platform. The goal of this testing plan is to ensure the project meets its specifications and user expectations.

##### Current Approaches

###### Test Environment

* Features of the system are developed locally using standardized tools such as Visual Studio Code and python virtual environments.
* Software/ packages currently used for testing code includes:
  + - [Coverage.py](http://coverage.py)
    - [Django unit tests](https://docs.djangoproject.com/en/5.0/topics/testing/overview/)
    - [Pytest](https://docs.pytest.org/en/8.0.x/)

###### Functional Requirements

The following functional requirements are currently in development for the project and, consequently, we have implemented tests for each of these components:

* User Registration and Login:
  + Test Case: Registration with valid and invalid data.
  + Expected Outcome: Successful registration or appropriate error message.
* Event Listing:
  + Test Case: Browse, filter, and search for events.
  + Expected Outcome: Events displayed according to search and filter criteria.
* Payment Processing:
  + Test Case: Booking tickets using valid and invalid payment details.
  + Expected Outcome: Successful booking and payment or appropriate error message.

###### Test Reporting

During scrums, each team member provides a summary of the testing effort for their assigned issue: including total tests conducted, pass/fail outcomes, and unresolved issues. Additionally, tests are included in code submitted to the Github repository.

##### Future Testing

As we continue to develop the key functions of the web app, additional functional testing is planned for:

* Ticket Booking
  + Test Case: Booking tickets using valid and invalid payment details.
  + Expected Outcome: Successful booking and payment or appropriate error message.
* Event Verification
  + Test Case: Admin approving a pending event.
  + Expected Outcome: Pending event becomes an approved event that is now visible on the event listing page.
* User Profile Management
  + Test Case: Update user profile and preferences.
  + Expected Outcome: Profile updated successfully.

Additionally, this project looks to (in the future) conduct *Usability Testing to* evaluate the platform's ease of use, navigation, and overall user experience. Security testing will also be completed to ensure the web app is not affected by common vulnerabilities such as *SQL Injection and Cross-Site Scripting (XSS).* The project will also conduct *Performance Testing* (such as load testing and stress testing)and *Compatibility Testing to* ensure the platform works across different devices, browsers, and operating systems.

## Design Patterns

This project will utilize the following design patterns:

##### MVT Pattern

The Sublime app will be developed using Django - web framework which utilizes the Model-View-Template (MVT) pattern to create web apps. In Django, the *Model* represents the data and business logic while the *View* is responsible for presenting data to the user. The *Template* in Django handles user input and acts as a bridge between the Model and View.

Identified entities for the sublime app such as *Event*, *Customer*, *Review* will be coded into the Django *Model*. HTML elements such as forms and tables will be built into *Templates* and served to users via *Views*.

##### Factory Method

This pattern is useful for creating objects without specifying the exact class of object that will be created. It's particularly useful in our system for creating different types of user accounts (Administrator, Event Organizer, Customer) without binding our code to specific classes. This pattern promotes loose coupling and scalability, which is essential for managing various user roles with distinct permissions and functionalities.

Chosen because it allows the system to introduce new user types in the future (like vendors, venue owners, etc.) without altering the existing code that creates users. It encapsulates the user creation process, making the system more maintainable and scalable.

##### Observer Pattern

This pattern is valuable for a system where changes to one object (subject) need to be reflected in multiple other objects (observers) without making the observers dependent on the subject. It's particularly applicable for notifying users (Event Organizers and Customers) about changes in event details, approval statuses, and ticket purchases. For example, when an event's details are updated and approved, all subscribed customers (observers) can receive notifications about the update.

Selected because it efficiently manages notifications related to event updates, ticket purchases, and other real-time changes that need to be communicated to multiple users. This pattern ensures that the system remains flexible and loosely coupled, as it can easily add or remove observers (users) as needed without modifying the subject (event details, ticket inventory).