

Impulse

Team 10 - CS 4485.0W1 Group Project

Patrick Guinn,

Esha Kumar,

Sai Neethu Bonagiri,

Sophia O'Malley

Supervisor: Professor Sridhar Alagar

Course Coordinator: Thennannamalai Malligarjunan

Erik Jonsson School of Engineering and Computer Science

University of Texas at Dallas

May 11th, 2025

Table of Contents

1. Introduction

- 1.1 Background
- 1.2 Objectives, Scope, and Goals Achieved
- 1.3 Key Highlights
- 1.4 Significance of the Project

2. High-Level Design

- 2.1 System Overview & Proposed Solution
- 2.2 Design and Architecture Diagrams
- 2.3 Overview of Tools Used

3. Implementation Details

- 3.1 Technologies Used
- 3.2 Code Documentation
- 3.2 Development Process

4. Performance & Validation

- 4.1 Testing and Validation
- 4.2 Metrics Collected and Analysis

5. Lessons Learned

- 5.1 Insights Gained
- 5.2 Lessons from Challenges
- 5.3 Skills Developed and Improved

6. Future Work

- 6.1 Proposed Enhancements
- 6.2 Recommendations for Development

7. Conclusion

- 7.1 Summary of Key Accomplishments
- 7.2 Acknowledgements

8. References

9. Appendices

Chapter 1: Introduction

1.1 Background

Impulse is a mobile application that seeks to improve the college experience by providing students a direct way to meet new people and engage in events on campus. There are many potential barriers that can prevent students from engaging in campus events. The location and time of the event are oftentimes the preceding factor that determines whether a student can attend or not. This is especially true if a campus has a high percentage of commuter students, such as the case for the University of Texas at Dallas.

Our solution to improve campus life and engagement takes into account the busy and sometimes erratic schedule of the college student. Instead of conforming strictly to the schedule of university-ran events, through our application the student has the power to create and join events with their fellow students. These events can range from anything to chess, studying, sports, or just simple hangouts. We believe that empowering the students in such a way will vastly improve their college experience as well as unlock new potential for the campus life of the university.

1.2 Objectives, Scope, and Goals Achieved

- **Objectives & Scope:**

Event Finder:

- The primary goal of this application is to provide a way for students to create and join events. Created events must be customizable to the students preference since they have the power to create the events. This includes details such as event type, location, and maximum participants.

Leveling System w/ Rewards:

- Upon participating in events the students are rewarded with unlockables through our implemented leveling system. The leveling system awards experience points upon either creating or joining an event. By providing rewards we allow the student to feel more rewarded for their time spent on our application.

Profile Page:

- Student accounts will be provided with a profile page that is customizable. The profile page will allow the student to express themselves and showcase the rewards gained through the leveling system

Interactive Campus Map:

- In order to account for new students, an interactive campus map provides a convenient way for the student to navigate to events.

Student Verification:

- For safety and security of the users of the applications, accounts will go through a verification process to ensure that the user is indeed a current student of the university.

- **Goals Achieved:** Our application provides student-verification through student email verification. Event creation and subsequent participation is also provided through a user-friendly interface.

Events are customizable and can vary in size, type, and location. An interactive map is also provided to allow users to easily determine event locations. Impulse also includes a rewards system that allows the user to gain experience points after successfully creating or joining an event. The user also is provided with a customizable profile page where the rewards earned by the user can be showcased. Altogether we believe that Impulse provides a very convenient and rewarding experience to our users that helps encourage student engagement on campus.

1.3 Key Highlights

- **UTD-Only Secure Authentication:** Users register and log in exclusively with their @utdallas.edu email. Supabase handles verification and the Flask backend enforces JWT-encrypted tokens on every protected route.
- **Comprehensive Event Browsing & Reservation:** A responsive catalog of upcoming UTD events with full detailed views.
- **Student-Led Meetups:** Create or join informal study groups and campus meetups. Hosts can set date, time, location and capacity; attendees see a live head-count and can opt in or out at any time.
- **QR Code Check-In:** Each created meetup generates a unique QR code. Scanning this code adds the student to the attendee list and automatically awards participation points.
- **Points & Gamification System:** To boost engagement, attendees earn experience points for every event they attend. Points unlock badges, motivating students to discover new activities.
- **Customizable Profile & Social Features:** Users select an avatar, set a display name, add friends, and manage notification and privacy settings from a unified profile interface.
- **Personal Stats Dashboard:** Displays key engagement metrics- “Events Attended”, “Meetups Hosted”, and “Points Earned” –to provide users with an at-a-glance summary of their activity.
- **Maps:** Integrated interactive campus maps using Leaflet. Each event and meetup is pinned with a marker; users can zoom and pan, filter by category, and click a marker to view details.

1.4 Significance of the Project

This project resulted from a common issue that students from UTD face: the lack of community, a flexible student-centered platform for organizing and finding campus activities. Official university event apps do exist, but many students—especially commuters—struggle to participate due to time constraints, lack of visibility, or irrelevance to their interests. There are also UTD-related Discord servers that offer some level of community interaction; they are often unorganized, difficult to navigate, and not focused specifically on in-person campus events. The Impulse app solves this problem for both ideas because it is two platforms in one: one screen and timeline is for official, UTD-led events so that students can easily see and join activities that are planned and sponsored by the university. A second screen where students create and share their meetups—anytime, any topic, and anywhere. This is a great advantage for commuter students because they can plan or join activities in between classes and eliminates the need to coordinate long-term commitments or serious planning efforts.

By combining both the UTD-led events and student organized meetups, Impulse creates the highest opportunity for a stronger sense of community, visibility of on-campus opportunities, and spontaneous

social interaction. Impulse does the hard-work of generating a sense of community on a primarily commuter campus where students can make meaningful connections on their own terms.

Chapter 2: High-Level Design

2.1 System Overview & Proposed Solution

API Implementation:

- Our system utilizes a RESTful API setup for the many benefits that this structure provides such as, reliability, scalability, and performance.
- RESTful APIs are implemented throughout the application and handle our core features such as user login, event creation, chat, and personalized event suggestions
- For routes that require security and authentication, we utilize token-based authentication through Supabase Auth.
- Role-based access is also utilized for the certain routes where access needs to be restricted to special users, such as admin.
- Input validation is also featured in our APIs in order to prevent spam and abuse.
- In order to expose our routes, the Flask framework is utilized to help create our route blueprints that are registered within the main application.

RESTful API Structure:

- Resource-based URLs:
 - Following RESTful principles, our routes contain Resource-based URLs with a hierarchical structure, for example, `"/events/<int:event_id>/join", methods=['POST']` is how the route for joining events is defined. As shown in this example, HTTP methods are also used to appropriately access data.
- Stateless Communication:
 - Our endpoints also use stateless communication by making each request independent and not reliant on any previous request.
- JSON-formatted data:
 - Also all endpoints return JSON-formatted data through use of the helper function `jsonify()` provided by the Flask framework.

2.2 Design and Architecture Diagrams

[Include system architecture, flow diagrams, and database schemas, ER diagrams.]

Figure 1.1 System Architecture Diagram

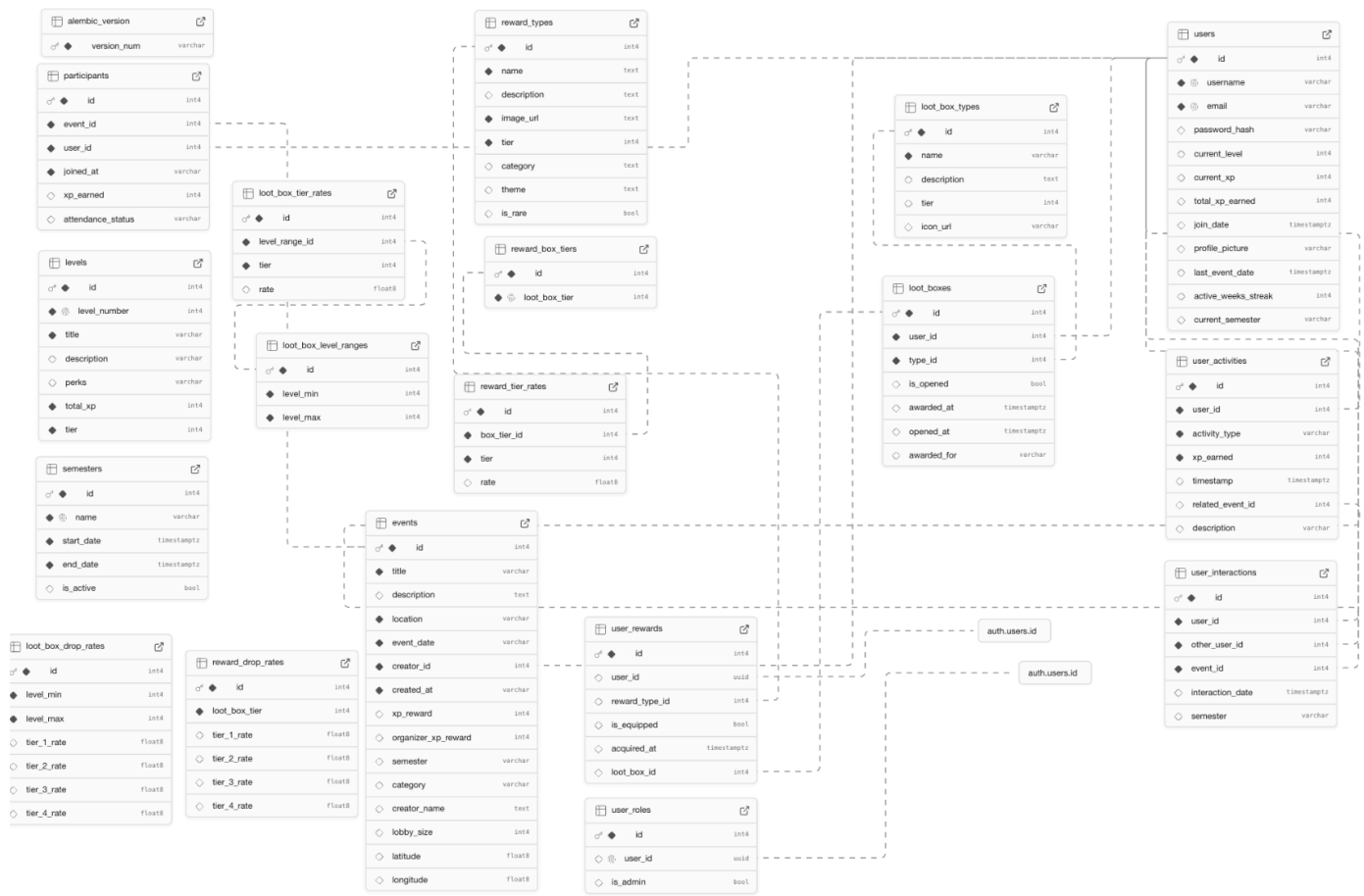


Figure 1.4 Screens from Impulse

impulse

UTD Events

Browse student-led events across campus

Q Search events...

All

Academic


Social

Sports

Tech

A

Music



UTD Jazz Ensemble Spring Concert

The UTD Jazz Ensemble presents their Spring concert featuring classic and contemporary ja...

Date: May 1, 2024

Time: 7:30 PM - 9:30 PM

Location: University Theatre, UTD

Home

Events

Meetups

Profile

Maps

impulse

Meetups

Find student-organized meetups

0 pts

Level 1

Q Search meetups...

All

Academic

Social

Sports

Tech

A

List View

Map View

+ Create New Meetup

View All Meetups on Map

Points by Group Size

Small Group: 10 pts (1-5 people)

Medium Group: 20 pts (6-15 people)

Large Group: 30 pts (16-30 people)

Mega Event: 50 pts (31-100 people)

Calculus II Late-Night...

academic

Home

Events

Meetups

Profile

Maps

Create New Meetup

Title

Chess at the Student Union

Description

Looking for chess partners at the Student Union. All skill levels welcome!

Date

Pick a date

Time

Select a time

Location

Search campus locations...

Category

Select a category

Lobby Size (max participants)

5

Create Meetup

Meetup Details

Calculus II Late-Night Study Jam

+20 pts

academic

Struggling with integrals and series? Join fellow undergrads for a collaborative problem-solving session—bring your notes, snacks welcome!

05/10/2025 8:00 PM

McDermott Library

S Organized by Sai Neethu Bonagiri

Attendees

4/12

S Sai Neethu Bonagiri

Attending

S Sophia O'Malley

Interested

E Esha Kumar

Attending

P Patrick Guinn

Attending

Join Lobby

impulse

Profile

Manage your account

John Doe

johnndoe@example.com

Technology Gaming Edit

Level 1

0 points

Progress to Level 2

0/10 points

Attended 0 meetups

Activity Insights

Meetups

Avg. Points

Friends

Add

No friends added yet. Add your first friend!

Account Settings

Edit Profile

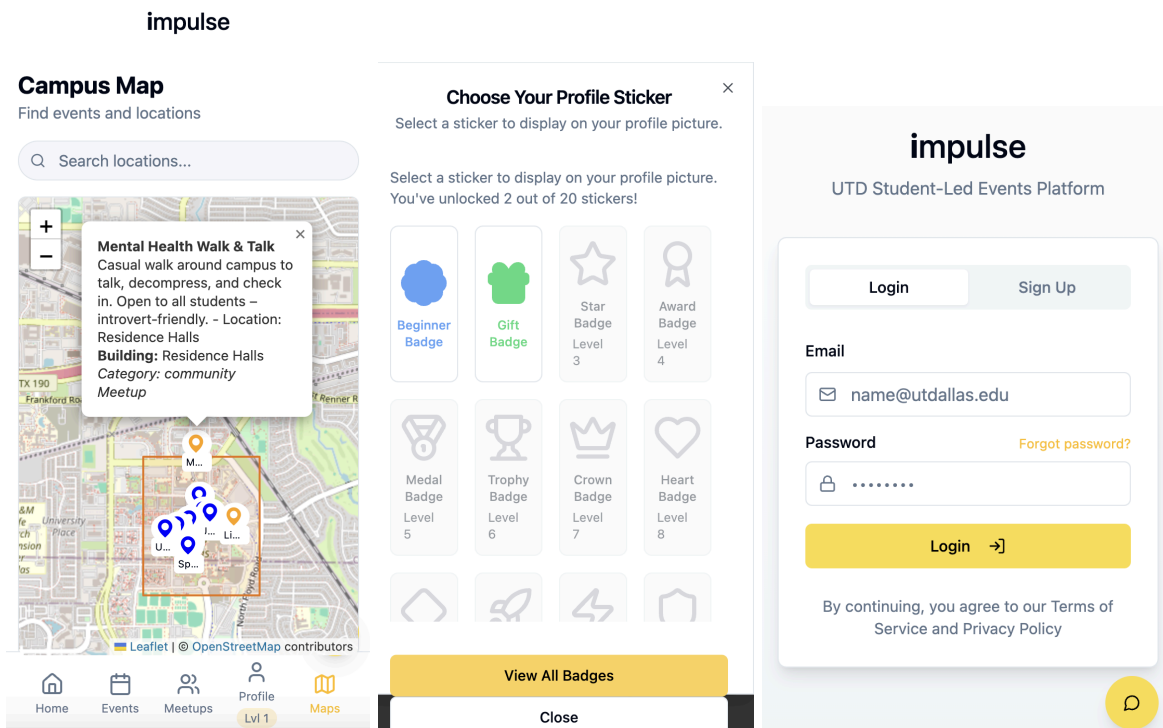
Choose Avatar

Upload Profile Picture

Notification Settings

Privacy Settings

Logout



2.3 Overview of Tools Used

- **Supabase:** Used as the backend database and authentication provider during early stages. It offered a PostgreSQL database with a built-in REST API, making it easy to store and query event and user data.
- **Loveable:** A no-code/low-code platform used to rapidly prototype the app interface. It helped us visualize and test design flows before finalizing development in code.
- **ngrok:** Used for tunneling localhost to a public URL during development. This made it easy to test mobile builds and share the running backend with teammates or Postman for testing.
- **Capacitor:** Integrated to convert the web-based React app into a mobile build for Android devices. It allowed us to test and deploy the app on actual phones without rewriting the codebase.
- **Postman:** Used for testing API endpoints during development. It helped us verify request/response behavior, handle authentication headers, and debug backend issues efficiently.
- **Figma:** Helped us plan the screens structure/design and the UI for how the users would interact with the application.

Chapter 3: Implementation Details

3.1 Technologies Used

Frontend

- **React:** Used to build the user interface with reusable components and dynamic rendering of event pages and user profiles.
- **Vite:** A fast build tool and dev server used to run the frontend with hot module replacement during development.
- **TypeScript:** Added type safety and helped catch errors during development, especially when working with props and API responses.
- **Tailwind CSS:** Used for styling the UI with utility classes, making the layout clean, responsive, and easy to maintain.

Backend

- **Python + Flask:** Used to build lightweight RESTful API endpoints for event management, user data handling, and recommendations. Flask handled routing, request parsing, and integration with the database.
- **PostgreSQL:** A relational database used to store event details, user accounts, chat messages, and other structured data.
- **Supabase:** Used during development for database hosting, authentication, and its built-in REST API before switching to Flask.

Development Tools

- Virtual Environment, VsCode, Github, Anaconda, Postman, Figma

3.2 Code Documentation

Backend: Show example of: routes, database table, app creation, app.py

Database Table Structure:

```

4  class Event(db.Model):
5      __tablename__ = 'events'
6
7      id = db.Column(db.Integer, primary_key=True)
8      title = db.Column(db.String(100), nullable=False)
9      description = db.Column(db.Text)
10     location = db.Column(db.String(100), nullable=False)
11     event_date = db.Column(db.String(50), nullable=False)
12     creator_id = db.Column(db.Integer, db.ForeignKey('users.id'), nullable=False)
13     created_at = db.Column(db.String(50), nullable=False)
14
15     # Add semester tracking for events
16     semester = db.Column(db.String(20), nullable=True)
17
18     # Fixed XP values as per the design
19     xp_reward = db.Column(db.Integer, default=50) # Base XP for attending (50)
20     organizer_xp_reward = db.Column(db.Integer, default=200) # XP for organizing (200)
21
22     # Update relationship
23     participants = db.relationship('Participant', back_populates='event', lazy=True)
24     creator = db.relationship('User', backref=db.backref('created_events', lazy='dynamic'))

```

- Through the use of SQLAlchemy, a Python Object-Relational Mapper the database tables are defined by the code structure shown above.
 - “db.Model” - is a base class that maps Python classes to database tables
 - “__tablename__ = 'events'” - determines the actual name of the table in the database
 - “db.Column()” - is used to define each table column in the table
 - “primary_key=True” and “nullable=False” - are examples of additional parameter specifiers that place constraints on the types of data that is allowed

Application Creation:

```

2  import os
3  from flask import Flask
4  from flask_cors import CORS
5  from flask_migrate import Migrate
6  from models import db, initialize_default_data
7  from routes import events_bp, leveling_bp, rewards_bp
8  from routes.auth import auth_bp
9  from config import config_by_name
10
11
12  def create_app(config_name='dev'):
13
14      # Initialize Flask application
15      app = Flask(__name__)
16
17      # Configure CORS to allow requests only from frontend development server
18      CORS(app, origins=["http://localhost:8080"])
19
20      # Load configuration settings based on environment
21      app.config.from_object(config_by_name[config_name])
22
23      # Initialize database and migration support
24      db.init_app(app)
25      migrate = Migrate(app, db)
26
27      # Register blueprints for different API areas
28      app.register_blueprint(events_bp, url_prefix='/api')
29      app.register_blueprint(auth_bp, url_prefix='/auth')
30      app.register_blueprint(leveling_bp, url_prefix='/api/leveling')
31      app.register_blueprint(rewards_bp, url_prefix='/api/rewards')
32
33      # Initialize default application data
34      with app.app_context():
35          # When using Flask-Migrate, db.create_all() is not needed
36          # as migrations handle database schema changes
37          # db.create_all()
38
39          initialize_default_data()
40
41      return app
42
43
44  if __name__ == '__main__':
45      # Get environment from system or default to 'dev'
46      flask_env = os.getenv('FLASK_ENV', 'dev')
47
48      # Create and configure application
49      app = create_app(flask_env)
50
51      # Print all registered routes for debugging
52      print("Registered routes:")
53      for rule in app.url_map.iter_rules():
54          print(f"{rule} - Methods: {rule.methods}")
55
56      # Run development server with debug mode enabled
57      app.run(debug=True)

```

- Through the use of Flask, app.py sets up an application with multiple endpoints, authentication, database integration, and CORS (cross-origin resource sharing) support. The functionality of the specific methods used within this file are explained briefly in the comments shown within the example above.
 - Key Components:
 - “def create_app(config_name='dev')” - This method is the factory function that creates and configures the Flask application. The application can load different configurations based upon the desired environment (in this case ‘dev’ is chosen).
 - “CORS(app, origins=[“http://localhost:8080”])” - This line configures from where cross-origin requests are allowed.
 - “db.init_app(app)” - Provides the setup for the database by connecting the SQLAlchemy database instance to the Flask application
 - “migrate = Migrate(app, db)” - Initializes Flask-Migrate with the application and database instance which allows for database migrations that handle schema changes.
 - “app.register_blueprint(events_bp, url_prefix='/api')” - Here is how our feature-specific blueprints are registered. These blueprints define the APIs of the application.

Below is an example of how our authentication routes are structured in **auth.py**. We use a dedicated Flask blueprint, obtain a Supabase client via our configuration, and implement the **POST** /auth/register, **POST** /auth/login, and **GET** /auth/verify endpoints—each of which parses input, applies validation rules, calls Supabase, and returns structured JSON responses.

```

from flask import Blueprint, request, jsonify
from config import Config # Import Config class

auth_bp = Blueprint("auth", __name__)

# Get the Supabase client instance using Config class
supabase = Config.get_supabase_client()

@auth_bp.route("/register", methods=["POST"])
def register():
    data = request.get_json()
    email = data.get("email")
    password = data.get("password")

    if not email or not password:
        return jsonify({"error": "Missing email or password"}), 400

    if not email.endswith("@utdallas.edu"):
        return jsonify({"error": "Only @utdallas.edu emails are allowed"}), 403

    # Check if the password length is less than 6 characters
    if len(password) < 6:
        return jsonify({"error": "Password is too short. It must be at least 6 characters."}), 400

    print(f"Attempting to register user with email: {email}")

    try:
        response = supabase.auth.sign_up({"email": email, "password": password})

        # Proper error checking using attribute access
        if hasattr(response, "error") and response.error:
            error_message = response.error.message

            if "email already exists" in error_message.lower():
                return jsonify({
                    "error": "Email already registered",
                    "message": "Account with this email already exists"
                }), 409

            return jsonify({
                "error": error_message,

```

```

        message: "Account with this email already exists"
    }, 409

    return jsonify({
        "error": error_message,
        "message": "Registration failed"
    }), 400

# Handle case where user object might be None
if not hasattr(response, "user") or response.user is None:
    return jsonify({
        "error": "Unexpected error during registration",
        "message": "User not created"
    }), 500

return jsonify({
    "message": "Registration successful. Please check your email to confirm your account.",
    "email": email
}), 201

except Exception as e:
    return jsonify({
        "error": str(e),
        "message": "Registration failed due to server error"
    }), 500

```

- Blueprint(auth_bp): All authentication routes live in this Flask blueprint, which is registered in app.py with app.register_blueprint(auth_bp, url_prefix="/auth"). This keeps auth logic modular and isolated from other features.
- Configuration & Supabase Client: We retrieve a single Supabase client instance via Config.get_supabase_client(). This client handles all calls to Supabase's auth API—sign-up, log-in, and token verification.
- A representative endpoint is POST /auth/register, defined in auth.py. When a client sends a JSON payload containing "email" and "password", the handler:
 - Parses JSON input via request.get_json() to extract email and password.
 - Validates that both fields are present, that the email ends with @utdallas.edu, and that the password is at least six characters long—returning HTTP 400 or 403 for invalid input.
 - Calls supabase.auth.sign_up({ email, password }) to register the user.
 - Checks response.error for duplicate-email conflicts (HTTP 409) or other sign-up failures (HTTP 400).
 - Returns HTTP 201 with a confirmation message on success, or HTTP 500 for unexpected errors.
- Data Flow: Each route reads JSON input via request.get_json(), applies business rules (email domain, password length), invokes Supabase's Python SDK, and converts the SDK's response objects into clean JSON for our frontend. Errors are caught and mapped to clear messages and status codes so the client can handle them predictably.

This example shows how feature-specific blueprints, centralized configuration, and consistent validation patterns work together to create a maintainable and secure authentication layer.

Frontend Structure: <https://github.com/eshakumar0920/impulse>

The frontend of Impulse is built using React and TypeScript, with Vite as the dev server and Tailwind CSS for styling. The project is organized into folders with “meetups”, “profile”, “chatbot”, and “ui”, which helps keep things clean and easy to work with. Most of the reusable components (like buttons, modals, and form elements) are stored in the “ui” folder, while specific pages like “CreateMeetupForm.tsx” and “MeetupsList.tsx” handle user-facing features. Pages are built using Tailwind utility classes to keep the design consistent and mobile-friendly.

API calls are all handled in “api.ts” and, which is where we define all the routes to talk to our Flask backend. It takes care of adding auth tokens, formatting requests, handling timeouts, and showing error messages. For example, when a user joins an event, “eventsApi.joinEvent” sends a POST request with the user ID to the correct route. We also validate email domains and manage session tokens through “localStorage” here. Having all the fetch logic in one file keeps our components simple and focused on UI. For navigation and protected pages, we used “ProtectedRoute.tsx” to block access unless the user is logged in. Components like “EventCard.tsx” and “MapView.tsx” display event info and maps, while files like “AvatarSelector.tsx” and “NotificationSettings.tsx” let users customize their profile.

```
if (isLoading) {
  return <ContentLoader message="Loading meetups..." />;
}

if (meetups.length === 0) {
  return (
    <div className="py-8 text-center">
      <p>No meetups found. Create one!</p>
    </div>
  );
}
```

src/components/meetups/MeetupsList.tsx

The logic here shows the different UI states whether the data is still loading or if there are no meetups. It uses a custom ContentLoader component and conditional rendering for the UI to work.

```
const ContentLoader = ({ message = "Loading content..." }: ContentLoaderProps) => {
  return (
    <div className="py-20 text-center">
      <Loader2 className="h-8 w-8 animate-spin mx-auto mb-2 text-muted-foreground" />
      <p>{message}</p>
    </div>
  );
};
```

src/components/home/ContentLoader.tsx

Use case: This example keeps the UI consistent while waiting for the data, using a reusable spinner with the Tailwind and lucide-react icons.

```
<div className="grid grid-cols-1 gap-4">
  {meetupsWithCorrectPoints.map(meetup => (
    <div
      key={meetup.id}
      onClick={() => onMeetupClick(meetup.id)}
      className="cursor-pointer"
    >
      <MeetupCard meetup={meetup} />
    </div>
  ))}
</div>
```

src/components/meetups/MeetupsList.tsx

The code here maps through the meetup data and renders a MeetupCard for each one. The grid is styled with Tailwind and each card can be clicked, showcasing our dynamic navigation/actions.

```
<Button
  className="w-full flex items-center justify-center"
  onClick={handleJoinMeetup}
  variant={isJoinedLobby ? "outline" : "default"}
  disabled={isJoinedLobby || isLobbyFull || isLoading}
>
  {isLoading ? "Loading..." : isJoinedLobby ? "In Lobby" : isLobbyFull ? "Lobby Full" : "Join Lobby"}
</Button>
```

src/components/MeetupCard.tsx

Inside each card, the user can join a meetup lobby. The button changes state based on whether the user is logged in, already joined, or if the lobby is full.

```
return fetchApi('/auth/login', {
  method: 'POST',
  body: JSON.stringify({ email, password }),
});
```

src/services/api.ts

This is the login function inside authApi, used to send a POST request to the backend with user credentials. It also validates that the email ends with @utdallas.edu. This function is part of a centralized API service (fetchApi) that handles authentication tokens and error responses across the app.

3.3 Development Process

Planning:

- Initially our team began by creating a project timeline that provides a long-term structure to the project development by assigning goals to each weekly sprint of the development timeline.
- Weekly in-team meetings were held during development in order to properly communicate and determine both group and individual tasks for the current week, as well as, plan for future weeks.
- Following the Agile development process, Jira was utilized as our project management tool and our team made weekly updates to our scrum board as new tasks were assigned.

Debugging:

- Team members performed a majority of their own debugging while developing through analyzing error messages and utilizing online resources to help resolve issues.
- At times, bugs and issues were resolved as a team through online meetings if the bug or issue was not able to be solved individually or if multiple members had the same issue.

Collaboration Tools:

- Microsoft Teams was primarily utilized by our team to hold our in-team group meetings.
- Our team also created a Discord server in order to localize important information about the project and development. This server also provided an alternate method of communication where our team members could conveniently send messages to each other while working on the project.

Chapter 4: Performance

4.1 Testing and Validation

Frontend:

During development of the app, we used a variety of testing and validation approaches to ensure that the front-end was functioning as intended given different use cases and devices. First, for local development, while obtaining real time feedback from implementation, we used `npm run dev` to run the front-end and test component and UI behavior on localhost. This platform accounting for quick iteration, hot-reloading, and active debugging, allowed our team to be efficient and accurate when developing the user interface. Secondly, we used the Android Studio emulator that can simulate how the app will perform on a mobile device. The emulator helped us test the layout responsiveness and touch interactions, as well as navigation flows in an environment that resulted in a similar context to real world experience. Using the emulator allowed us to catch and correct mobile-specific UI issues, such as scaling proportionately, padding adjustments, and component alignments.

When debugging we worked with the browser developer tools, inspecting component states, console logs, and network requests. This fully capped our ability to isolate rendering errors, debug event handling, and confirm that data was properly flowing between components. Overall, with the tools we

used throughout the developments of the user interface, we validated that the UI's behavior was consistent across each environment, screen size, and context, for a seamless and consistent user experience for each user.

Backend:

During development, we manually verified each API route with Postman by sending requests to our primary endpoints—authentication (`/auth/register`, `/auth/login`, `/auth/verify`), events (`/api/events`), leveling (`/api/leveling`), and rewards (`/api/rewards`). For each endpoint, we tested both successful scenarios using valid JWTs and correctly formatted JSON payloads and failure scenarios such as missing or expired tokens, malformed input data, and edge cases like attempting to RSVP to a full meetups. This thorough manual testing allowed us to detect any discrepancies in HTTP status codes, response schemas, and error messages before moving on to automated checks.

To ensure proper performance and correct logic, we implemented automated unit tests using `pytest` to validate our API's behavior. Each `pytest` suite simulates HTTP requests against Flask's test client and verifies HTTP status codes, JSON responses, and the correct creation or rollback of database records. Test cases cover standard workflows—such as user registration, event creation, and RSVP—as well as boundary and error conditions, including invalid resource IDs, duplicate actions, and JWT authentication failures. Running these tests locally takes only a few seconds, allowing us to detect regressions immediately whenever new code is merged. Additionally, we developed end-to-end integration tests with Python's `requests` library, which send real HTTP calls to our live server at `localhost:5000`. These integration tests exercise the full request–response flow—from authentication headers and JSON payloads through to database persistence and error handling—ensuring the API behaves correctly under realistic conditions.

Beyond our automated test suite, we performed manual database validation by connecting to our local PostgreSQL instance with the `psql` command-line tool. After critical operations—such as user registration, event creation, and RSVP submissions—we executed SQL queries on the `users`, `events`, and `rsvp` tables to confirm that records were persisted accurately. We also verified that foreign-key relationships and cascade-delete constraints behaved as intended, ensuring data integrity across related tables.

During debugging, we configured Python's logging module to outputs `DEBUG`-level output and added strategic log statements around JWT validation and core business-logic branches. In parallel, we set breakpoints in VSCode within our Flask route handlers to step through code execution interactively. We also enabled the Flask Debug Toolbar to monitor each request's headers, database queries, and response rendering. This combination of logging, live breakpoints, and framework tooling allowed us to observe application behavior at runtime and isolate errors efficiently.

Finally, we ran the frontend (`npm run dev`) and backend (`flask run`) concurrently. In both desktop browsers and the Android emulator, we performed complete user flows—registering with an `@utdallas.edu` email, logging in, browsing events, and creating and joining meetups. Combined with manual API verifications via Postman, automated `pytest` test suites, direct database inspections using

psql, and structured application logging, these integration tests confirmed that our backend services are robust, secure, and production-ready.

4.2 Metrics Collected and Analysis

Development Phase Observations

The following metrics were observed during local testing using tools like Ngrok for tunneling and browser developer tools:

- **API Response Time:** Supabase API endpoints (e.g., /api/search, /meetups) consistently responded within ~100ms locally.
- **Error Rate:** No unexpected errors were observed during development. Common edge cases (like full lobbies or unauthenticated users) were caught and handled via toast notifications and conditional checks.
- **Join Meetup Success Rate:** When valid input and authentication were present, the user was successfully added to meetups.
- **Page Load Time:** Vite-powered React pages loaded in under 1 second based on Firefox Network tab logs, with transferred HTML around 1.15kB and 0–2ms server-side load times.
- **Latency:** Observed latency via Ngrok remained low (avg ~37ms).
- **Throughput:** API handled multiple concurrent /api/search requests (as seen in Ngrok logs) with consistent 200 OK responses.

Example Tools Used

- **Ngrok:** For external API testing and tunnel exposure (<https://8e47-70...ngrok-free.app>).
- **Firefox Developer Tools:** Tracked network performance, caching, and asset delivery timings.

Ngrok tunnel running locally, forwarding requests to localhost:5000 with real-time API request logs visible for /api/search endpoints.

```
Administrator: Command Prompt - ngrok http 5000
ngrok
( Ctrl+C to quit )
@ Want to hang with ngrokkers on our new Discord? http://ngrok.com/discord

Session Status      online
Account             [REDACTED]@gmail.com (Plan: Free)
Version             3.22.1
Region              United States (us)
Latency             37ms
Web Interface       http://127.0.0.1:4040
Forwarding           https://8e47-70-119-113-79.ngrok-free.app -> http://localhost:5000

Connections
-----
t1    opn    rt1    rt5    p50    p90
32    0        0.05   0.06   0.02   0.03

HTTP Requests
-----
21:51:18.546 CDT OPTIONS /api/search 200 OK
21:51:18.222 CDT OPTIONS /api/search 200 OK
21:51:03.286 CDT OPTIONS /api/search 200 OK
21:51:01.129 CDT OPTIONS /api/search 200 OK
21:50:54.807 CDT OPTIONS /api/search 200 OK
21:50:29.282 CDT OPTIONS /api/search 200 OK
21:50:29.510 CDT OPTIONS /api/search 200 OK
21:50:28.640 CDT OPTIONS /api/search 200 OK
21:50:28.288 CDT OPTIONS /api/search 200 OK
21:50:28.974 CDT OPTIONS /api/search 200 OK
```

Impulse app running on localhost:8080 with Firefox Developer Tools showcasing fast load times, asset caching, and successful GET requests.

The screenshot shows the Impulse app interface in a browser window at localhost:8080. The app has a dark theme and a navigation bar with icons for Home, Events, Meetups, Profile (Lvl 4), and Maps. A yellow notification bubble in the top right corner says "46 pts". Below the navigation bar, the "Network" tab of the Firefox Developer Tools is open, displaying a list of requests. The first request is a 200 OK GET request to localhost:8080 /, which is the main document. Subsequent requests include client.js, main.tsx, gptengineer.js, and @react-refresh, all of which are cached and return 304 status codes. The table below shows the details of these requests.

Status	Method	Domain	File	Initiator	Type	Transferred	Size	0 ms
200	GET	localhost:8080	/	document	html	1.15 kB (raced)	902 B	2 ms
304	GET	localhost:8080	client	script	js	cached	137.35...	0 ms
304	GET	localhost:8080	main.tsx	script	js	cached	1.94 kB	0 ms
200	GET	cdn.gpteng.co	gptengineer.js	script	js	cached	91.32 kB	0 ms
304	GET	localhost:8080	@react-refresh	script	js	cached	75.96 kB	0 ms

In a production setting, tools like Supabase logs, Firebase Analytics, or PostHog could help us collect these metrics to guide future improvements.

Metrics to Track Post-Deployment

If deployed to production, we would monitor the following to drive product decisions:

- **User Engagement:** Track events viewed, meetups joined, and session duration.
- **Conversion Rate:** % of users who join a meetup after viewing it.
- **Authentication Drop-Off:** % of users who abandon registration/login mid-way.
- **Map Interaction:** Engagement with Leaflet map (pin clicks, zooms, movement).
- **API Throughput & Latency:** Requests per second and time to respond under real load.
- **Resource Utilization:** Server CPU, memory, and database usage during peak and idle times.

Suggested Analytics Tools

To gather this data in production:

- **Supabase Logs** for backend request monitoring.
- **Firebase Analytics** or **PostHog** for frontend behavior tracking.
- **Uptime Monitoring Tools** for tracking error rates and response latency.

Chapter 5: Lessons Learned

5.1 Insights Gained

Throughout the developments of the Impulse app, we learned several lessons that extended beyond how to implement technical components. One of the most important takeaways was the need for clear and consistent communication. By having regular check-ins throughout our build process, sharing our ideas openly, and working collaboratively, we were able to stay aligned and resolve issues more efficiently. We also grew our understanding of Github as a version control, branching, and collaboration tool. This experience reinforced certain provided practices on project code management, including expectations on team-based environment, and maintaining a clean, organized workflow.

Another practical lesson we learned was that detailed planning early on reduces unnecessary or unexpected issues later on. By outlining the structure, roles, and tasks we were able to move forward with clarity around the project. Additionally, we also learned how powerful and flexible the backend framework Flask can be. We used Flask in ways that facilitated the rapid creation and seamless connectivity to the backend. We learned how easily we could implement tools for different functions and how it can be adapted in various types of projects. Overall, these lessons elevated our knowledge of the technical processes involved in project management and prepared us to work more effectively on our collaborative projects in the future.

5.2 Lessons from Challenges

As we worked on this projects, we met several obstacles mainly due to our inexperience with certain technologies. One major challenge was our lack of familiarity with Flask, PostgreSQL, and Supabase. These three technologies were key in the functioning of the app, but we had little to no experience with them at the start. We combated this by spending some time utilizing online tutorials, documentation, videos, and in particular, large language models. for quick reference of usage, to develop and ultimately debug our issues

On the front-end side we faced the issue of how to manage state and data flow of components, especially the dynamic aspects of the search bar, category filters, and map pins. Initially we had many issues where the UI would not render correctly or show the wrong data due to a lack of consistency in the in state handling. Similar to the back-end, we overcame these issues by watching videos and using online documentation. Through debugging and revisiting the state of our components we learned the importance of data flow when creating a reliable and responsive user interface and how managing state effectively is necessary to achieve seamless transitions. This journey gave us confidence and highlights the importance of carefully planning the behavior of components, but it also signifies the importance of testing every interaction in each pathway a user could take.

5.3 Skills Developed and Improved

During the entirety of this project, we built on a wide variety of technical and teamwork skills. On the technical side, we developed strong front-end skills such as component based UI, controlling dynamic state, and ensuring our application was responsive on mobile devices. Utilizing tools like npm, browser developer tools, and the Android Studio emulator, we received hands-on experience with testing real user interaction and debugging.

We also gained back-end experience with Flask, PostgreSQL, and Supabase that improved our ability to integrate with APIs and handle data. One of the habits we developed was detailed documentation—keeping track of the various initialization set ups, debugging efforts, and installation procedures. This was particularly helpful in documentation so we would not receive the same errors as a group and could deploy our effort quickly to resolve problems. On the collaboration side, we improved in areas like version-control tasks, resolving conflicts, and communication skills. Being able to use GitHub effectively put us in a position where managing branches, pull requests, and code reviews were conducted with minimal uncertainty. As a group, we learned how to better assign tasks based on team members' strengths and how to coordinate deeper by providing consistent updates and planning.

Regular standups, pair-programming sessions, and sprint retrospectives improved our communication and task planning, while self-directed learning (via official docs, targeted video tutorials, and LLMs) empowered each of us to tackle new technologies under tight deadlines. Collectively, these technical and teamwork skills have left us well prepared for future full-stack challenges.

Chapter 6: Future Work

6.1 Proposed Enhancements

- **Automatic Event Syncing:** Integrate with the official UTD event calendar so events populate in the app automatically, reducing manual entry and keeping the listings up-to-date.
- **Enhanced Interactive Map:** Improve the event map with real-time walking directions, cluster pins for events in the same area, and accessibility details like elevator and ramp locations.
- **Personalized AI Recommendations:** Use AI to suggest events tailored to users' interests, past participation, and what their friends are attending.
- **Timely Push Notifications:** Send alerts for RSVPs, upcoming events, and last-minute changes to keep users informed and engaged.
- **Expanded Chatbot Features:** Enable group messaging, general Q&A support, emoji usage, and an anonymous mode for students who prefer more privacy when engaging.
- **Event Participation Incentives:** Introduce a rewards system with check-ins and badges or achievements to motivate continued involvement.
- **Richer Profile Customization & Safety:** Allow users to add bios and list shared interests while incorporating two-factor authentication, content moderation, and block/report features for a safer community.
- **Admin Dashboard for UTD Staff:** Provide backend tools for university staff to create and manage events, view analytics, and monitor student engagement.

6.2 Recommendations for Development

Automate testing and deployment

Implement a CI/CD pipeline (using a third-party service such as GitHub Actions) that automatically runs your pytest suite and Postman/Newman collections on every push, plus a small set of end-to-end smoke tests. Automating these checks catches regressions early and ensures merges are safe.

Plan for scale.

Add an in-memory cache (using Flask-Caching) for frequently called endpoints such as event listings and RSVP checks. Once traffic increases, switch the cache backend to Redis or refactor those routes into standalone microservices or serverless functions. This ensures the API remains responsive as usage grows.

Improve onboarding & docs.

Maintain a live OpenAPI (Swagger) specification and a concise “Getting Started” guide covering environment setup, database migrations, and Git workflows. Clear, up-to-date documentation significantly reduces ramp-up time for new contributors and minimizes configuration errors.

Add observability.

Integrate structured logging and a third-party error-tracking service (e.g. Sentry) along with distributed

tracing via OpenTelemetry. This automated monitoring surfaces slow database queries or application crashes in staging or production before they impact students.

Keep agile habits.

Continue running short sprints with defined goals, holding quick demos, and conducting brief retrospectives. Rotate pair-programming sessions and code-review duties to spread knowledge and keep the codebase robust as team membership changes.

Chapter 7: Conclusion

7.1 Summary of Key Accomplishments

Built a visually appealing and functioning mobile application that achieves the main core goals stated in the project proposal

We successfully delivered a polished and functional mobile-first web application that fulfills our initial vision for fostering student engagement at UTD. The platform enables UTD students to securely register with their university email, discover and RSVP to campus events, and create or join student-led meetups in real time. Our gamified rewards system promotes continued participation through points and badges, while the responsive Vite + React interface ensures a smooth user experience across desktop and mobile. The backend, powered by Flask and PostgreSQL, features modularized APIs, robust schema migrations, and automated test coverage to ensure long-term stability and scalability. (same stuff)

7.2 Acknowledgements

Our team is very thankful for the advice provided by both Prof. Alagar and Thenn, his assistant for this course. The feedback provided during our team meetings was very helpful and provided guidance to our team in how we should properly approach developing our application. Thenn showed genuine interest in our project and often freely provided suggestions and advice based upon his own knowledge and experience. This advice and encouragement provided by both Prof. Alagar and Thenn helped motivate our team, especially during times where the project seemed overwhelming.

References

- [1] Meta, "React – A JavaScript library for building user interfaces," reactjs.org. <https://reactjs.org/> (accessed May 10, 2025).
- [2] Supabase Inc., "The open source Firebase alternative," supabase.com. <https://supabase.com/> (accessed May 10, 2025).

- [3] M. Grinberg, “The Flask mega-tutorial, part I: Hello, World!,” miguelgrinberg.com. <https://blog.miguelgrinberg.com/post/the-flask-mega-tutorial-part-i-hello-world> (accessed May 10, 2025).
- [4] Leaflet Contributors, “Leaflet – an open-source JavaScript library for mobile-friendly interactive maps,” leafletjs.com. <https://leafletjs.com/> (accessed May 10, 2025).
- [5] Postman, “Postman API Platform,” postman.com. <https://www.postman.com/> (accessed May 10, 2025).
- [6] pytest-dev, “pytest Documentation,” docs.pytest.org. <https://docs.pytest.org/en/stable/> (accessed May 10, 2025).
- [7] The PostgreSQL Global Development Group, “PostgreSQL: The World’s Most Advanced Open Source Relational Database,” postgresql.org. <https://www.postgresql.org/> (accessed May 10, 2025).
- [8] GitHub, “GitHub Actions Documentation,” docs.github.com. <https://docs.github.com/en/actions> (accessed May 10, 2025).
- [9] JWT.io, “JSON Web Tokens,” jwt.io. <https://jwt.io/> (accessed May 10, 2025).
- [11] Ionic Team, “Capacitor – Cross-Platform Native Runtime,” capacitorjs.com. <https://capacitorjs.com/> (accessed May 10, 2025).
- [12] Android Developers, “Android Studio – The Official IDE for Android,” developer.android.com. <https://developer.android.com/studio> (accessed May 10, 2025).

Appendices

Test script (example):

```
(py3.12) C:\Users\patri\Team-10>python test_api.py

=== Using fixed test user ID: 1 ===

=== Testing Event Creation ===
Status code: 201
Response: {
  "id": 28,
  "message": "Event created successfully"
}

=== Testing Get All Events ===
Status code: 200
Found 22 events
First event: {
  "created_at": "2025-03-05T23:16:28.398037",
  "creator_id": 2,
  "description": "Test description",
  "event_date": "2023-12-01",
  "id": 8,
  "location": "Test location",
  "organizer_xp_reward": 200,
  "semester": null,
  "title": "Test Event 1",
  "xp_reward": 50
}

=== Testing Get Event 28 ===
Status code: 200
Response: {
  "created_at": "2025-05-11T03:45:55.676954+00:00",
  "creator_id": 1,
  "description": "Learn Python basics and Flask development",
  "event_date": "2025-03-15T14:00:00",
  "id": 28,
  "location": "Computer Lab 101",
  "organizer_xp_reward": 200,
  "participants_count": 0,
  "semester": "Spring 2025",
  "title": "Python Coding Workshop",
  "xp_reward": 50
}

=== Testing Join Event 28 (User 1) ===
Status code: 200
Response: {
  "message": "Successfully joined the event"
}

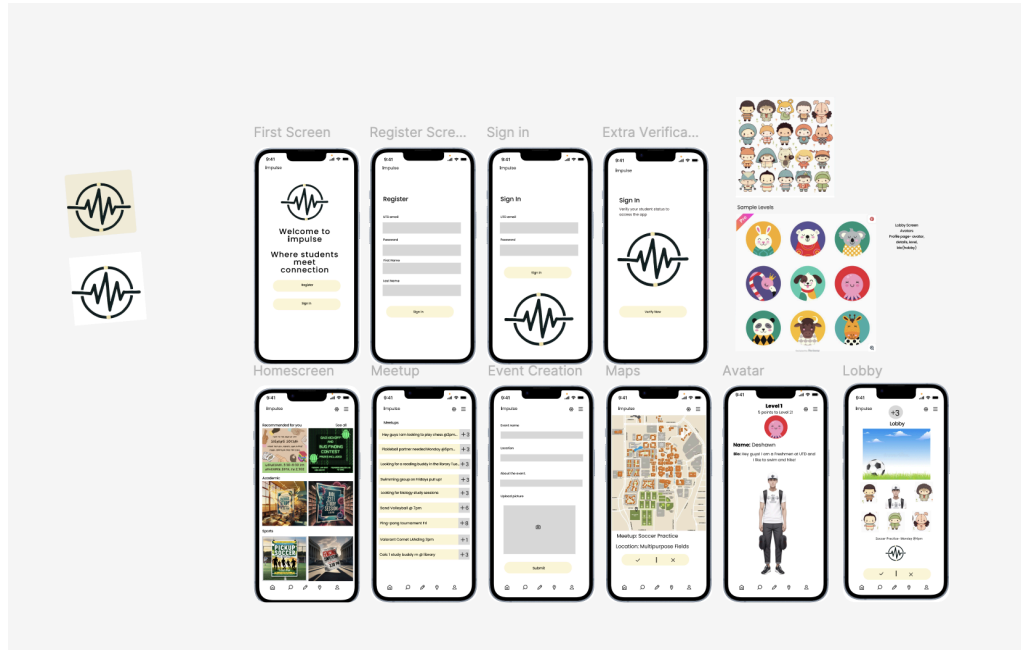
=== Testing Get Participants for Event 28 ===
Status code: 200
Response: [
  {
    "joined_at": "2025-05-11T03:46:01.796953+00:00",
    "user_id": 1
  }
]

=== Testing Leave Event 28 (User 1) ===
Status code: 200
Response: {
  "message": "Successfully left the event"
}

=== All tests completed ===

(py3.12) C:\Users\patri\Team-10>
```

Figma Screens



ChatBot Frontend:

