Team 4

Jason Anghad, Ben Bucaj, Nicole Lin, Ryan Rodriguez

Professor Levinger, Professor Meding

CS 391 S1

01 May 2025

Master Document: All Things Documentation

Internal Documentation: Aside from the in-project README, throughout our code, we've made sure to use clear, descriptive variable names and added helpful comments wherever things might get confusing. The goal was to make it easy for anyone reading the code (whether it's us later on or someone new) to quickly understand what's going on without needing to guess.

External Documentation: We've created comprehensive external documentation for every major file in the SparkBytes project. Following one of the suggestions posted by Professor Meding on Piazza, we used Google Docs to organize our documentation. As emphasized in the April 22, 2025 Documentation lecture, our documentation focuses on three core areas drawn directly from the lecture notes:

"■ Purpose and functionality: What the software does and how it does it.

■ Code structure and logic: How the code is organized and the reasoning behind it.

■ Usage instructions: How to use the software, including tutorials and examples."

This structure ensures clarity and consistency for anyone reviewing or extending the codebase. **The external documentation is provided below, with files listed in the same order they appear in the project structure (below this section is a diagram and appendix section):**

//

Documentation of files in src/app (COLOR CODED: YELLOW)

File: src/app/(dashboard)/landing_page.tsx

Purpose and Functionality

This file defines the main **Landing Page** for Spark!Bytes, which acts as the screen a user sees when visiting the website (after logging in / signing up). It introduces the application, shows a live map of sample events, and has sections explaining how Spark!Bytes works and why students should use it. It also handles location-based features to eventually recommend nearby events.

Code Structure and Logic

- **Map integration**: The Map component from @/components/map/Map is shown in the background using mocked mockEvents. Later, this can be swapped with real event data from the backend.
- **useUserLocation**: A custom hook is called to get the user's location and passes that data to the map using userPos={coords}.
- **Get Started Button**: The handleGetStarted function uses router.push('/login') to redirect users to the login page when they click the green button.
- **LocationPrompt**: This component shows loading or error messages depending on whether the location is loading or has failed to load.
- **How It Works Section**: Uses 3 steps to explain the app: Find Events, Claim Food, and Pickup.
- **Why Use SparkBytes Section**: Highlights Spark!Bytes benefits like reducing food waste, getting real-time updates, being location-aware, and offering quick access to food.

Usage and Integration

- This is the first page visitors see.
- It does not require login.
- Uses Map, LocationPrompt, and useUserLocation to make the page dynamic.
- Navigates users to the login page to continue into the dashboard.

File: src/app/api/auth/callback/page.jsx

Purpose and Functionality

This file handles the **callback route after authentication** (especially for Google login). When Supabase redirects back after logging in, this file checks if the user has a profile. If they don't, it creates one in the profiles table.

Code Structure and Logic

- Uses useEffect to run syncProfile() after page load.
- Uses supabase.auth.getSession() to get the current logged-in user.
- Checks if the user already has a profile in the profiles table.
- If not, inserts a new profile with fields like auth_id, email, and name.
- Has fallback logic using an RPC function create_profile_for_user in case regular insert fails.
- Redirects to /dashboard either using router.push() or fallback window.location.href.

Usage and Integration

- This page is **not directly visited by users**.
- It's called automatically by Supabase when someone signs in via OAuth.
- Prepares user data and then sends them to the main dashboard.

File: src/app/api/auth/register/route.js

Purpose and Functionality

This file defines a **custom registration API endpoint** used to register new users. It signs them up using Supabase Auth and stores their full name and role in the profiles table.

Code Structure and Logic

- Gets username, email, and password from the request body.
- Uses supabase.auth.signUp() to register the user.
- After success, inserts a profile in the profiles table with auth_id, email, full_name, and default role: student.
- If either the auth or profile insert fails, returns a JSON error.

Usage and Integration

- Called by the Sign Up page.
- Can be extended for error handling or custom roles later.

File: src/app/api/users/route.js

Purpose and Functionality

This is **nearly identical** to register/route.js. It's either a duplicate or a fallback endpoint that also handles user sign-up.

Code Structure and Logic

- Does exactly the same thing: registers a user with Supabase and adds them to the profiles table.

Usage and Integration

- If used, must be deduplicated. Otherwise, one should be removed to avoid confusion.
- Could be a backup for the main register route.

File: src/app/dashboard/page.tsx

Purpose and Functionality

This is the **main page for logged-in users**. It shows a live event map, a sidebar with navigation, and a list of food events. It also supports RSVP, event creation, and modals for managing events.

Code Structure and Logic

- **Location Hook**: Gets user's location with useUserLocation.
- **Modals**: Controls AddEventModal, EventDetailsModal, MyEventsModal, and FacultyCodeModal.
- **User Role**: Fetches user info and role from profiles table to conditionally allow faculty to add events.
- **Supabase Realtime**: Subscribes to changes in the events table (INSERT, UPDATE, DELETE) and updates the event list accordingly.
- **RSVP System**:
    - Loads RSVPs on mount.
    - Allows toggling RSVP by inserting or deleting from event_attendees table.
    - Updates event attendee count locally for better UX.
- **Event Creation**:
    - Faculty can click Add Event.
    - Modal form uses createEvent() to add a new one.

Usage and Integration

- Central hub of the SparkBytes app.

- Connects to Supabase to load, edit, and RSVP to events.
- Handles all real-time updates, user role control, and modals.
- Integrates with:
    - NavBar
    - Map
    - eventService.ts
    - Supabase events and attendee tables

File: src/app/login/page.jsx

Purpose:

This file defines the login page for Spark!Bytes. It lets BU users log in using email/password or Google. It also checks their account and handles redirecting them to the dashboard after login.

Main Features:

- **Email/Password Login** using Supabase
- **Google OAuth login**
- **Role selection** (student or faculty)
- **Validation for BU email address**
- **Redirect if user is already logged in**

How It Works:

- The component uses React's useState to keep track of form inputs (email, password, and user type).
- useEffect checks if there is an existing session. If so, the user is sent to /dashboard automatically.
- When the login form is submitted (handleEmailLogin), it:
    - Validates that the email and password aren't empty

    - Validates the email ends in @bu.edu
    - Sends a login request to Supabase
    - If login is successful, it checks if the user already has a profile in the profiles table. If not, or if something is missing, it still sends them to /dashboard.

Google Login Flow:

- handleGoogleLogin() uses Supabase's built-in Google OAuth provider.
- After successful login, the user gets redirected to /api/auth/callback, which handles session and profile creation.

Usage in App:

- This page is accessible at /login.
- It is the main login entry point for the app and connects to both Supabase and the custom profiles table.

File: src/app/signup/page.jsx

Purpose:

This file lets new users create an account. It validates that users are from BU and allows role selection. If the user chooses "faculty," they need to be verified through a modal.

Main Features:

- Full sign-up form for new users
- Validation for BU email (@bu.edu)
- Password and Confirm Password fields
- First and last name input
- Radio button selection for "student" or "faculty"

- Google login option
- Faculty verification with a modal (FacultyCodeModal)

How It Works:

- It uses useState to track user input for email, role, and modal state.
- If the role is changed to faculty, a modal pops up to verify with a special code.
- The handleSubmit() function:
    - Validates email domain
    - Validates passwords match
    - Builds a user profile with name, email, password, and role
    - Signs the user up with Supabase Auth
    - Inserts the profile into the profiles table
    - Redirects the user to the login page afterward

Google Login Flow:

- Works similarly to the login page, but with redirectTo: /auth/callback

Usage in App:

- This is the /signup page and is where new students and faculty register.
- Saves profile info into localStorage and Supabase for use during login and in the dashboard.

File Referenced for above 2 ^ (login and signup): Auth.css

Purpose:

This CSS file styles both the login and signup pages. It gives them a clean, modern, and centered look with BU-themed colors (red and green) and responsive form elements.

Highlights:

- .auth-container, .auth-card: Define layout and centering
- .auth-btn, .auth-btn.primary: Style buttons for login/signup
- input[type="text"], input[type="password"], input[type="email"]: Style all input fields
- .radio-group, .checkbox-grid, .button-row: Layout helpers for roles and preferences
- .logo-circle: Gives the animated breathing effect to the top logo on login/signup pages
- Mobile responsive design with max-width and padding

Usage:

- Automatically applied to the pages via import '../../Auth.css'
- Provides all styling for the login and signup forms, logos, spacing, error messages, and form controls

---

File: src/app/globals.css

Purpose:

This is the global CSS file for Spark!Bytes. It controls the base styles, themes, and reusable utility classes across the entire application.

Highlights:

- Uses **Tailwind CSS** with the directives: @tailwind base, @tailwind components, and @tailwind utilities
- Defines **custom color variables** (like --primary-green) for brand consistency
- Supports **dark mode** using @media (prefers-color-scheme: dark)
- Adds shared styles for:
    - Form inputs (.form-input)

    - Buttons (.btn-primary)
    - Modals (.modal-content)
- Enhances **date/time picker fields** for dark mode
- Adds smooth scrolling across the site (html { scroll-behavior: smooth; })
- Styles the Map container and popup boxes (#map-container, .mapboxgl-popup)

Usage in App:

- Automatically applied to all pages through layout.js and layout.tsx
- Used heavily in the login/signup pages, modals, and map components

---

File: src/app/layout.js

Purpose:

This is the layout file for older Next.js pages using JavaScript. It applies global styles and fonts to all pages underneath it.

Highlights:

- Imports global styles from globals.css
- Loads the **Inter** font from Google Fonts
- Sets the page metadata (title and description)

Usage in App:

- Wraps pages rendered under the older layout system. Most modern code now uses layout.tsx.

File: src/app/layout.tsx

Purpose:

This is the main layout wrapper for the app (TypeScript version). It applies the global font and includes a built-in **Toaster** component to show pop-up messages across the site.

Highlights:

- Loads and applies the **Inter** font
- Imports global styles (globals.css)
- Includes <Toaster /> from react-hot-toast for notifications
- Customizes toast style (background, colors, success/error icons)

Usage in App:

- Every page and component rendered in the app uses this layout
- Ensures visual and behavioral consistency everywhere

File: src/app/page.jsx

Purpose:

This is the basic home page of the app. It gives users two big buttons to either log in or sign up.

Highlights:

- Uses simple layout from auth.css
- Displays the logo (styled circle with “S!B” text)
- Has two buttons: “Login” and “Sign Up”
- Uses useRouter to redirect when buttons are clicked

Usage in App:

- This is the **root route /** for Spark!Bytes
- Acts as a gateway for new and returning users

File: src/app/page.module.css

Purpose:

This is a custom CSS module for special pages (like marketing or info pages). It defines a grid layout and customized button styles.

Highlights:

- Uses CSS variables for dark and light modes
- Creates a 3-row page layout using grid
- Styles buttons (.ctas a.primary, .ctas a.secondary)
- Styles list items and code blocks
- Responsive styling for mobile devices
- Footer styling and hover effects

Usage in App:

- This file is used by styled pages (possibly temporary or landing pages)
- Meant to style one-off pages that don't use Tailwind

File: src/app/page.tsx

Purpose:

This is the **new landing page** for Spark!Bytes, built with TypeScript. It's a full-featured introduction to the app and replaces older page.jsx usage.

Highlights:

- Includes:

    - Map section with mock events
    - Hero section with Get Started / Learn More buttons
    - “How It Works” steps
    - “Why Use Spark!Bytes?” value props
    - Faculty contact section
    - About section with developer bios
- Uses useUserLocation() to center the map
- Smooth scroll to About section using scrollIntoView()

Usage in App:

- This is the new **homepage of the site**
- Meant to impress users and help them understand the purpose and value of the app before signing up

//

Documentation of files in src/components (COLOR CODED: BLUE)

File: src/components/common/AddEventModal.tsx

Purpose and Functionality

This file defines a reusable modal component called AddEventModal, which allows faculty users to create new food events on the Spark!Bytes platform. It displays a form that collects event details such as title, time range, location, organizer info, food offerings, and dietary tags. The modal includes form validation and displays success or error messages using toast notifications.

Code Structure and Logic

- **Props:**
    - isOpen controls whether the modal is visible.

    - onClose is a callback to close the modal.
    - onSubmit is a callback to handle event creation with form data.
    - initialData allows preloading of form fields for editing.
- **Form State:**
    - State is stored in formData using the useState hook.
    - errors tracks field validation issues.
    - isSubmitting prevents multiple submissions while data is being sent.
- **Validation (validateForm):**
    - Checks all required fields like title, date/time, location, organizer info, and each food item.
    - Includes custom error messages (e.g., “End time must be after start time”, “Invalid email format”).
- **Submission Handler (handleSubmit):**
    - Prevents default form behavior.
    - If validation passes, calls onSubmit(formData) and shows toast.
    - Resets the form and closes the modal on success.
- **Form Fields Include:**
    - Title, start and end datetime, location dropdown using BUILDINGS, optional description.
    - Dynamically managed list of food offerings (with tags, temperature, serving size, etc.).
    - Organizer name, email, phone.
    - Optional max attendees and checkbox to make the event public.
- **Food Offering Logic:**
    - Users can add or remove multiple items (addFoodOffering, removeFoodOffering).

    - Each item supports optional description, dietary tags, temperature, quantity, and serving size.
    - toggleDietaryTag adds/removes dietary tags for each food item.
- **UI Design:**
    - Uses Headless UI Dialog component for accessibility.
    - Tailwind CSS classes style inputs, errors, and layout.
    - Includes loading spinner and button states during form submission.

Usage Instructions

- Used when a faculty user clicks “Add Event” on the dashboard.
- Valid form submission sends formData to onSubmit, which handles actual event creation via Supabase.
- Automatically shows error/success feedback via toast popups.

File: src/components/common/AddEventModal.test.tsx

Purpose and Functionality

This file contains Jest unit tests for the AddEventModal component. It verifies that the modal renders correctly, handles user input, validates form fields, and calls the appropriate callbacks for submit and close.

Code Structure and Logic

- Uses @testing-library/react and jest for rendering and simulating interactions.
- Mocks:
    - mockOnClose simulates closing the modal.
    - mockOnSubmit simulates submitting the event data.
- **Tests include:**
    - **Rendering:**

        - Modal displays when isOpen: true.
        - Modal doesn't render when isOpen: false.
    - **User Interaction:**
        - Clicking the close button triggers onClose.
        - Submitting an empty form shows required field errors.
    - **Form Validation:**
        - Submitting invalid email shows "Invalid email format".
        - If end time is earlier than start, shows "End time must be after start time".
    - **Successful Submission:**
        - Populates required fields like title, name, email, and date/time.
        - Adds a food item and submits it with dietary tags.
        - Verifies onSubmit is called with the correct structure.
    - **Food Offering Behavior:**
        - Can add/remove food items.
        - Validates dietary tag toggle logic using selects and buttons.
    - **Preloaded Data:**
        - Component populates form fields with initialData correctly.

Usage Instructions

- Run using npm test or yarn test.
- Ensures that any changes to the modal do not break core functionality like input validation, submission, or UI behavior.

File: src/components/common/EventDetailsModal.tsx

Purpose and Functionality

This file defines a modal component used to display detailed information about a specific event and manage RSVPs. The modal includes event details like time, location, food offerings, organizer information, and RSVP controls. It also verifies whether the current user is the faculty organizer and optionally allows them to edit the event.

Code Structure and Logic

- **Props**:
    - isOpen: Controls modal visibility.
    - onClose: Callback for closing the modal.
    - event: The DashboardEvent object to display.
    - isRsvpd: Boolean indicating if the current user has RSVP'd.
    - onToggleRsvp: Callback to RSVP or cancel RSVP.
    - onEditEvent: (Optional) Callback to edit the event if the user is authorized.
- **Authorization Check**:
    - Uses useEffect to fetch current user and profile from Supabase.
    - Sets isAuthorized to true only if:
        - User is authenticated,
        - User has role faculty,
        - User email matches the event's organizer email.
- **Conditional UI Rendering**:
    - Displays an "Edit" button only for authorized faculty.
    - RSVP button becomes disabled when event is at capacity and user hasn't RSVP'd.
- **UI Sections**:
    - Header with title and close button.
    - Event Info:

        - Title, time, location, and status badge (available or starting_soon).
    - RSVP:
        - Uses RsvpButton component to show attendee count and toggle RSVP.
    - Description:
        - Renders if event has a description.
    - Food Offerings:
        - Each item includes name, optional temperature, description, tags, quantity, and serving size.
    - Organizer Info:
        - Name, email, and optionally phone.
    - Additional Settings:
        - Max attendees (if specified), and visibility (public/private).
    - Action Buttons:
        - Close and Edit (if authorized, else disabled).

- **Toast Feedback**:
    - Error messages are shown via react-hot-toast if unauthorized users attempt to edit.

Usage Instructions

- Used inside a dashboard view when a user clicks on an event to view more details.
- Displays all information in a user-friendly modal.
- If the user is the faculty creator of the event, they can click "Edit" to begin editing via parent component logic.

File: src/components/common/FacultyCodeModal.test.tsx

Purpose and Functionality

This file defines unit tests for the FacultyCodeModal component using React Testing Library and Jest. It verifies that the modal behaves correctly in response to user interactions and input validation.

Code Structure and Logic

- **Mocks**:
    - mockClose: Simulates the closing behavior of the modal.
    - mockSuccess: Simulates successful validation.
- **Test Cases**:
    - **Modal Rendering**:
        - Confirms modal content is shown when isOpen is true.
    - **Validation Handling**:
        - Invalid code shows error message and does not call onSuccess.
        - Valid code (i.e., '123') triggers the onSuccess callback.
    - **Closing Modal**:
        - Clicking the cancel button calls onClose.
        - Clicking the X (close icon) calls onClose.
    - **Form Reset**:
        - Ensures the code input is cleared if modal is closed and reopened.

Usage Instructions

- Run with npm test or yarn test.
- Ensures secure access enforcement for faculty-only functionality.
- Prevents regressions in input handling or modal UI logic.

File: src/components/common/FacultyCodeModal.tsx

Purpose and Functionality

This file defines the FacultyCodeModal component, which acts as a gatekeeper modal requiring faculty to enter a verification code before creating events. It ensures that only authorized users can access event creation features.

Code Structure and Logic

- **Props**:
    - isOpen: Controls modal visibility.
    - onClose: Function to close the modal.
    - onSuccess: Callback triggered when a valid code is entered.
- **State**:
    - facultyCode: Tracks the user's input.
    - error: Stores validation errors (e.g., incorrect code).
    - isSubmitting: Tracks submission state (shows loading spinner).
- **Validation**:
    - Accepts only the hardcoded value '123'.
    - On correct entry, triggers onSuccess.
    - On incorrect entry, shows error message.
- **UI Components**:
    - Uses Headless UI Dialog for accessibility and consistent modal design.
    - Contains an input for the code, error feedback, and CTA buttons.
- **Extra UX Features**:
    - Modal reset on close (clears input and errors).
    - Includes static instructions for requesting access via sparkbytesbu@gmail.com.

Usage Instructions

- Shown when a user selects the "Add Event" button and has not verified their faculty status.
- Successful validation proceeds to the event creation modal.
- If the code is incorrect, the user receives immediate feedback and remains in the modal.

File: src/components/common/FavoritesModal.tsx

Purpose and Functionality

This file defines the FavoritesModal component, a modal interface that allows users to view and manage events they've marked as favorites. It displays a scrollable list of all favorite events with quick-access buttons to view full event details.

Code Structure and Logic

- **Props:**
  - isOpen: Controls whether the modal is displayed.
  - onClose: Callback to close the modal.
  - favoriteEvents: List of DashboardEvent objects marked as favorites.
  - onViewDetails: Callback triggered when a user clicks "View Details" for an event.
- **UI Layout:**
  - Uses @headlessui/react's Dialog for modal functionality and accessibility.
  - Modal is responsive and scrollable for long lists.
  - Each event card includes:
    - Title, location, time, and status (e.g. "Available Now").
    - Icons for visual aid (location pin and clock).
    - View Details button styled with hover effects.
- **Empty State:**
  - If favoriteEvents.length === 0, displays a placeholder message encouraging users to favorite events.
- **Design:**
  - Consistent Tailwind styling with rounded borders and backdrop blur.
  - Hover transitions and mobile responsiveness for modern UI experience.

Usage Instructions

- Used in the app wherever users can manage favorited events (likely tied to a heart icon).
- Users can open the modal to browse saved events and click to navigate to more detailed views.

File: src/components/common/LocationPrompt.tsx

Purpose and Functionality

This lightweight component provides visual feedback when the app attempts to access the user's location. It displays a loading spinner or error message at the bottom center of the screen.

Code Structure and Logic

- **Props:**
  - loading: Boolean that triggers a spinner and "Getting your location…" message.
  - error: String that, if set, displays a red error banner with the error message.
- **Conditional Rendering:**
  - If loading is true, a spinner and loading text are shown.
  - If error exists, a red alert with an error icon and message appears.
  - If neither, the component renders nothing.
- **Design:**
  - Fixed bottom placement with centered alignment.
  - Background colors differ based on status: dark blur for loading, red for error.
  - Uses Tailwind classes for styling and transitions.

Usage Instructions

- Meant to be conditionally rendered in parent components that request geolocation (e.g. when filtering events by proximity).
- Automatically disappears once location is fetched or error is resolved.

File: src/components/common/MyEventsModal.tsx

Purpose and Functionality

MyEventsModal is a modal component that lists all events the user has RSVP'd to. It serves as a dashboard feature allowing users to review and manage their upcoming commitments.

Code Structure and Logic

- **Props:**
  - isOpen: Boolean to toggle modal visibility.
  - onClose: Callback to close the modal.
  - myEvents: Array of RSVP'd events (DashboardEvent[]).
  - onViewDetails: Callback when a user wants to view more information about a specific event.
- **Modal Content:**
  - Each event card shows:
    - Title, location, time, number of attendees.
    - RSVP badge with a check icon ("You're Going").
    - Button to view detailed event information.
- **Empty State:**
  - If the myEvents array is empty, displays a message encouraging users to RSVP.
- **Design:**
  - Uses Dialog for modal behavior and keyboard accessibility.
  - Tailwind styling includes consistent layout, hover effects, and scrollable content.

Usage Instructions

- Invoked from the dashboard or nav bar to show the user's RSVP history.
- Users can quickly jump into event detail views or verify they are attending.

File: src/components/common/RsvpButton.tsx

Purpose and Functionality

This file defines the RsvpButton component, an interactive button used to RSVP or cancel an RSVP for a given event. It visually reflects the user's RSVP status and provides a clear attendee count beneath the button.

Code Structure and Logic

- **Props:**
  - eventId: Unique identifier of the event.
  - count: Number of attendees currently RSVP'd.
  - isRsvpd: Whether the current user is already RSVP'd.
  - onToggle: Function triggered when the button is clicked.
  - disabled (optional): Disables the button, usually when max capacity is reached.
- **Button Behavior:**
  - Shows **"RSVP"** with a plus icon if not RSVP'd.
  - Shows **"I'm Going"** with a check icon if RSVP'd.
  - Visually styled differently depending on RSVP status (green for RSVP'd, zinc for not RSVP'd).
  - Disabled when capacity is full and user has not RSVP'd.
- **Accessibility:**
  - Uses aria-pressed to reflect toggle state.
  - Descriptive aria-label based on RSVP state.
- **UI Design:**
  - Tailwind CSS for styling.

    - Text below the button shows the total count and adjusts for singular/plural.
    - Displays a warning if the event is full and the user hasn't RSVP'd.

Usage Instructions

- Typically used inside EventDetailsModal to allow a user to RSVP.
- Parent component must track RSVP state and attendee count to pass as props.
- onToggle should handle updates to Supabase or local state.

File: src/components/common/RsvpButton.test.tsx

Purpose and Functionality

This file includes Jest unit tests for the RsvpButton component. It verifies proper rendering and interactivity based on RSVP state, capacity, and user input.

Code Structure and Logic

- **Mocks:**
    - mockToggle: Simulates the parent function that handles RSVP toggling.
- **Test Cases:**
    - Renders correctly when not RSVP'd.
    - Renders correctly when RSVP'd.
    - Displays correct attendee count (singular/plural).
    - Triggers onToggle with the correct event ID when clicked.
    - Disables button and prevents toggle when disabled is true.
    - Does not show the "maximum capacity" message if the user is RSVP'd (even if disabled).

- **Testing Tools:**
    - Uses @testing-library/react for rendering and simulating user events.
    - Assertions check button content, accessibility roles, and text presence.

Usage Instructions

- Run tests via npm test or yarn test.
- Ensures logic and accessibility remain intact during UI or logic refactors.

File: src/components/common/SettingsModal.tsx

Purpose and Functionality

SettingsModal is a modal component used to display account-related information and allow users to sign out. It fetches additional user role data from Supabase and presents a read-only overview of the user's profile.

Code Structure and Logic

- **Props:**
    - isOpen: Controls whether the modal is visible.
    - onClose: Function to close the modal.
    - userData: Includes username and email to display in the modal.
    - onSignOut: Callback function to log the user out.
- **State & Fetching:**
    - userRole is fetched from the profiles table in Supabase via the authenticated user's ID.
    - Defaults to "Loading…" then updates to role or fallback if error occurs.
- **Modal Layout:**
    - Username, email, and account type displayed in styled containers.

    - Password is masked and accompanied by a “Change Password” button (non-functional placeholder).

    - A red “Sign Out” button signs the user out and closes the modal.

- **Error Handling:**
    - If user info or role cannot be loaded, displays fallback messages and logs errors to console.

Usage Instructions

- Invoked from the navbar or a profile button.

- onSignOut should handle Supabase sign-out and session clearing.

- Meant to offer a non-editable summary of the account, with potential future extension (e.g., changing password or role).

File: src/components/layout/Header.tsx

Purpose and Functionality

Defines the **Header** component for the SparkBytes app. This component appears at the top of the landing page and provides users with navigation options, including links to login and signup pages.

Code Structure and Logic

- **Logo Area**:
    - Displays the application name (APP_CONFIG.name) as a bold heading.

    - Clicking it routes users to the homepage (/).

- **Navigation Links**:
    - “Login” and “Sign Up” links displayed on the right-hand side.

    - Styled with Tailwind classes for hover and transition effects.

- **Layout**:

- Responsive container with padding for various screen sizes.
- Light background and subtle shadow for visual separation.

Usage Instructions

- Used on the **public-facing pages** of the application (e.g., home, login, signup).
- Included at the top of the layout in pages like index.tsx or _app.tsx.

File: src/components/map/Map.tsx

Purpose and Functionality

This file defines the **Map** component, an interactive, real-time map built using Mapbox GL JS. It visualizes event locations around the Boston University campus and includes lighting changes based on time of day, custom markers, and user location tracking.

Code Structure and Logic

- **Initial Setup**:
  - Default map settings: centered at BU, with zoom/pitch/bearing.
  - Lighting themes (dawn, day, dusk, night) update dynamically based on system time.
- **State & Refs**:
  - mapRef stores the Mapbox instance.
  - markersRef tracks all dynamically added markers.
  - currentLightPreset sets the current lighting mode.
- **Effects**:
  - On mount: initializes Mapbox map, applies light presets, and sets intervals to update lighting.
  - On events or userPos change: removes old markers, adds new markers for each event, and optionally centers on the user's location.

- **Helper Functions**:
    - getColorByStatus: maps event status to Tailwind CSS marker styles.
    - resetMapPosition: animates map back to the default BU view.
    - updateMapLighting: adjusts light preset and intensity.
- **UI Elements**:
    - Map container and reset button with icon.
    - Light preset indicator (e.g., 🌅 Dawn, ☀️ Day).
    - Legend for marker colors (available, upcoming, unavailable).

Usage Instructions

- Used in the dashboard or nearby event views.
- Expects events, onMarkerClick, and userPos props.
- Requires valid NEXT_PUBLIC_MAPBOX_TOKEN in environment config.

File: src/components/navigation/NavBar.jsx

Purpose and Functionality

The **NavBar** component renders a persistent navigation bar across authenticated pages of the app. It shows the user's initials, full name, Spark!Bytes branding, and a button to access the settings modal.

Code Structure and Logic

- **User Info (Left)**:
  Displays a circle with the user's initials and their name next to it.
    - Uses cached data from localStorage if available, otherwise queries Supabase.
- **Logo (Center)**:
    - Shows "Spark!Bytes" text, centered and styled in brand green.

- **Settings Button (Right)**:
    - Opens the SettingsModal component.
    - Includes a gear icon from Heroicons.
- **Dropdown Functionality**:
    - SettingsModal includes username/email display, sign-out button, and change password placeholder.
- **Sign-Out**:
    - Clears session using Supabase's auth.signOut() method and redirects to /login.

Usage Instructions

- Included in the layout of authenticated pages like /dashboard, /favorites, /nearby-events.
- Works in conjunction with SettingsModal to manage user preferences and account actions.

//

Other files:

File: src/constants/config.ts

Purpose and Functionality

This file defines core configuration constants used throughout the Spark!Bytes application. It centralizes static values such as app metadata, user roles, food item statuses, notification types, and API routes. These constants are used for permission logic, UI labels, data consistency, and route management.

Code Structure and Logic

- APP_CONFIG: Contains display information like the app name, description, and version.
- ROLES: Defines three user roles (admin, donor, and recipient) used for access control and feature visibility across the app.

- FOOD_STATUS: Represents the different stages a food item can be in:available, reserved, claimed, or expired.

- NOTIFICATION_TYPES: Categorizes system notifications as:
    - food_available: new food items available
    - food_claimed: items recently claimed
    - system: general messages

- API_ROUTES: Stores base paths for key internal API endpoints: food, notifications, and auth-related operations.

Usage Instructions

- These constants are imported anywhere in the code where status checks, role logic, or API routing is needed.

- Helps ensure consistency in string values throughout frontend and backend logic.

File: src/constants/eventData.ts

Purpose and Functionality

This file provides static data and TypeScript types for buildings and dietary tags relevant to food events. It ensures consistent naming, coordinates, and filtering options when users create or interact with events.

Code Structure and Logic

- Building Interface: Describes the structure of each campus building with id, name, coordinates, and address.

- BUILDINGS: Array of common BU locations that can be selected when creating events. Each entry includes geolocation data and display labels.

- DietaryTag Interface: Structure for individual tags with id, name, and category.

- DietaryCategory Type: Enum-like union of category types: allergens, preferences, intolerances, religious, special, and general.

- DIETARY_TAGS: A comprehensive list of dietary tags used for tagging food offerings. Organized by category:
    - **Allergens** (e.g., Nut-Free, Contains Dairy)
    - **Preferences** (e.g., Vegan, Halal, Keto)
    - **Intolerances** (e.g., Garlic-Free, Spicy)
    - **Religious** (e.g., Jain, Hindu)
    - **Special** (e.g., Kid-Friendly, Senior-Friendly)
    - **General** (e.g., Contains Alcohol, Caffeine-Free)

Usage Instructions

- Imported into forms and filters for event creation and dietary preference management.

- Provides dropdown options and validation checks in the AddEventModal and dietary settings.

File: src/constants/map.ts

Purpose and Functionality

Defines constants related to map rendering and event display on the interactive Mapbox map. These values control the map's visual style, default view, and color schemes used to indicate event availability.

Code Structure and Logic

- BU_CENTER: Default geographic coordinates centered on Boston University's campus.

- DEFAULT_ZOOM, DEFAULT_PITCH, DEFAULT_BEARING: Control the initial camera angle and zoom level for the Mapbox instance.

- MAP_STYLE: Specifies the Mapbox style URL used to render the map visuals.

- EVENT_STATUS: Defines two visual statuses for events:
    - available: event is currently live
    - starting_soon: will open soon

- EVENT_STATUS_COLORS: Associates each event status with a set of Tailwind classes for:
    - Background color (e.g., bg-green-500)
    - Text color (e.g., text-green-800)
    - Shadow effects for marker glow

Usage Instructions

- Imported by the Map component to style markers dynamically.

- Used to set visual lighting and differentiate between active and upcoming events at a glance.

File: src/hooks/useUserLocation.ts

Purpose and Functionality

This custom React hook provides real-time access to the user's geolocation using the browser's native navigator.geolocation API. It manages asynchronous location fetching, handles errors like permission denial or timeouts, and cleans up geolocation watchers on unmount.

Code Structure and Logic

- **State:**
    - coords: stores [longitude, latitude] if permission granted.
    - error: stores error messages related to location access.
    - loading: boolean that is true while location is being fetched.

- **Geolocation Flow:**

    - If geolocation is unsupported, sets an error immediately.
    - Uses navigator.geolocation.watchPosition to track user location and update state.
    - Includes custom error messages for common geolocation errors (permission, timeout, etc.).

- **Cleanup:**
    - Returns a cleanup function that removes the geolocation watch on unmount.

Usage Instructions

Import the hook and destructure the return values:

const { coords, error, loading } = useUserLocation();

File: src/lib/eventService.ts

Purpose and Functionality

This module provides service functions for interacting with the Supabase backend to manage events. It handles fetching public events, RSVPs, creating events, and transforming database records into a frontend-friendly DashboardEvent format.

Code Structure and Logic

- **fetchPublicEvents:**
    - Fetches all available and public events.
    - Joins event organizer profiles.
    - Transforms data into a usable format using transformEvents.
- **getUserRsvp(userId):**
    - Fetches all events the user has RSVP'd to using Supabase join queries.
    - Transforms results into DashboardEvent[].
- **createEvent(eventData, userId):**

    - Accepts event form data and user ID.
    - Formats coordinates and timestamps.
    - Inserts a new event into the Supabase database.

- **rsvpToEvent(eventId, userId):**
    - Validates event capacity before allowing RSVP.
    - Inserts RSVP record if under capacity.

- **cancelRsvp(eventId, userId):**
    - Deletes RSVP record from event_attendees.

- **transformEvents:**
    - Converts raw Supabase event rows into DashboardEvent objects.
    - Extracts and parses coordinates, formats time range, counts attendees, and maps organizer info.

- **formatTime:**
    - Helper function that returns human-readable time string like “1:30 PM”.

Usage Instructions

Import and use as needed in components:

await fetchPublicEvents();

await createEvent(formData, userId);

await rsvpToEvent(eventId, userId);

File: src/lib/supabase.ts

Purpose and Functionality

Creates and exports a typed Supabase client for use across the application.

Code Structure and Logic

- **Environment Variables:**
    - Reads NEXT_PUBLIC_SUPABASE_URL and NEXT_PUBLIC_SUPABASE_ANON_KEY from .env.

- **Typed Client:**
    - Uses Supabase's createClient method with non-null assertions to ensure keys are loaded.

Usage Instructions

Import supabase anywhere to make queries:

import { supabase } from '@/lib/supabase';

File: src/lib/supabaseClient.js

Purpose and Functionality

Provides an alternative (untyped) Supabase client for general use. This is functionally identical to supabase.ts but uses plain JavaScript.

Code Structure and Logic

- Loads environment variables for Supabase URL and anon key.

- Initializes the client using createClient.

Usage Instructions

Typically used in JS-based files or legacy modules:

import supabase from '@/lib/supabaseClient';

File: src/pages/_app.tsx

Purpose and Functionality

This file defines the custom App component for the Next.js application. It wraps all page components and provides a global entry point for injecting common settings like CSS imports or context providers.

Code Structure and Logic

- **Imports**:
    - AppProps from Next.js to type the props passed to the component.
    - Global styles from ../app/globals.css are loaded once for the entire app.
- **Component Function**:
    - App({ Component, pageProps }): A wrapper that renders each page dynamically based on the current route.
    - Returns the current page (Component) with its props (pageProps).

Usage Instructions

- Required in all Next.js projects to customize or extend the root App.
- This is where you'd typically add providers (e.g., for Redux, authentication, or global themes).

File: src/types/event.ts

Purpose and Functionality

Defines the type system for all event-related structures in Spark!Bytes, including basic event data, dashboard-enhanced events, form data, and validation errors.

Key Interfaces

- **Event**: Core event object with fields like id, title, location, status, and geocoordinates.
- **DashboardEvent**: Extends Event to include description, foodOfferings, organizer details, and RSVP-related metadata.
- **EventFormData**: Data shape used when creating or editing an event.

- **EventFormErrors**: Defines optional fields for validating form input.
- **EventFormProps**: Props expected by a form component, including onSubmit, onClose, and optional initialData.
- **RsvpStatus**: Describes a user's RSVP status with a boolean and a count.
- **FoodOffering**: Reused structure for a food item with dietary tags.

Usage Instructions

- Used in event creation/editing modals, dashboards, and data services across the app.
- Ensures type safety for form submissions, event displays, and database mappings.

File: src/types/index.ts

Purpose and Functionality

Defines foundational TypeScript types for key entities in the system: users, food items, and notifications.

Key Interfaces

- **User**: Represents a system user (admin, donor, or recipient), with optional metadata.
- **FoodItem**: Represents a food donation, including status, expiry, location, and timestamps.
- **Notification**: Represents a system-generated message targeted to a specific user, with read status and type (e.g., food available).

Usage Instructions

- Imported in backend services, API routes, or UI components when interacting with user, food, or notification data.

File: src/types/map.ts

Purpose and Functionality

Defines types used by the Map component and its subcomponents for rendering food events on a campus map.

Key Interfaces

- **Event**: Similar to the one in event.ts, but with optional distance string for map proximity.
- **MapProps**: Props accepted by the Map component, including a list of events, marker click callback, and optional user location.
- **MarkerProps**: Props expected by an individual marker on the map, consisting of an event and a click handler.

Usage Instructions

- Used exclusively in the interactive map component to structure and type-check event marker data.

File: src/auth.css

Purpose and Functionality

This CSS file defines the styling for the authentication-related pages (e.g., login and signup forms).

Main Style Classes

- **.auth-container**: Full-page layout centering the auth card on screen.
- **.auth-card**: White box for the form with padding, shadow, and rounded corners.
- **.logo-circle & .logo-text**: Branding circle with animated pulse.
- **Form Elements**:
    - Labels and inputs are styled with spacing and clarity for usability.
    - Includes responsive .form-row, .checkbox-grid, and .button-row.

- **Buttons**:
    - .auth-btn and .auth-btn.primary control color and hover behavior.
- **Extras**:
    - .error-msg, .signup-link, and form feedback styles for user experience.

Usage Instructions

- Automatically applied to /login and /signup pages through import '../app/globals.css' in _app.tsx.

Notes Taken on Feature Implementation Process:

Figuring Out How to Approach "Create Account"

- At the beginning of the project, we built a simple, barebones "Create Account" page.
- Our next goal was to verify that users were affiliated with BU (e.g., students, faculty).
- Initially, we used Supabase to store entered usernames and passwords, which provided basic functionality.
- We then added a requirement that all emails entered must end with "@bu.edu" to confirm university affiliation.
- However, we realized this check alone wasn't secure enough.
- To improve security, we decided to implement a "Continue with Google" option that redirected users to the Google login screen.
- This method was promising because all BU students and faculty should have access to a Google account linked to their @bu.edu address.
- It also ensures added security by triggering BU's Duo two-factor authentication during login.
- We encountered a challenge: by default, the Google login allowed any Google account, not just @bu.edu addresses.
- After consulting our TA, we learned that Google Cloud allows you to restrict sign-ins to your specific domain. So we configured it to accept only emails ending in @bu.edu.
- With this setup, the "Continue with Google" feature became a secure method for account creation.
- We then faced a decision: should we completely replace our manual "Create Account" form with Google sign-in?
- Ultimately, we chose to keep both options to give users the flexibility of either continuing with Google or creating an account manually.

- To make the manual method equally secure, we implemented two additional steps:
    - Required that the entered email ends in @bu.edu.
    - Added multi-factor authentication by sending a confirmation email to verify the address.
- This combination allowed us to offer both convenience and security in the account creation process.

Figuring Out How to Approach "Dietary Preferences"

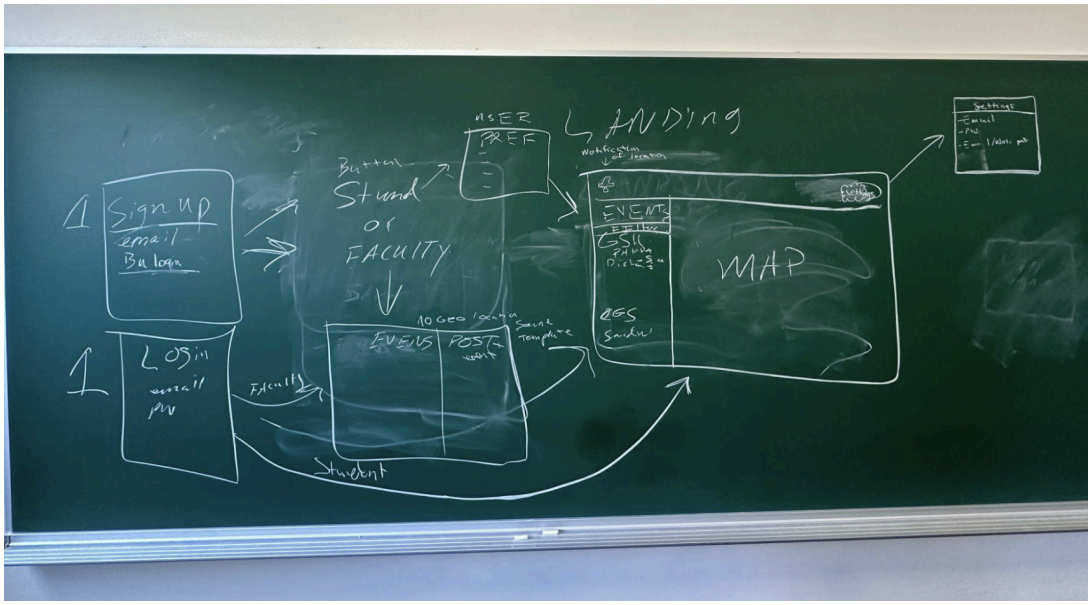
- At the start of our project, we created a basic dietary preferences page.
- After signing up as a "student" (i.e., general "user") you were directed to a page listing common dietary restrictions, each with a checkbox.
- The goal was to store each student's dietary preferences in Supabase for personalized event filtering.
- However, we realized that storing this data would require many columns and make it difficult to efficiently cross-reference events with specific dietary needs (event info table with dietary need table).
- This made filtering and matching users to relevant events overly complex and resource-intensive.
- So, we decided to remove the dietary preferences page entirely and shift this responsibility to event organizers.
- Instead of storing preferences per user, we now have organizers specify the dietary accommodations offered for each event in the event creation form.
- Users can then filter events by dietary restriction directly from the landing page.
- This solution is more consistent with how our other *intended* filters (like distance) work, since all filters now reference event details.
- It also streamlines the process and simplifies data management while still giving users control over the kinds of events they see.

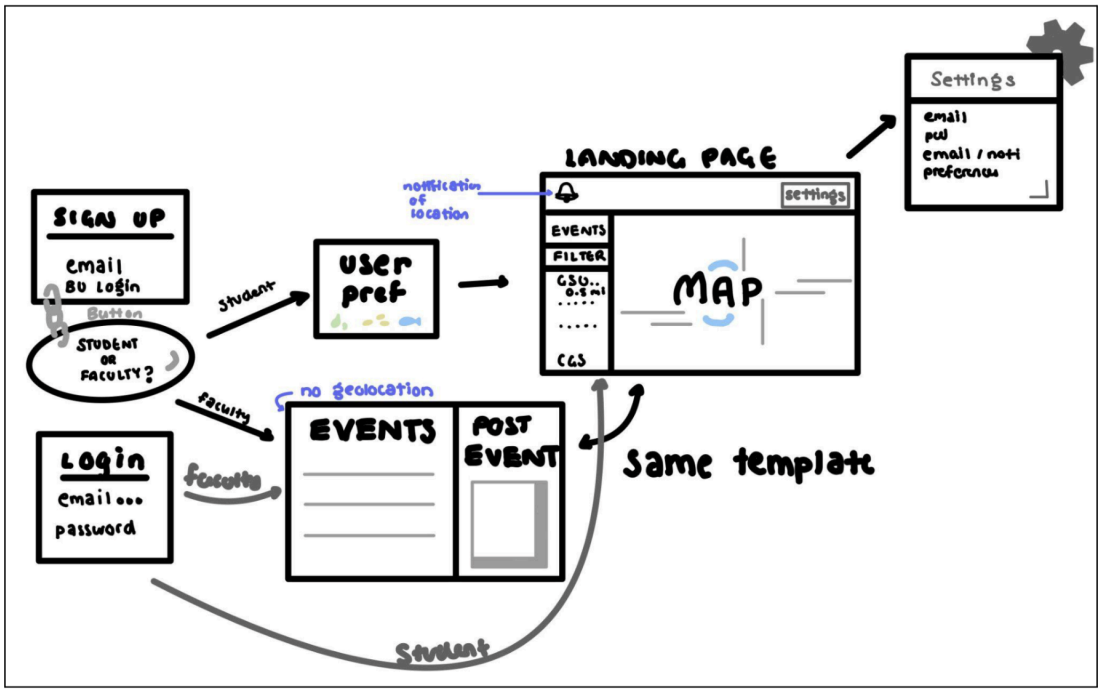
Figuring Out How to Approach "Faculty / Event Organizer Verification"

- Verifying faculty or event organizers was one of the most challenging parts of our project.
- We did not have access to any internal BU systems or official databases listing faculty emails or club leadership.
- Because of this limitation, we decided on a somewhat manual verification process for users who try to sign up as faculty/event organizers.
- When someone signs up under one of these roles, an email is sent to a "SparkBytes admin account"
- From there, one of us manually verifies the user's identity by checking publicly available information, such as their official BU profile or club listing (which also lists email).
- If the email they used matches what's publicly listed, we proceed to the next steps.
- The user is then required to complete multi-factor authentication (MFA) via their email.
- They are also prompted to enter a "Faculty Code Button" on the signup page. They then are given 1 of 10 codes stored in the database to be entered
- Once the entered code is validated, the account is approved as a verified faculty or event organizer.

Diagrams

While most of our documentation is text-based, our group did incorporate diagrams into our workflow (just not in the traditional, formal sense). Since none of us were experienced with creating polished diagrams before the April 22 Documentation lecture (which was just a week before the project deadline), we relied more on informal visual planning. During in-person meetings, we sketched out ideas on whiteboards and chalkboards to visualize the structure and flow of the final project. We also created a few digital diagrams along the way. Examples of both are included below:





Appendix

AI/GenAI Usage Disclosure (ChatGPT)

Extent of Use:

ChatGPT was used in a limited and responsible manner during the development of Spark!Bytes. Specifically, it was used for:

- **Troubleshooting and debugging** specific coding errors (e.g., fixing syntax issues, resolving unexpected behavior).
- **Clarifying errors** in React and Supabase integration during development.
- **Assist with internal code documentation,** such as generating docstrings for complex components.
- **External documentation** support (e.g., structuring summaries of component purpose, logic, and usage instructions in accordance with the April 22 lecture on documentation best practices).

Why It Was Used:

We used this resource to save time during repetitive or unclear debugging tasks and to assist in organizing documentation content clearly and efficiently.

How It Helped:

- It provided quick insights when searching StackOverflow or documentation wasn't yielding fast answers.
- It helped organize our external documentation using the Purpose / Structure / Usage format discussed in class.
- It was *not* used to generate entire codebases or substitute for original development work. All code was student-authored, with ChatGPT used to troubleshoot or clarify logic.

Policy Compliance:

As per the CDS policy (which is referenced on our CS 391 S1 syllabus) on the use of AI tools and the April 22 documentation lecture by Professor Meding, we verified the accuracy of all AI-assisted content, especially in documentation. We cite this use here in accordance with the course and university guidelines.

Representative Prompt Examples (from our exchange with ChatGPT):

- "Explain why this React component isn't rendering correctly"
- "Generate documentation for this TypeScript file using the format: Purpose / Code Structure / Usage Instructions"
- "Help me summarize how this modal works and what props it expects"