# Boop Design

JIB Team 0318

Grace Rarer
Jiale Zheng
Gunnar Andra-Thomas
Sanjana Jampana
Juan Caicedo
Porter Hunley

Client: Professor Stallworth and Dr. Shelley

# Table of Contents



# Table of Figures

# Terminology

**Angular**: a frontend framework created by Google for creating web-apps in TypeScript.

**Bcrypt:** the password hashing and validation algorithm used by Boop.

**CRUD:** the four basic actions of most data-access APIs are create, read, update, and delete.

**ExpressJS:** the NodeJS web-server library used for Boop's backend.

**Gravatar:** a service that connects your email address to your profile image across different websites and apps.

**NodeJS:** a runtime environment that allows server-side software to be written using JavaScript, or languages in the JavaScript ecosystem such as TypeScript.

**PostgreSQL:** an open-source implementation of the SQL relational database standard.

**Progressive Web App (PWA):** A website that uses a Service Worker script to behave like a native application. PWAs use caching so they do not have to reload the entire webpage for every session, and support Push Notifications.

**RESTful API:** web service that utilizes the Representational State Transfer (REST) architecture to handle requests on the frontend.

**Web Push Notification:** push notifications sent to a user's web browser.

**Universally Unique Identifier (UUID):** a set of standards for producing identifiers or primary keys that are intended to be globally unique.

# Introduction

**Background**

Our project is Boop, the digital equivalent to crossing paths with a friend in the dining hall. We aim to have users utilize our platform to encourage spontaneous social messaging in their lives. Users registered to the service will receive spontaneous push notifications, reminding them to talk to a friend, encouraging a friendly, lively environment. Other features include profile creation and editing and control over notifications. We utilize a Progressive Web App (PWA) to present this platform on both desktop web browsers and mobile devices. We are hopeful that this idea will help users feel just a bit more connected in this increasingly online world.

**Document Summary**

The **System Architecture** section will highlight the static and dynamic elements of the Boop application, providing context into the structure and relationship between all components available.

The **Data Storage Design** section will describe the database section of our app, which consists of PostgreSQL integration and stores profile and account information, as well as other features.

The **Component Design Detail** section will show a more detailed glance into the components of our design, highlighting in greater depth the relationships between all available features.

The **UI Design** section presents the current state of the User Interface, and details the visual design decisions to make the app experience seamless.

# System Architecture

**Introduction**

We have included some diagrams explaining the components of our system architecture and showing examples of how these components work together in some of Boop's features.

Most of the features of Boop are user-initiated; users open our web-app to update information about themselves, manage their friends lists, and view information about their friends. However, Boop's headline feature, push notifications prompting users to chat with their friends, occurs spontaneously when the backend server sends a push notification without being prompted by a user's actions.

**Static System Architecture**

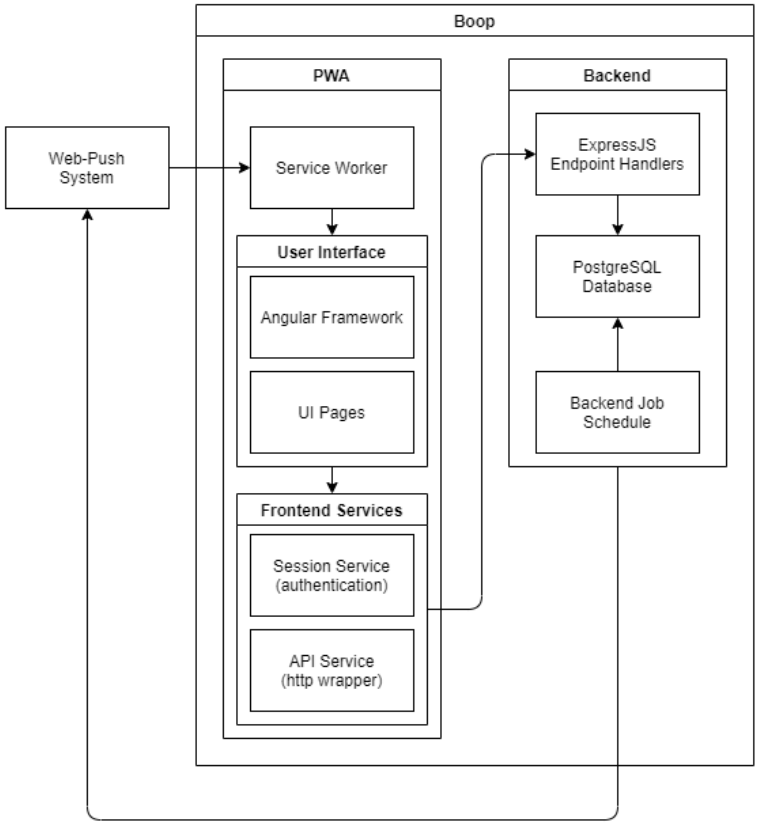



*Figure 1: Static System Architecture*

The frontend of our application is a Progressive Web App (PWA), meaning that it has a service worker script that is always running, even when the app is closed. This service worker improves performance by caching the application so that the browser does not have to download the app every time the user visits the page, but more importantly, the service worker is always available to receive push notifications [1]. The visible parts of our user interface are components created with Angular, and behind the scenes we have custom frontend services that wrap HTTP requests on behalf of the UI components and automatically handle authentication of each request. The frontend communicates with the backend through a RESTful API by sending JSON objects over HTTP.

HTTP requests made by the frontend are handled by our backend server using the ExpressJS framework. The handler functions rely on a Postgres database to store and retrieve information in response to the HTTP requests. In addition to handlers triggered by HTTP requests, the backend also has a cycle of scheduled jobs that it repeats every few minutes. This includes clearing out expired tokens from the database, and is also how we schedule the sending of spontaneous push notifications.

Push notifications are sent using the web-push protocol. Our server sends a request to the web-push service created by the organization that created the user's browser, which then forwards the push to the user's device. The service worker is able to receive these notifications, show them to the user, and open the app when the notification is clicked.

**Dynamic System Architecture**

This first sequence diagram shows how the user can check for and respond to friend requests. This is a straightforward application of CRUD principles (create, read, update, delete), with the backend server being an intermediary between the web-app and the database. Services on the frontend and Express handlers on the backend server act as intermediaries between the user interface and the database.

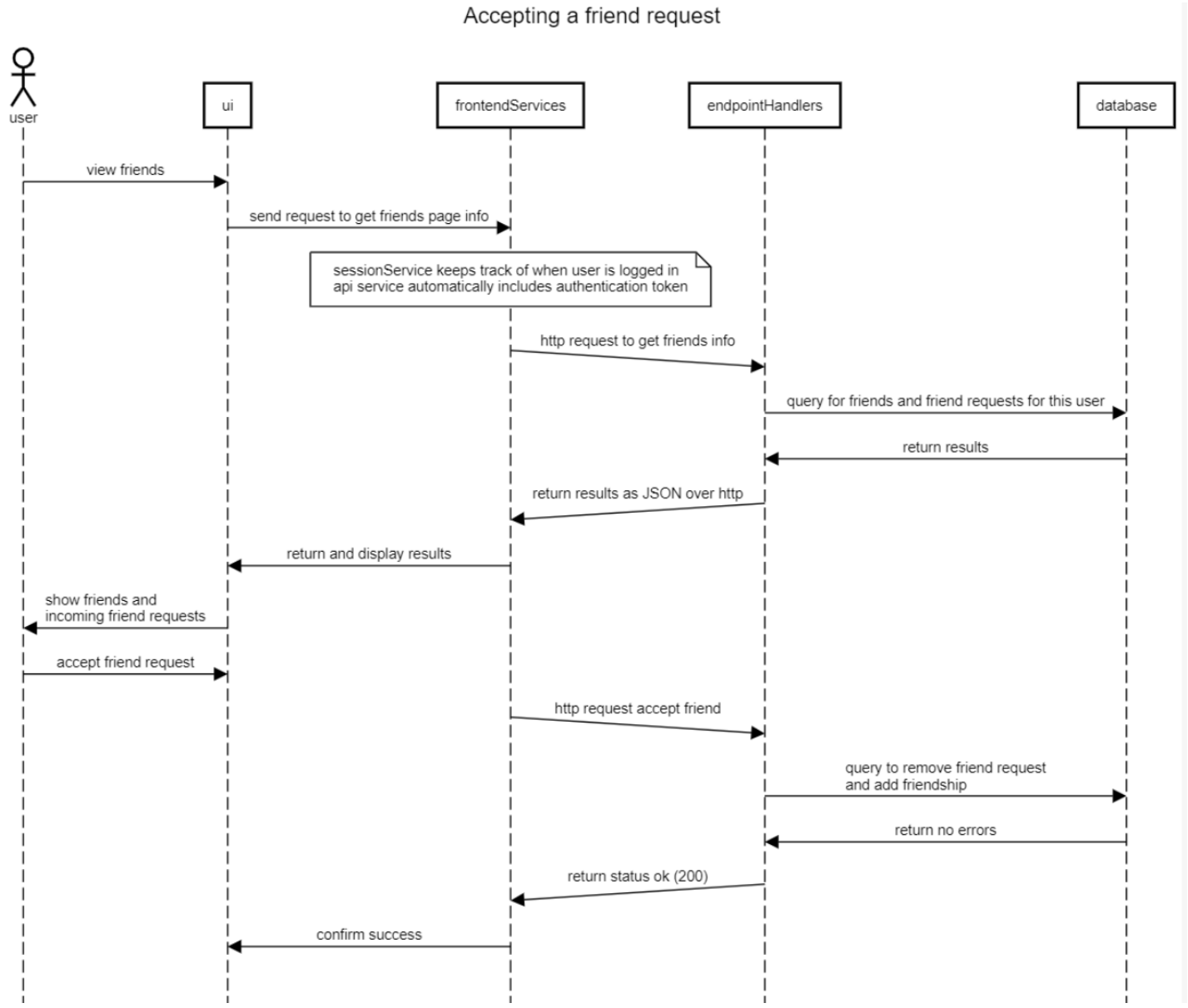



*Figure 2: Dynamic System Architecture for Accepting a Friend Request*

This second sequence diagram shows the process of spontaneous push notifications, which are started by a scheduled job rather than by a user action.

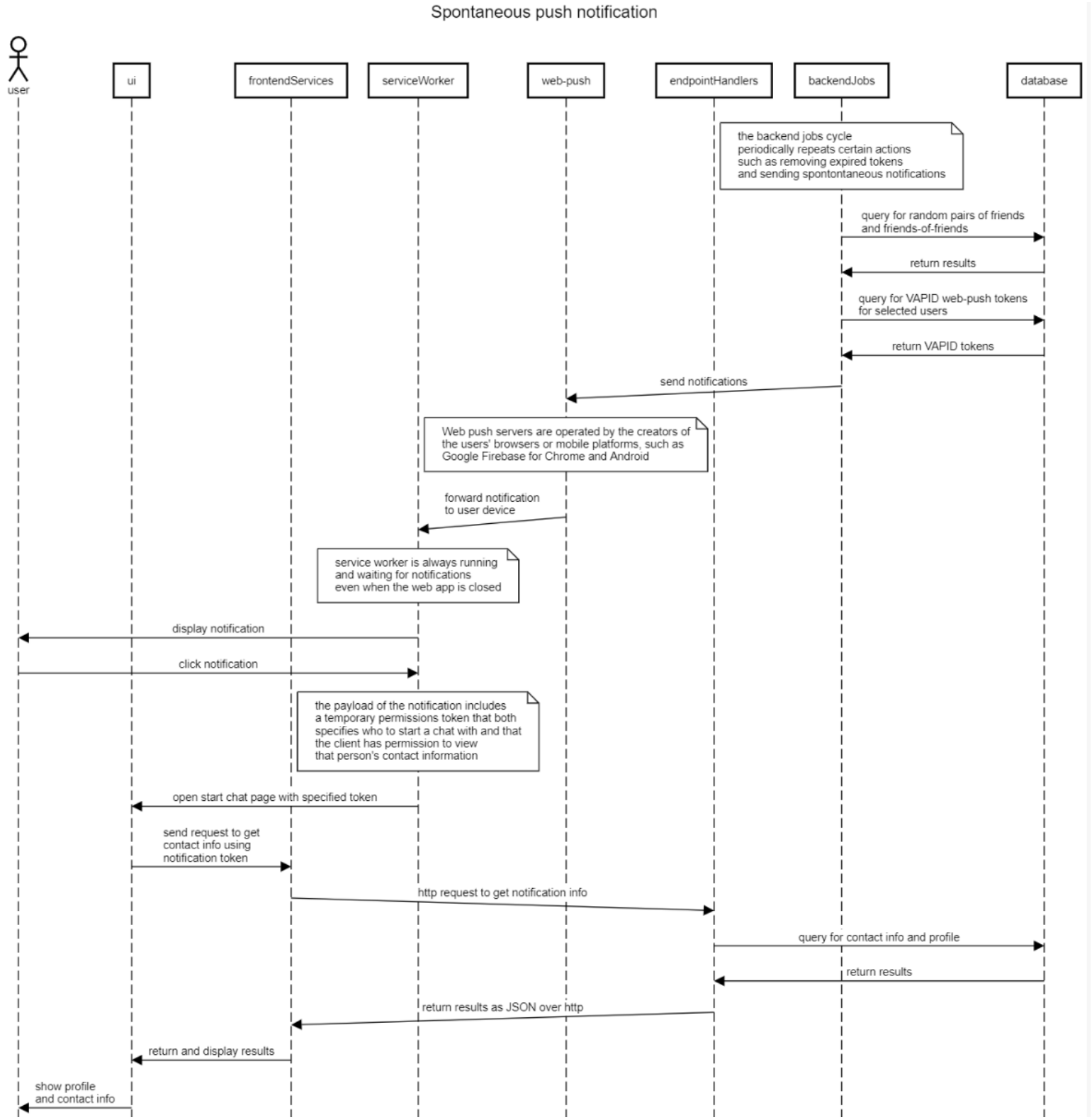



*Figure 3: Dynamic System Architecture for Spontaneous Push Notifications*

# Data Storage Design

**Introduction**

Although Boop is all about communication, we do not host users' conversations on our own platform. Boop's backend server only has to keep track of users' information and the friendships between users. To store this data, we use a PostgreSQL database.

**Entity Relationship Diagram**

This diagram shows the schema used by Boop's PostgreSQL database.

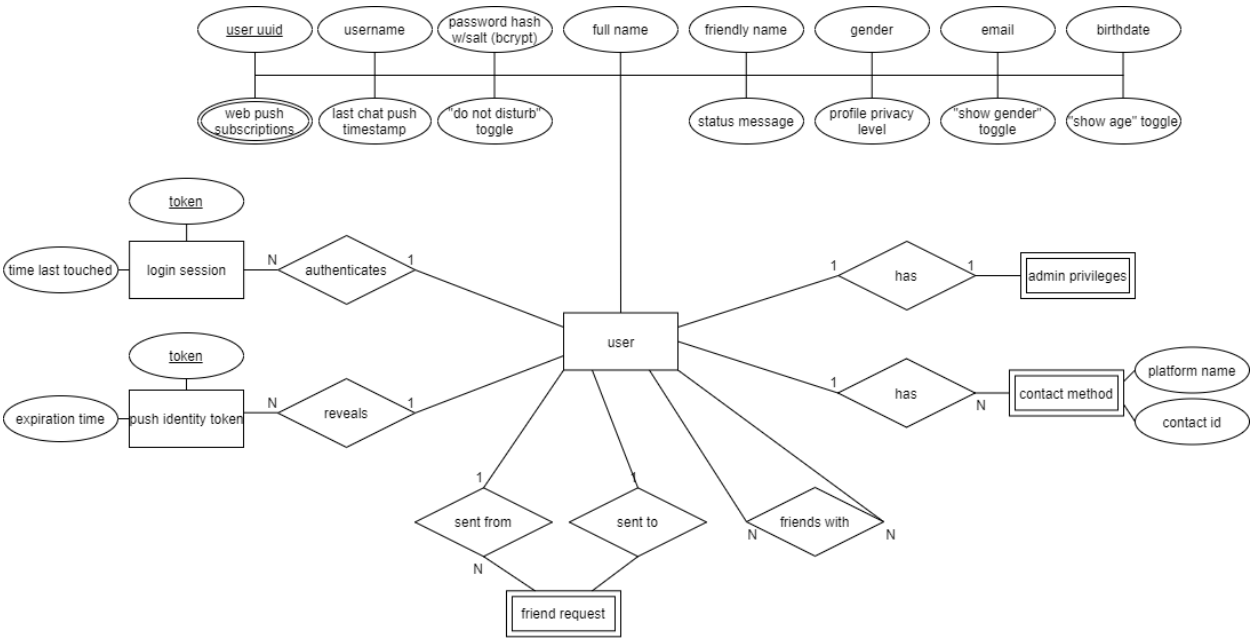



*Figure 4: Database Entity Relationship Diagram*

Since users are allowed to change their username, a universally unique identifier (generated with UUID v4) is used as the user's permanent primary key.

Since web push notifications are always sent to all of the devices associated with a given user, we always retrieve a user's notification subscriptions as a group rather than querying for just one. Therefore, web push subscriptions are represented as a multi-value column of JSON objects rather than as separate entities.

Our implementation of the database denormalizes the "friends with" relationship by representing each friendship as two rows. For example, if Alice is friends with Bob, we would have one row for "Alice is friends with Bob" and another row for "Bob is friends with Alice". We must create or delete both rows at the same time to maintain consistency, but this duplication makes it faster and easier to treat the network of friendships as an undirected graph, such as when calculating the "friends of friends" data.

Login session tokens are used to both identify and authenticate users who are logged in. Push identity tokens are temporary tokens that only allow a client to view the "start chat" page for a particular person. For both of these tokens, the keys are based on universally unique identifiers.

**Files**

Our backend does not store any files. User profile images are processed, stored, and served by Gravatar, a third-party service, so when we display an image the backend only needs to generate the Gravatar URL using a user's email address [2]. The frontend asset bundle will include images such as the Boop logo and the icons for popular social messaging platforms.

Europe's GDPR and other data privacy laws include a "right to data portability" which requires that users should be able to export their data in a format that can be easily understood [3]. To comply with this requirement, we give users the ability to download their information as a JSON file. This file is generated on-demand using information in the database.

**Data Exchange**

Our frontend and backend communicate via a RESTful API using JSON over HTTP. Both the Angular frontend and ExpressJS backend use middleware that automatically converts data to and from a JSON representation. The frontend and backend are both written in TypeScript and both depend on a separate NPM package called boop-core that provides them with shared type definitions for the objects that are sent through the API.

For example, here is the type definition for the request and response objects for logging in:

```
1  export type LoginRequest = {
2    username: string;
3    password: string;
4  };
5
6  export type LoginResponse = {
7    userUUID: string;
8    sessionToken: string; // used to verify identity when making requests to backend
9  };
10
```

You can see all of these shared type definitions under core/src/datatypes in our GitHub repo [4].

We also use a wrapper around our backend ExpressJS handlers so that exceptions thrown by backend endpoints can be automatically caught, transmitted with an appropriate HTTP error code, and handled by the frontend.

**Security**

Security best practices require that passwords never be saved in plain text; instead, they should be validated using an approach called "hashing and salting" [5]. This is where a random component called a salt is used to produce a different password hash for each user even when two users have the same password. We use the "bcrypt" password-hashing algorithm. The database stores the bcrypt "hash", which is a combination of the password hash and the salt as one string, and this hash is used to validate the password when a user logs in. Bcrypt's unique trait is that it is intentionally slow in order to prevent attackers from being able to easily crack password hashes [6]. By being a relatively expensive hashing algorithm that can be made even more expensive as computing hardware becomes more powerful, bcrypt is a future-proof defense against brute-force hashing attacks as well as "rainbow table" attacks.

When a user logs in, a session token is stored in the database and sent back to the client. The client stores this token in local storage on the user's browser and automatically transmits the token as an HTTP header when making API requests to our backend. The server compares this token to the sessions stored in the database to determine which user made the request. Sessions expire if they are not used for a long period of time (currently 30 days).

The "start chat" page required a different authentication system, since users might receive and open a Boop notification while not logged in to the app. When these notifications are sent, we create a special temporary token that allows the client to view the "start chat" page for a specific user, and this token is transmitted in the metadata of the push notification, which is encrypted before being sent to the web-push servers.

# Component Design Detail

**Static Component Diagram**

This diagram shows the static relationships between the components of our web-app frontend and Node backend. On the frontend, most major components of the UI are standalone screens, with the Angular Router determining which screen to show based on the current URL. The controller code for these UI screens accesses data from the backend via the API Service and Session Service, which automatically handle authentication for logged-in users.

On the backend, the ExpressJS server that handles requests from the frontend is divided into units called Express routers, which consist of handler functions for specific API endpoints. The Express app and its handler functions are stateless, with all data being stored in our Postgres database. A few tasks that are not based on user input, such as removing expired tokens from the database and sending spontaneous push notifications, are scheduled to repeat every few minutes.

Since Boop does not use an object-oriented design architecture and does not have a class-based data model of interacting objects, we chose not to use a standard UML diagram. The diagram below shows the connections between the functional components of our application, but most of the backend components are stateless collections of functions rather than true objects. For simplicity and legibility, this diagram omits smaller sub-components, such as loading bars and confirmation dialog pop-ups that are used as shared components of UI screens, or SQL query scripts that are used by multiple Express routers.

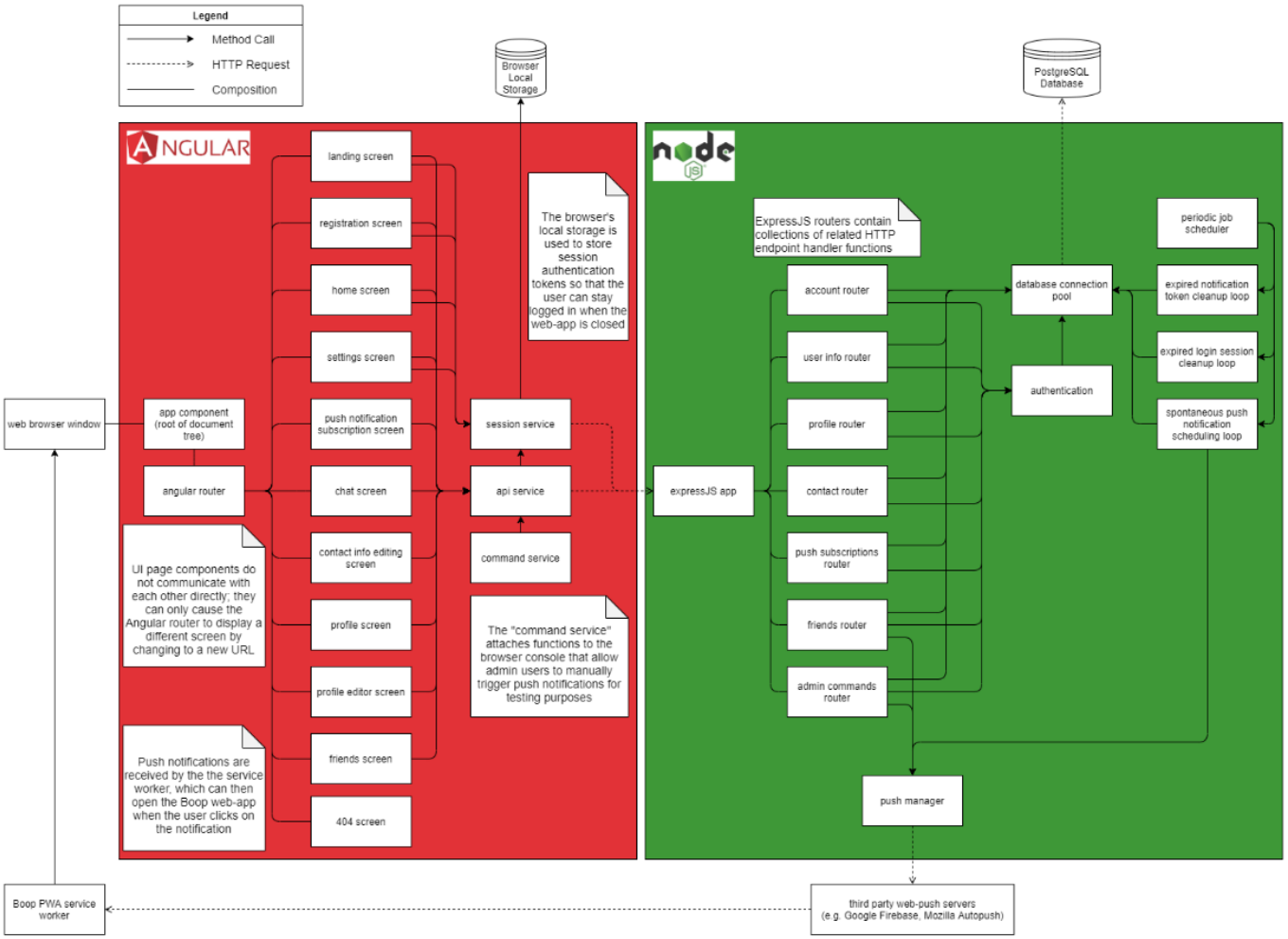



*Figure 5: Static Component Design Diagram*

**Dynamic Component Diagram**

When we develop features that are based around user interaction, we can safely ignore several layers of indirection and middleware to get a clearer picture of the flow of information. For example, although the static diagram shows that most UI screens do not directly communicate, it is more useful to think about the user interface in terms of links between pages rather than thinking about the implementation details of the Angular routing.

The diagram below depicts components related to viewing and editing user profiles and shows how these components interact from the perspective of user-driven interactions. For example, a user who is logged into the home screen could update their status message, navigate to their own profile, edit their profile description, navigate through their friends lists to see other users' profiles and send a friend request to another user who would be notified by a web-push notification. These pages form a vertical slice of our application that represents how most of Boop's features function.

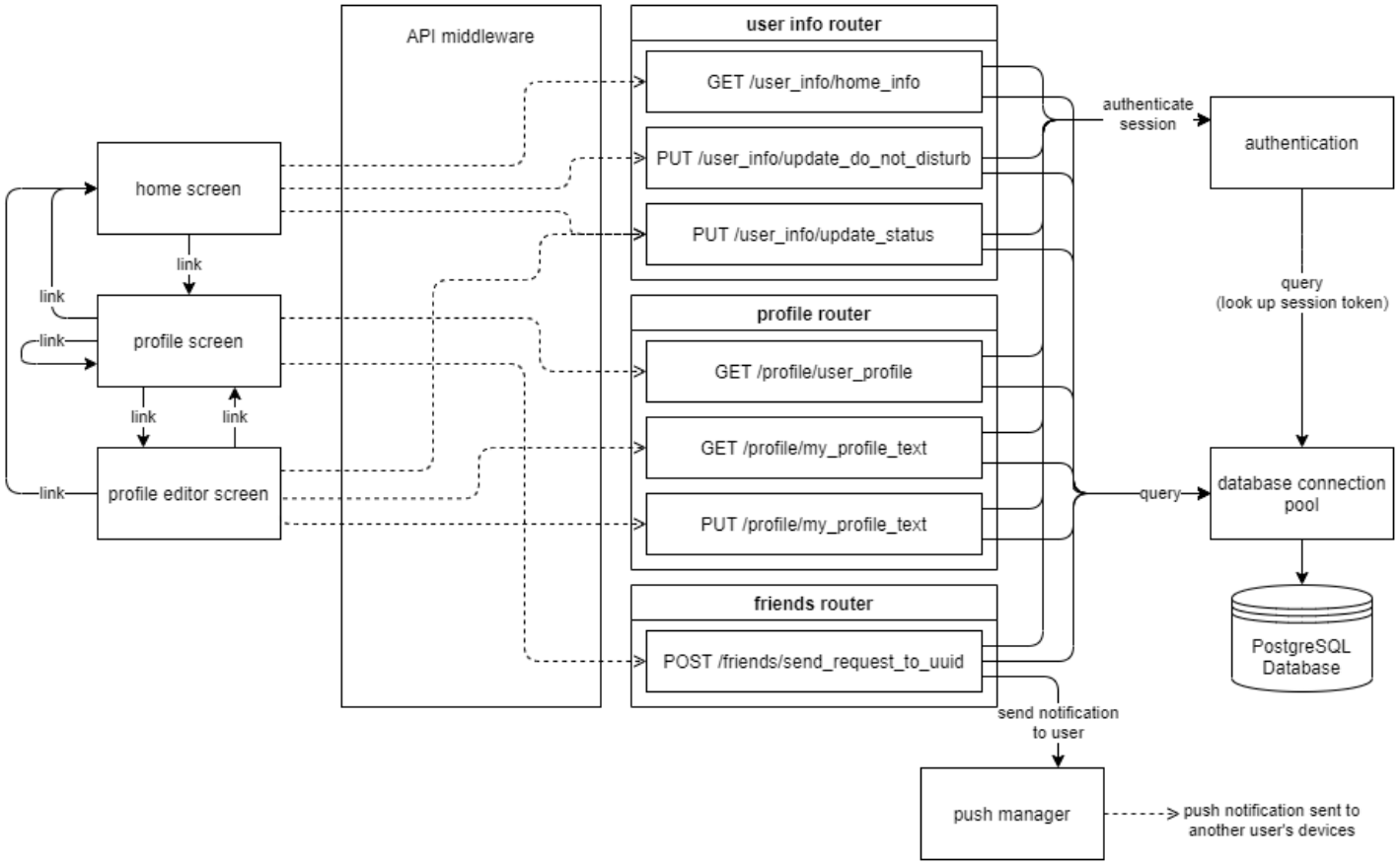



*Figure 6: Dynamic Component Design Diagram*

# User Interface Design

**Introduction**

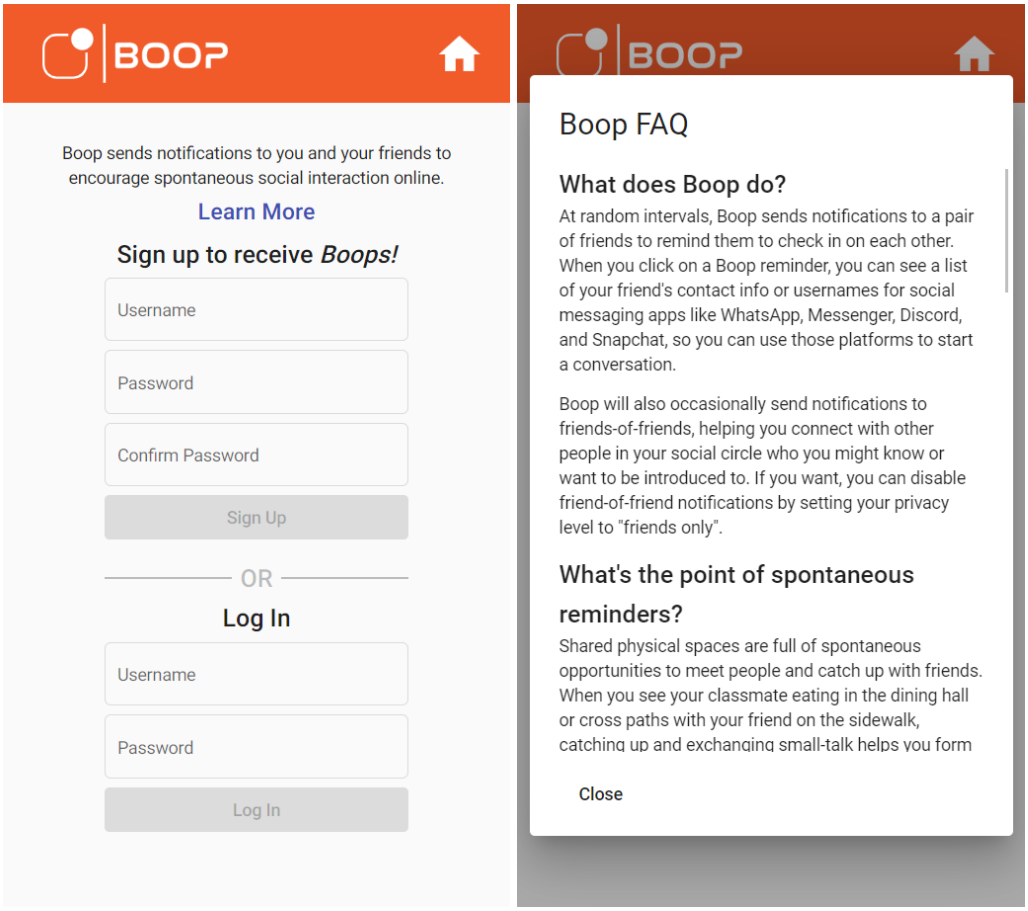
We created the user interface for Boop with the Angular framework and Google's Material Design component library. From the start, we designed the layout of our screens to work well at the size and aspect ratio of mobile devices. Our HTML is designed to work well on full-sized computer monitors as well, but Android devices are our primary target.

**Toolbar**

The toolbar is visible at the top of every Boop page. Clicking on either the Boop logo or the "home" icon takes the user to the home page if they are logged in, or to the landing page otherwise. We chose to use an orange color for the toolbar as well as on our favicon and other brand materials. In color psychology, orange is considered a "social color, making people open up and enhance their communication" [7]. This matches the friendly and playful feeling that we hope users will associate with Boop.

**Landing Page**

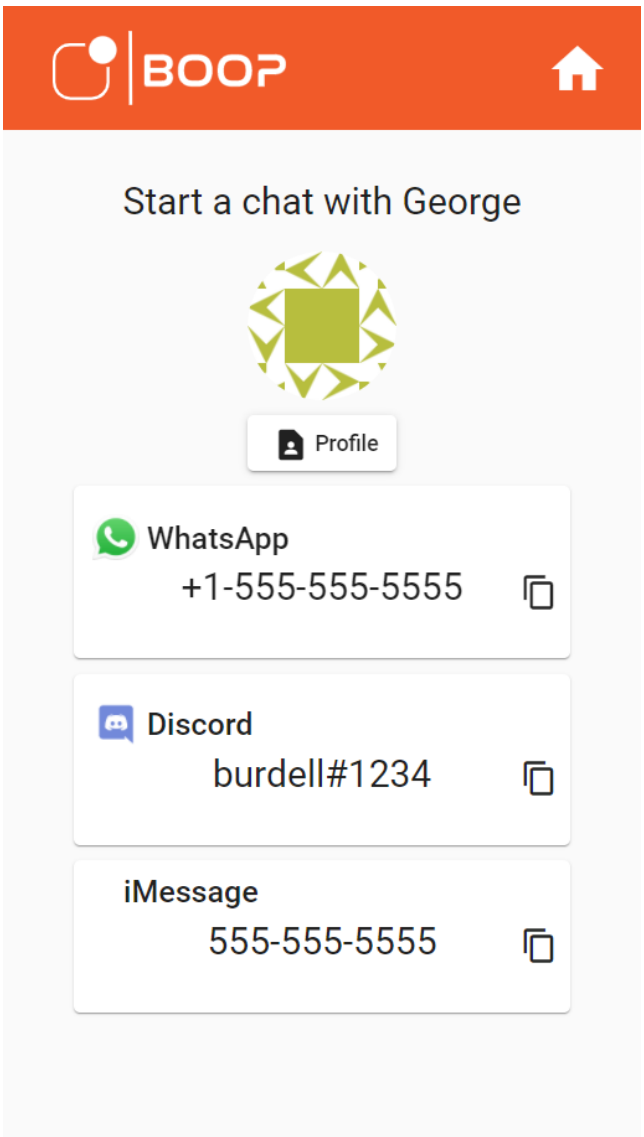
The landing page is the first thing that new visitors will see. Here, users can sign in or begin registering an account. The "Learn More" button opens a dialog box containing the Boop FAQ, which explains Boop's main features and how to use them.



**Chat Page**

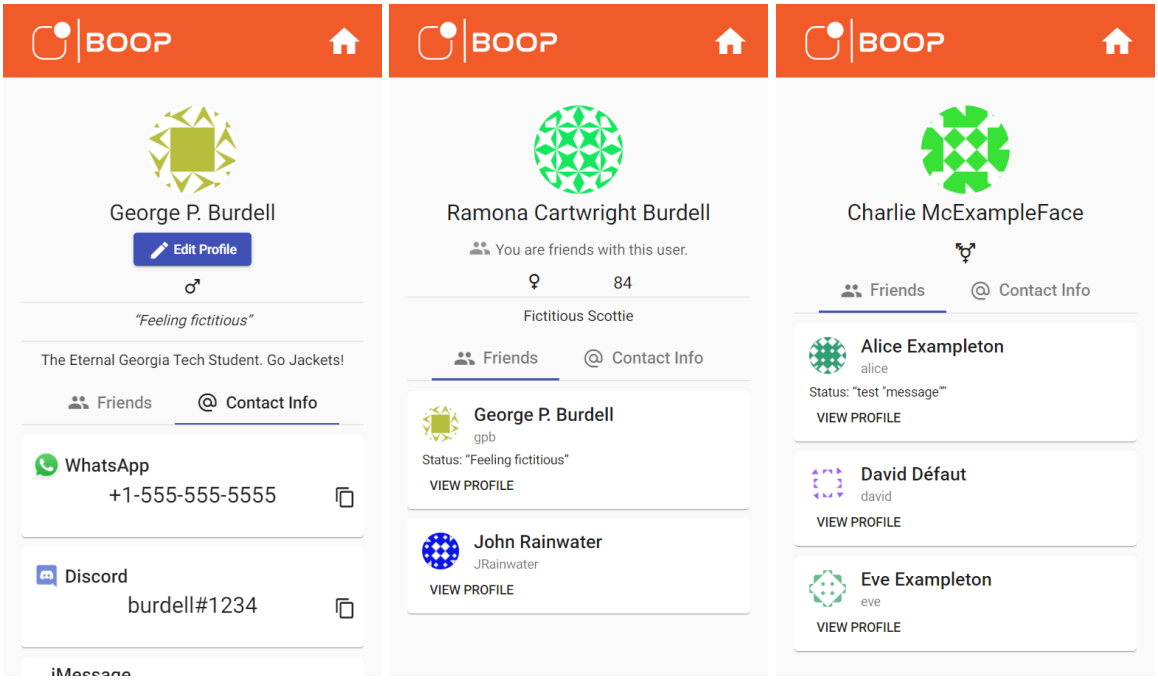
Boop reminder notifications prompt users to chat with a specific friend or friend-of-friend. The page shows that person's profile image and a link to their profile alongside a list of the contact methods that can be used to start a conversation with them.

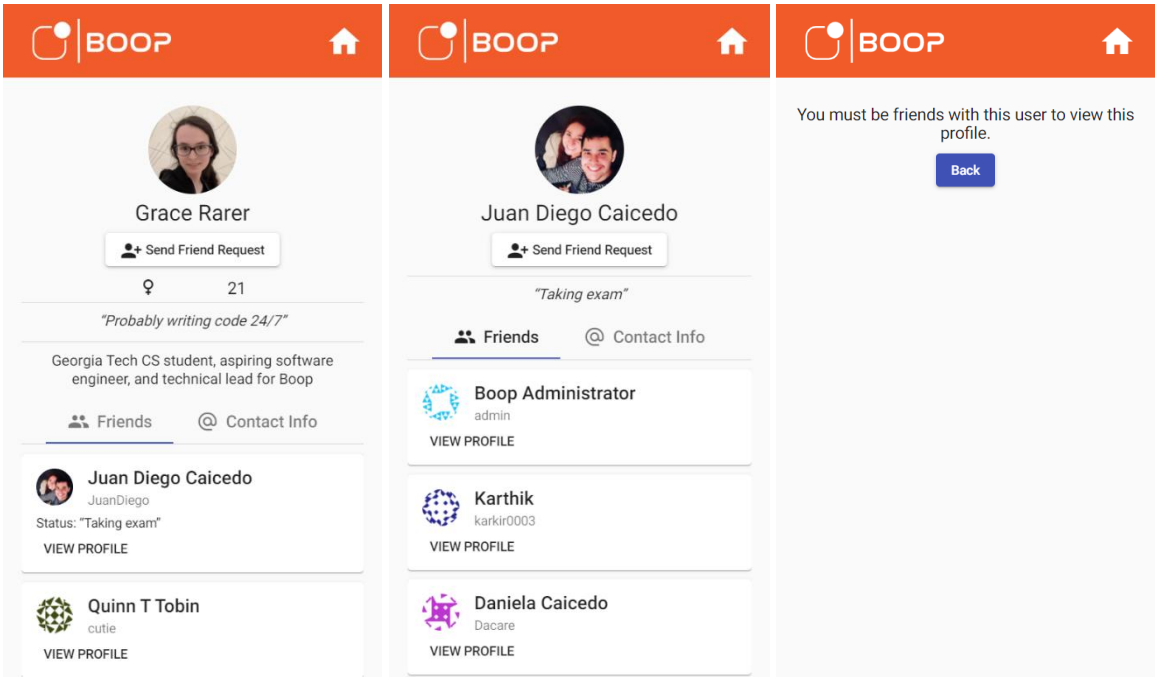
When we first created this page, we experimented with "deep-linking" into chat platforms' web clients so that clicking on a contact method would open a chat with that user on that platform. This worked in some cases, but not reliably. Each chat platform has a different format for these links, and many do not support deep links at all. Even for those that do support them, they may be unreliable if users provide their contact info in an inconsistent format, which is common if the platform uses phone numbers. Since it would not be possible to provide a consistent support for deep links across different chat platforms, we opted for a simpler design where clicking on a contact method copies that contact information to the user's clipboard.



**Profile**

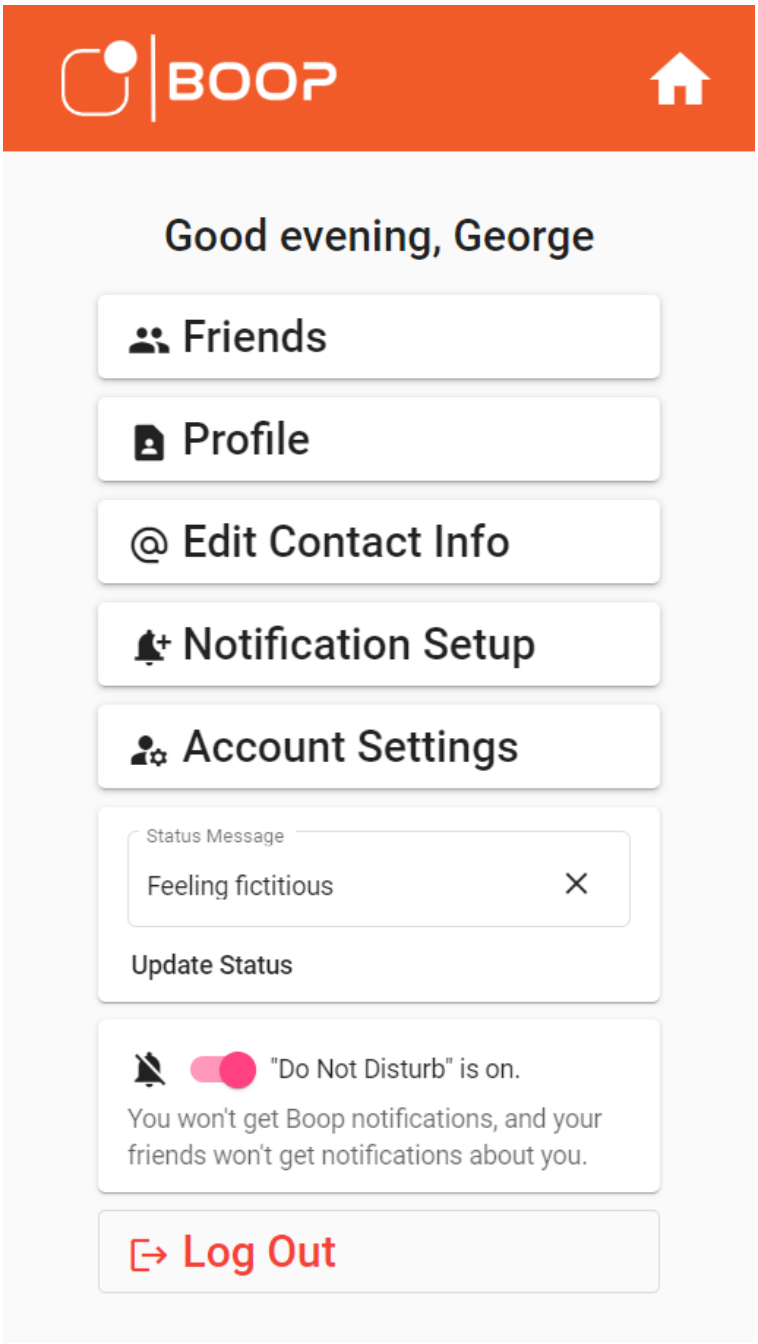
Profile pages are a way for users to introduce themselves and connect with potential friends in their social circles. Users can control which information is shown on their profile and who can view it. Profiles may include the profile image, gender, age, status message, and profile description. Users whose profile privacy is set to “everyone” can be viewed by users who are not logged in by navigating to boopboop.app/profile/<username>.



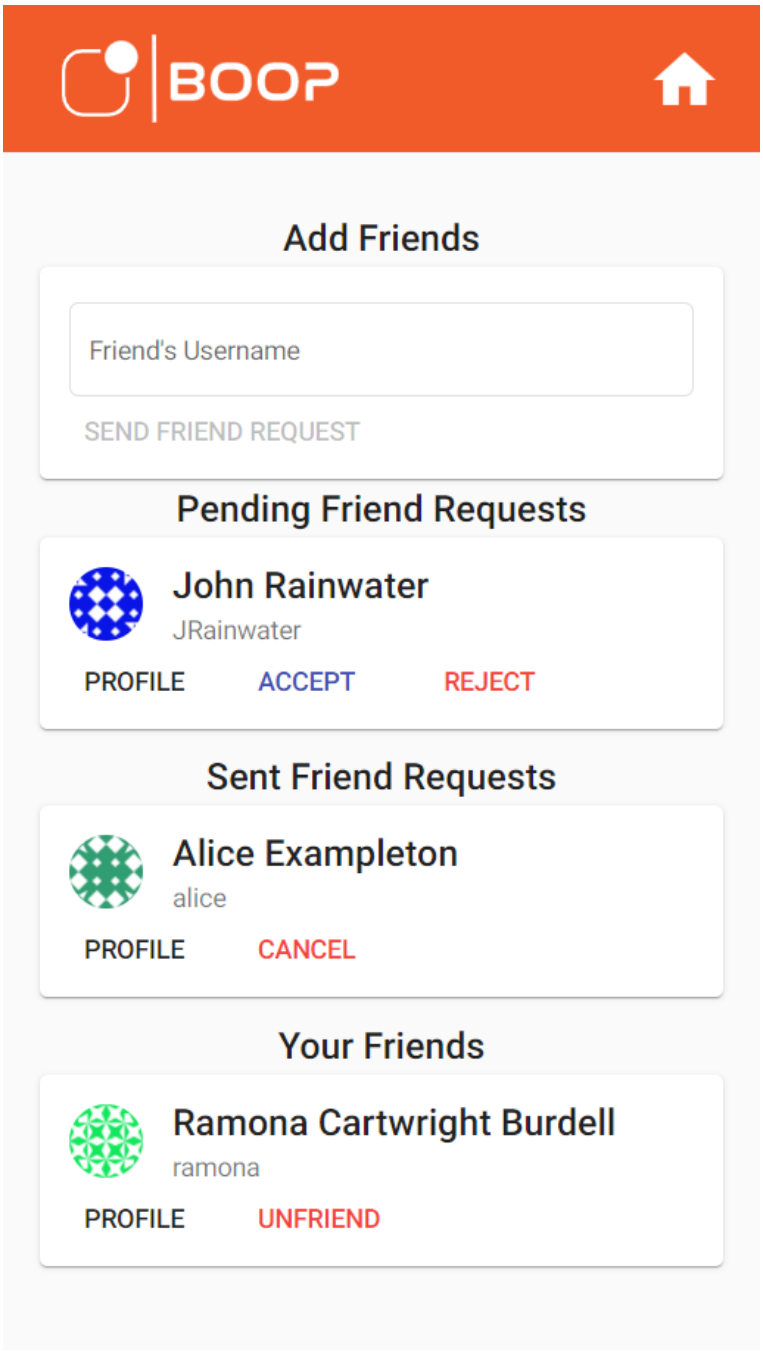
**Home Page**

The home page is the first thing that a logged-in user will see after opening Boop. The greeting at the top of the home page changes depending on the time of day in the user's time zone. In addition to linking to other screens, the home page contains the controls for the user's status message and "do not disturb" settings, since we want it to be easy for users to quickly change these.



**Friends**

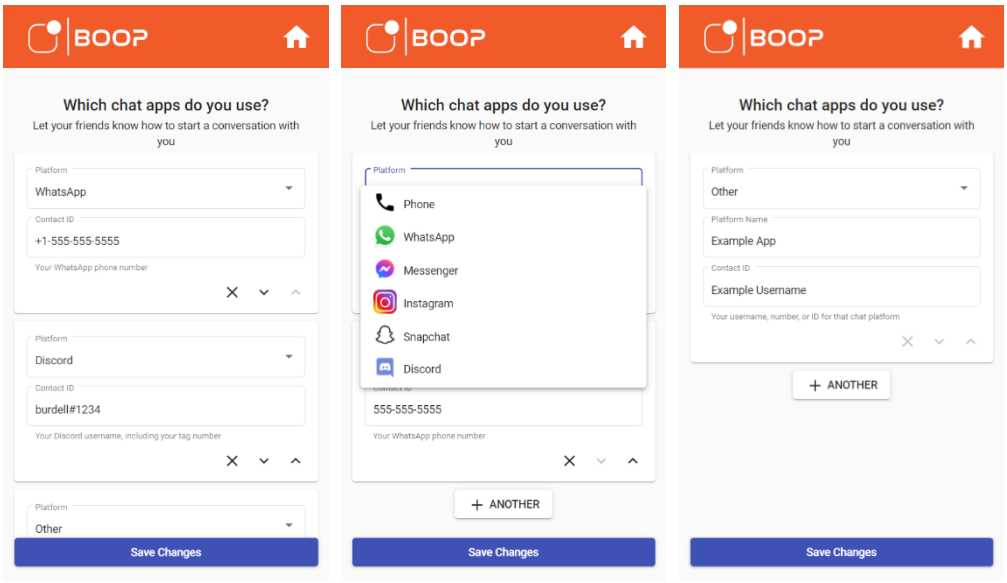
This page allows users to view their friends list as well as send and receive friend requests. Note that the example users in the screenshot below all have the default “identicon”-style profile images because they do not have profile images set with Gravatar.

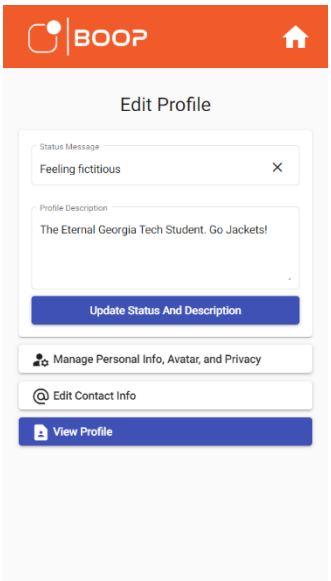


**“Edit Contact Info” and “Edit Profile”**

These pages allow users to change the information that is shown on their profile.

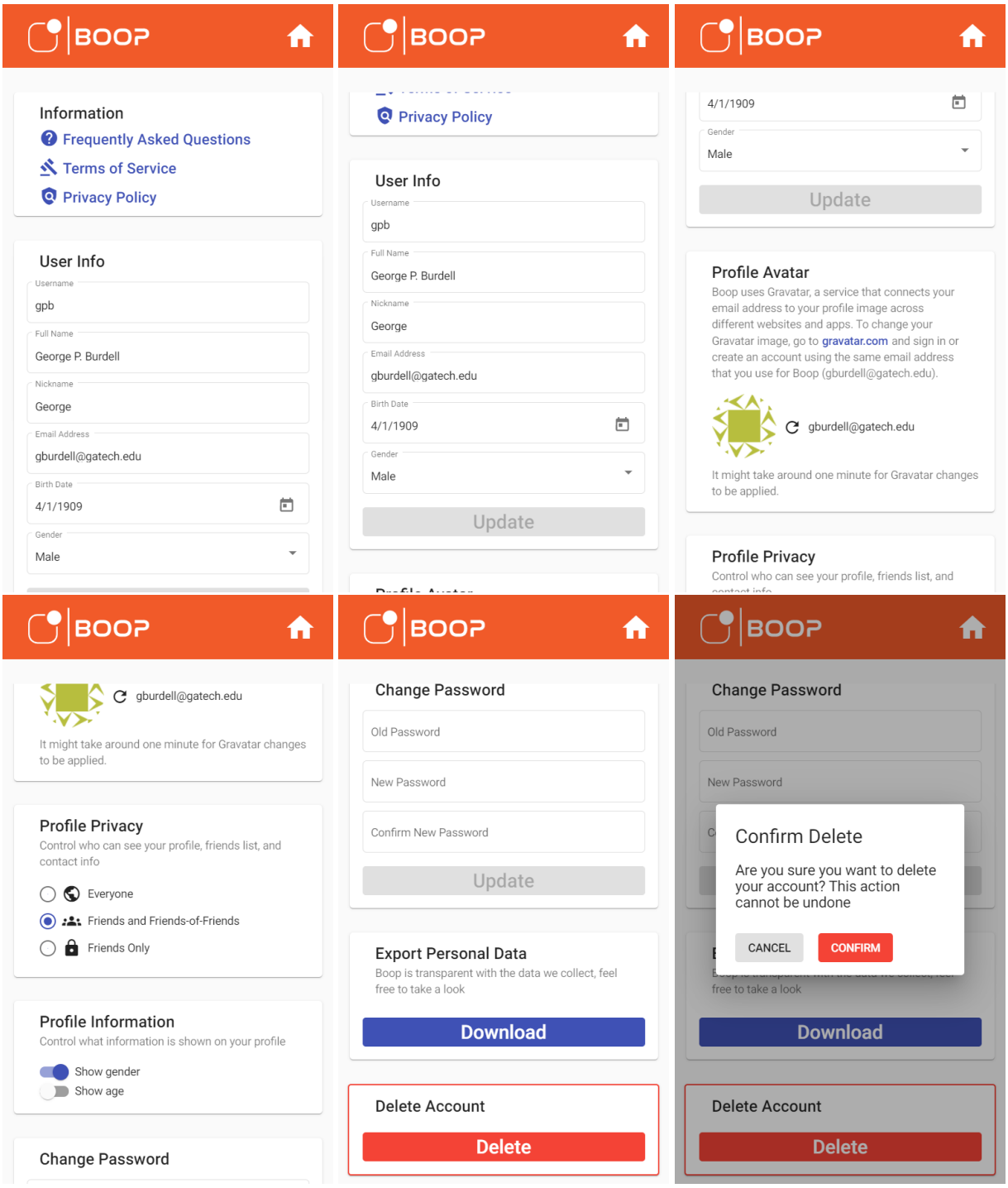
The dropdown list for chat apps shows the most common chat platforms, but users can also choose the “other” option and provide the name of any chat platform even if it is not in our list of options.



**Settings**

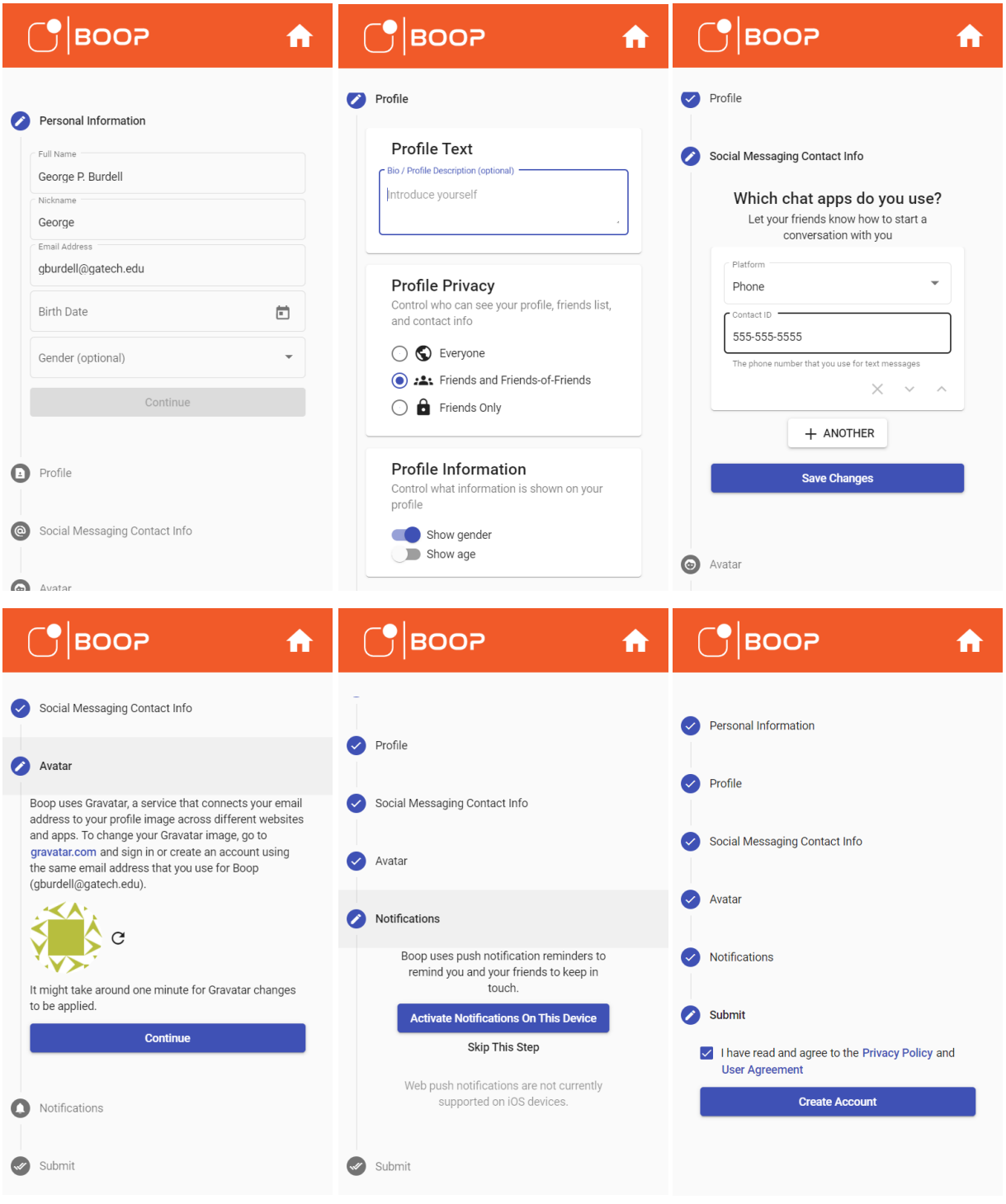
The settings page is arranged into “cards” that group various actions: viewing dialog boxes of information, updating account information, updating the Gravatar profile image, controlling privacy settings, changing passwords, exporting data to a prettified JSON file, and deleting the account.



**Registration**

Account registration is broken up into several steps. Users must complete one step before moving on to the next, but can go back to a previous step to change their information before submitting.

An earlier iteration of the registration process began with one page for collecting user information and then a sequence of “onboarding” pages for setting up contact details and notifications, but we found that users would get lost when trying to navigate or go back to the first page, so we decided to combine these steps into one page.



# References

[1] J. Medley, "Web Push Notifications: Timely, Relevant, and Precise," 12 February 2019. [Online]. Available: https://developers.google.com/web/fundamentals/push-notifications. [Accessed 2021].

[2] Automattic, "Image Requests," [Online]. Available: https://en.gravatar.com/site/implement/images/. [Accessed 2021].

[3] B. Wolford, "A Guide to GDPR data privacy requirements," [Online]. Available: https://gdpr.eu/data-privacy/. [Accessed 2021].

[4] Boop Team, "Boop Core Datatypes," 26 April 2021. [Online]. Available: https://github.com/GRarer/boop/tree/main/core/src/datatypes. [Accessed 2021].

[5] D. Arias, "Adding Salt to Hashing: A Better Way to Store Passwords," 25 February 2021. [Online]. Available: https://auth0.com/blog/adding-salt-to-hashing-a-better-way-to-store-passwords/. [Accessed 2021].

[6] C. Hale, "How To Safely Store A Password," 31 January 2010. [Online]. Available: https://codahale.com/how-to-safely-store-a-password/. [Accessed 2021].

[7] H. v. Braam, "Orange Color Psychology," 9 September 2020. [Online]. Available: https://www.colorpsychology.org/orange/. [Accessed 2021].