Table of Contents

Case 5 - Admin System

Background

Ascenda maintains a subsystem for facilitating administrative tasks and back-office operations. It is most frequently used by the employees of the company to assist customers, manage configurations and provide support to other departments.

The administrator system plays an integral role in increasing visibility of our data, accessing internal actions and helping the rewards platform run smoothly.

As the administrative system, we need to ensure that access to certain actions are protected against misuse and should be reserved to only higher level administrators. Therefore, the system needs to maintain a set of user roles, permissions and setup guards to prevent unauthorized use.

In this case study, you will be challenged to build an administrative system that will provide authentication and access control, expose data and actions from other backend systems and log critical user actions. By the end of this project, you will be exposed to access control, multi-app setup and logging practices.

Functional Requirements

Feature 1: Applications & Proxy design

Our system consists of multiple services, each responsible for a set of distinct business capabilities. In order to expose certain data and operations on an Admin User Interface (UI), the Admin system needs to connect and communicate with these services in a secure manner.

For the purpose of this project, set up three backend applications, each with their own databases:

Backend Application A: User storage

This application stores information of users. User information includes: name, email address, enrollment date and role.

This application exposes API endpoints for:

- Listing users
- Enrolling new users
- Updating and deleting user information
    - Deleting own user information is not permitted

Backend Application B: Points ledger

This application is a simplified version of our loyalty engine, where each user's points balances are calculated and kept. The balances are stored in points accounts. A user can have more than one points accounts at once.

The application exposes API endpoints for:

- Querying points balance of a user
- Adjusting points balance of a user

Points account information should be consistent in a sense that the points balance adjustment action is reliable. A good rule of thumb here would be to implement the transaction following A.C.I.D. rules. However, for the purposes of this case study, the transaction itself does not need to be very performant, so longer waiting times are acceptable.

Backend Application C: Admin Proxy App

This application is responsible for relaying requests from the Admin Panel UI and communicating with the other backend applications. It acts as a gateway between the client and the rest of the system.

When an Admin Proxy App's API endpoint is called, it:

1. Validates that user is authenticated (see more details in Feature 2)
2. Validates that the admin has the required access (see more details in Feature 3)
3. Forwards the request to the relevant backend application via API call
4. Records audit logs on certain more critical actions (see more details in Feature 4)
5. Returns the response from the backend application

In addition to these, set up an application that will enable surfacing the relevant information and actions:

Frontend application: Admin Panel (UI)

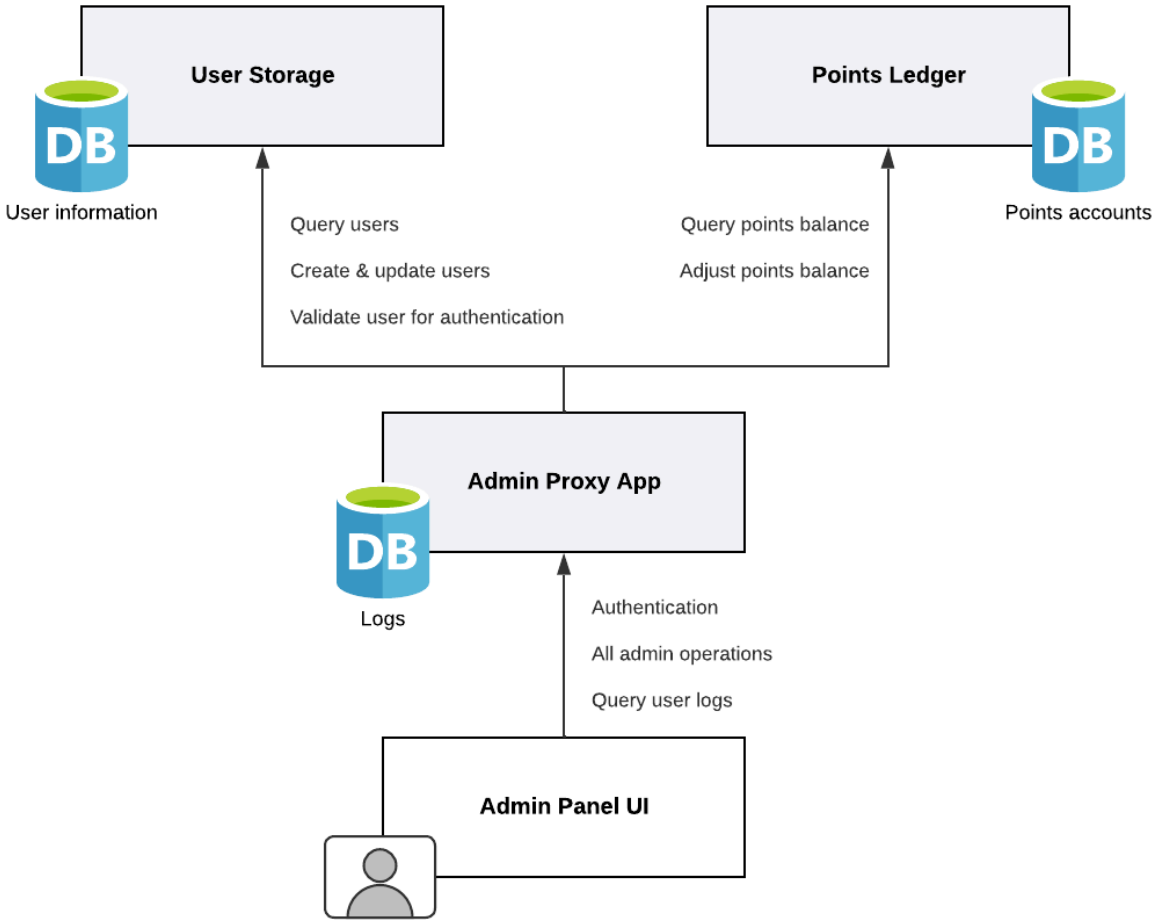
A simple frontend application to expose back-office operations and information. The layout and the design doesn't need to be fancy, and the focus is on functionality. Therefore, there aren't any specific usability or user experience requirements for the frontend. For the purposes of the project, it should include these pages and features:

- Users listing page - queried from **User Storage & Points ledger**
    - Display details: first name, last name, email address, role, points balance
- Form 1: Enroll new user - via **User Storage**
- Form 2: Update user details & role - via **User Storage**
- Form 3: Adjust points of a user - via **Points ledger**
- User logs listing page - queried from **Admin Proxy**

In summary, your solution should include:

- Setup of multiple backend applications
- Communication channels between their public endpoints
- An application that acts as a gateway between the frontend app and the backend services

Below is a diagram showcasing the overall architecture of the Admin system:



Feature 2: Authentication

Feature 2.1: User Enrollment

Before the administrators can access our admin system, they need to be enrolled with us. The first administrator user's information can be seeded into the database as we set up the application.

Following that, new admins are enrolled via an API call to the User Storage application. This can be done by administrators with certain access after logging in (see more in Feature 3).

To enroll a new user, set the following attributes:

- Name (required)
- Email address (required)
- Role (optional)

If the user is enrolled without a role specified, they would be treated as a customer (user without administrator access) in the system.

Besides this, use these files provided as seed data:

- users.csv: seed to the User Storage application's database
- points_accounts.csv: seed to the Points Ledger application's database

This is a list of non-admin users (customers) that will help act as a baseline customer dataset and help differentiate from subsequently setup admins. Refer to **Appendix 2** to gain a better understanding of the user attributes.

Feature 2.2: Admin Authentication

The next step is to enable enrolled admins to log in to the system. To keep things simple, you may use a third-party social identity provider (e.g. Google, Facebook) to manage the login flow.

By default, all of the pages in the Admin Panel UI are restricted, and only logged-in users can view them. After pressing the login button, the user is taken to a screen where they can log in. Upon successful login, the user should be redirected to the Admin Panel UI upon and see the protected pages.

An important restriction is that only previously enrolled administrators should be able to login. Therefore, implement a validation step in your login flow that will reject login attempts coming from non-admins. Refer to **Appendix 3** for a simple high-level sequence diagram of the login flow.

In summary, your solution should be able to:

- Enroll new admins to the system
- Authenticate admin users
- Restrict pages of the Admin Panel UI to be only accessible for logged-in users

Feature 3: Access Control

Another main feature of the admin system is access control management. We need to ensure that certain information and critical features are only accessible to higher level admins.

The access level of an admin is defined by their role attribute. Every admin with the same role has the exact same access level. The access to restricted features should be given to the admin after they successfully log in. A given role normally has multiple permissions, which provide a capability related to a given resource. For example, one permission could provide read access to the User Storage's users resource, while another could give write access to the Points Ledger's points accounts resource.

The goal here is to create a solution which allows roles and their permissions to be dynamically created and updated, so it will be scalable as our system grows. Roles and permissions should be manageable and changeable on-the-fly without redeploying the services.

For reference, set these roles up in your system as default roles:

Role name	User Storage	Points Ledger	Logs
Owner	Read, Create, Update, Delete	Read, Update	Read
Manager	Read, Create, Update	Read, Update	Read
Engineer	Read	Read	Read
Product Manager	Read - only see non-admins	Read	N/A

The Admin UI should display only features and information accessible to the logged-in admin's given role. For example, an admin with the Engineer role should not see the "Update user" button or access the page, since they don't have authority to use that feature.

Note that you should ensure the service validates access on the backend layer too, so a sophisticated user can't access a resource by sending a request directly. When a user tries to access a feature / resource they don't have access to, the service should return an Unauthorized error message.

In summary, your solution should be able to:

- Manage several roles
- Support multiple permissions per role
- Deny users the access for resources/actions that they do not have the permission to

Feature 4: Logging

Recording and making past user actions visible is a cornerstone feature of the admin system. To keep actions of our users accountable, set up a logging service which is easily configurable for any operation handled by the Admin Proxy app.

Configure logging on these actions, while keeping it easily extensible for other actions in the future:

- A new user is enrolled
- User information is updated
- A user is deleted
- A user's points are updated

A given log should include at least this infromation:

- Relevant information to the action, for example user's attributes, points amount that was added / deducted from user's points balance etc.
- Timestamp of the action
- User agent (browser) information
- The admin's approximate location information based on their IP address

An important addition is that our logs should not contain PII, such as email addresses and user's names. Configure the logging system in such a way that this information is filtered before it's recorded in the database.

Accuracy of the logs is also important, as a few missing logs could be impactful for the business. Therefore, minimize the amount of missing logs as much as possible.

Besides this, create a page on the Admin Panel UI where the user logs can be retrieved and viewed by higher level admins (refer to the table in Feature 3) in an easily digestible way. Since the total amount of logs can be large, you should ensure that the logs are paginated before they are displayed on the UI.

In summary, your solution should be able to:

- Record critical actions in logs
- Filter personal information from these logs
- Expose the logs to be viewed on the Admin Panel UI

Bonus:

- Implement a search functionality on the logs, so the administrators can filter the results by: description of the action, timestamp, and relevant user ID.
- Make the log retention period customizable, so old logs are not archived in the system forever.

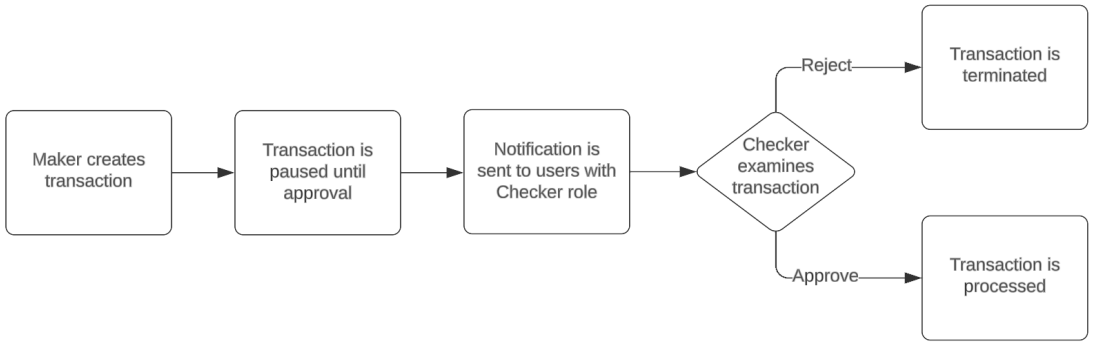
Feature 5: Maker-checker (Bonus)

In certain cases, we would like to expose more critical features to lower-level administrators, but also ensure safety and security of the system. Maker-checker is a workflow where at least two individuals need to agree on a transaction before it's actually processed. We would like to see the team explore this concept to enhance the Access Control feature and make it even more flexible for future use. This would be preferably a feature of the Admin Proxy App.

Generally, the Maker-checker process has three steps:

1. **Maker** makes a transaction on the Admin Panel UI. A transaction refers to any action that is configured in the Maker-checker system, e.g. points adjustment.
2. **Checker** receives notification via email and approves the transaction. You can implement the approval step in Admin Panel UI or directly via the email.
3. **Transaction** is processed on the backend.

Below is a more detailed flowchart of the process:



In summary, your solution should be able to configure:

- Which Admin Panel action should be part of the Maker-checker process
- Which role is the Maker
- Which role is the Checker

For each step of the Maker-checker process, logs should be stored in the system.
For the purpose of this task, you can work with this default configuration:

Action	Maker role	Checker role(s)
Update points balance in Points Ledger	Product Manager	Owner
Update user details in Users Storage	Engineer	Manager, Owner

Non-functional Requirements

Resilience & Disaster recovery

Even datacenters can go down. Our system should be resilient against this. Your deployment should at least be distributed over 2 availability zones (or the equivalent).

Should the system go down, our RTO is 1 hour with minimal loss of data. Based on this, propose a recovery process for how you would restore the system and the steps taken to achieving this. We would also like to see the team keep the RPO less than a day and make it measurable in hours, considering the cost and type of information stored.

Scalability

We need to be able to support high volumes of authentication API calls. The system should have the ability to scale to support elastic loads, ranging from 5 requests per second typically to over 100 requests per second at peak load. The most common queries in order would be fetching user information, fetching points balance, adjusting points balance and updating user information. Peak hours are normal working hours between 08:00-18:00, and even during the highest traffic hours, the peak loading time should remain below 2-3 seconds.

You should demonstrate the various steps taken to optimize both the application & how your services used can help scale our traffic.

Growth trend

Based on previous user base growth, it's expected that we would set up partnerships with 2-3 larger banks each year, each bringing at most around 100,000 new users and at most around 200 admins to the platform.

Typically an admin would visit the Admin UI every day and support 5-10 customers a day. Based on this, propose the plan for supporting the system load growth. You can also deduce the storage requirements of the system. There are no specific requirements whether relational or non-relational databases should be used.

Data security

Personal information

For privacy purposes, data for each bank should be separated (logically or otherwise). Any PII information should be encrypted or filtered (in logs).

GDPR (General Data Protection Regulation) allows for an end user to request that their information be destroyed. Similarly, partners can request for PII information to be destroyed/ made inaccessible.

You should have a plan in place for effective removal of user PII as needed. A straightforward strategy is to implement a “Delete Account” feature which allows users to delete their information completely from your database.

Systems security

As you are building a service that is utilized by banks globally, we would like to see the techniques employed to secure your application.

Defense in depth is key: In the event your one part of your deployment is compromised, what are the alerts & downstream defenses that can be employed to guard admin & customer data?

A consideration could be limiting access only on a private network. While backend systems need to be only privately accessible, for the purposes of this case study, you may assume that the Admin Panel UI is exposed on the public internet.

Bonus:

- Set up other tools to support observability, such as monitoring system load, response timings and issue detection.

Ease of Maintenance

We deploy daily. No downtime is allowed during deployments. Health-checks should be in place to validate new deployments.

Bonus:

- Set up tool(s) to support observability, such as monitoring system load, response timings and issue detection.
- Prepare documentation to showcase how the administrators can be enrolled to the system and how they can use the Admin Panel with some examples.

Budgeting

There aren’t any specific limitations set by the Ascenda requirements, but lowering the cost can be considered as an improvement to the system.

Appendix 1: Glossary

Acronym	Definition
API	Application Programming Interface
UI	User Interface
PII	Personal Identifiable Information
Term	**Definition**
API gateway	The primary function of the API gateway is to provide a single, consistent entry point for multiple APIs, regardless of how they are implemented or deployed at the backend.
User	Identifies a person. Represented as a row in the Users Storage application's database.
Administrator	A user who has their "role" attribute set. The role's value should be a valid role within the system.
Customer	A user who has no value in their "role" attribute.
Logging service	An external service that accepts and records audit logs of critical actions of the system
Maker	The person making a transaction
Checker	The person receiving the transaction request for approval

Appendix 2: Attributes

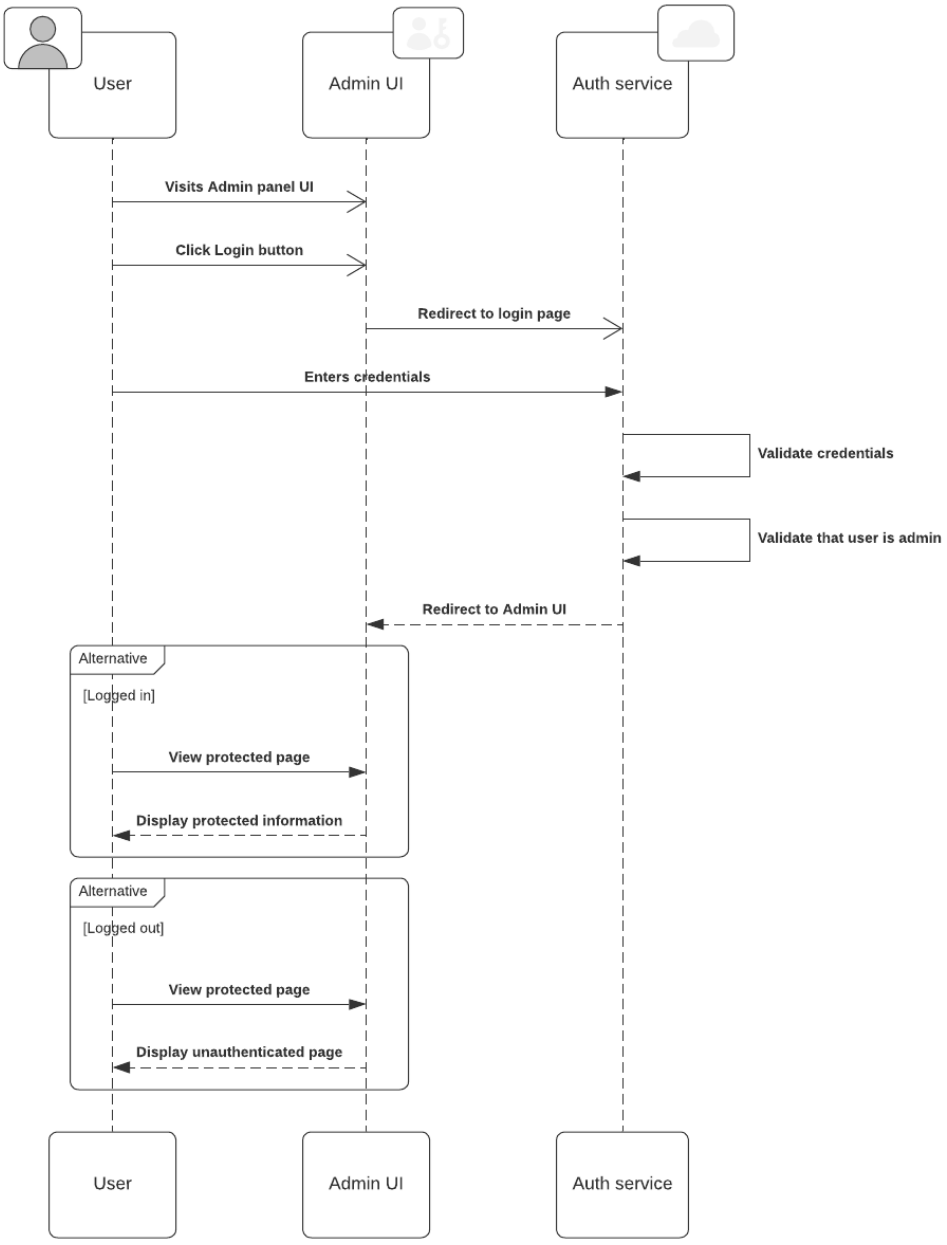
User Storage attributes:

Attribute	Description
id	Unique user ID
first_name	User's first name
last_name	User's last name
email	User's email. Does not need to be unique for each user.
role	Optional attribute that determines the user's type of access on Ascenda's platform. If they have a role, they are an admin. Otherwise, they can still log in but have no access to the Admin Panel features. Default roles: Admin, Manager, Engineer, Product Manager

Points Ledger attributes:

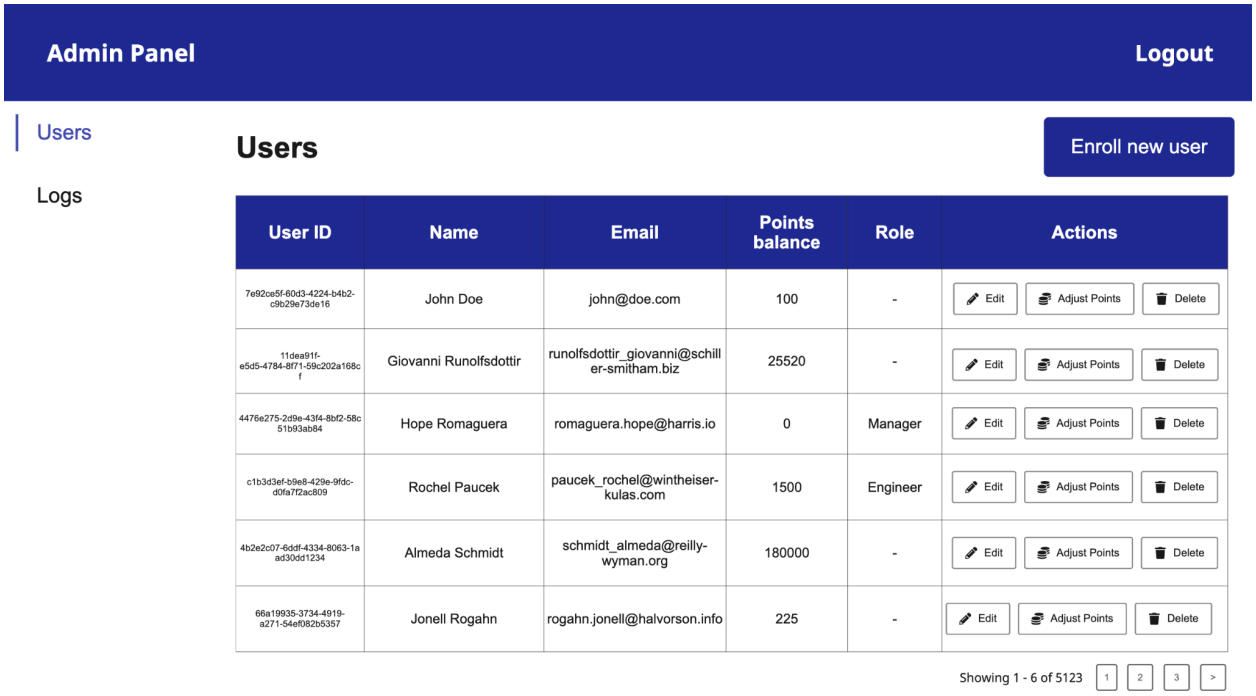
Attribute	Description
id	Unique points account ID
user_id	The owner user's ID. This is a foreign key to the ID in the User attributes.
balance	Points account's current balance

Appendix 3: Login flow sequence diagram

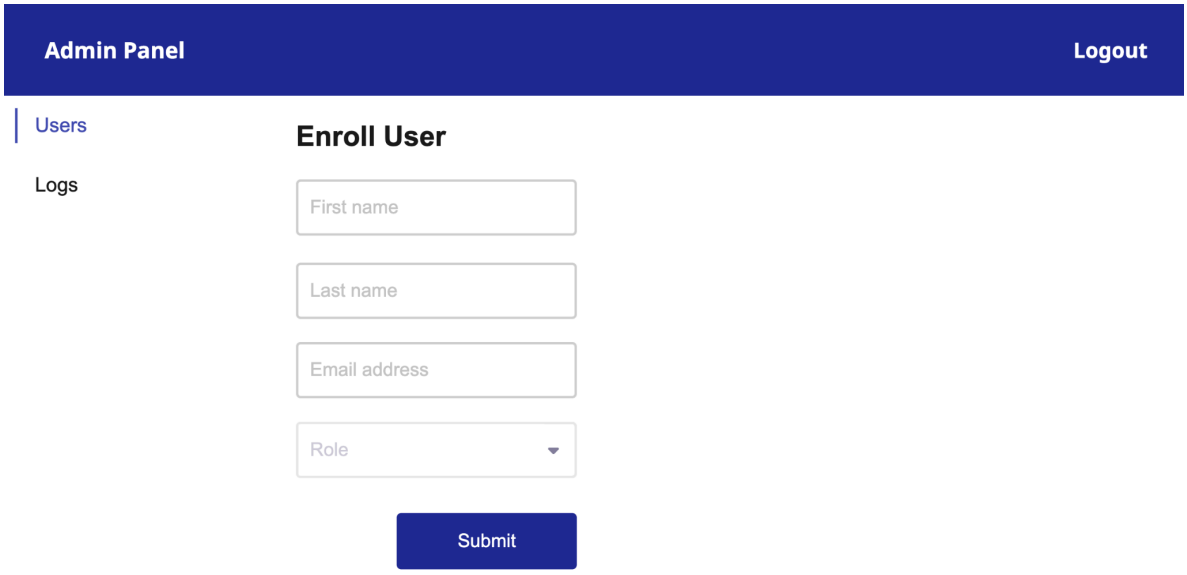


Appendix 4: Admin Panel UI Mockups

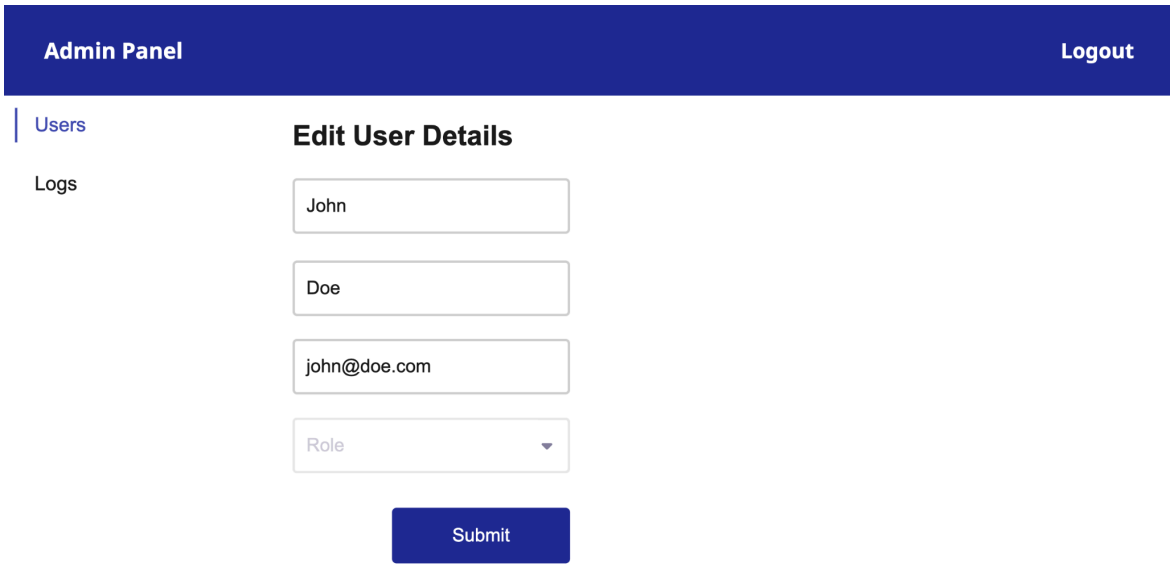
Use these mockups for some inspiration in your projects. However, feel free to create a completely different design for your solution.



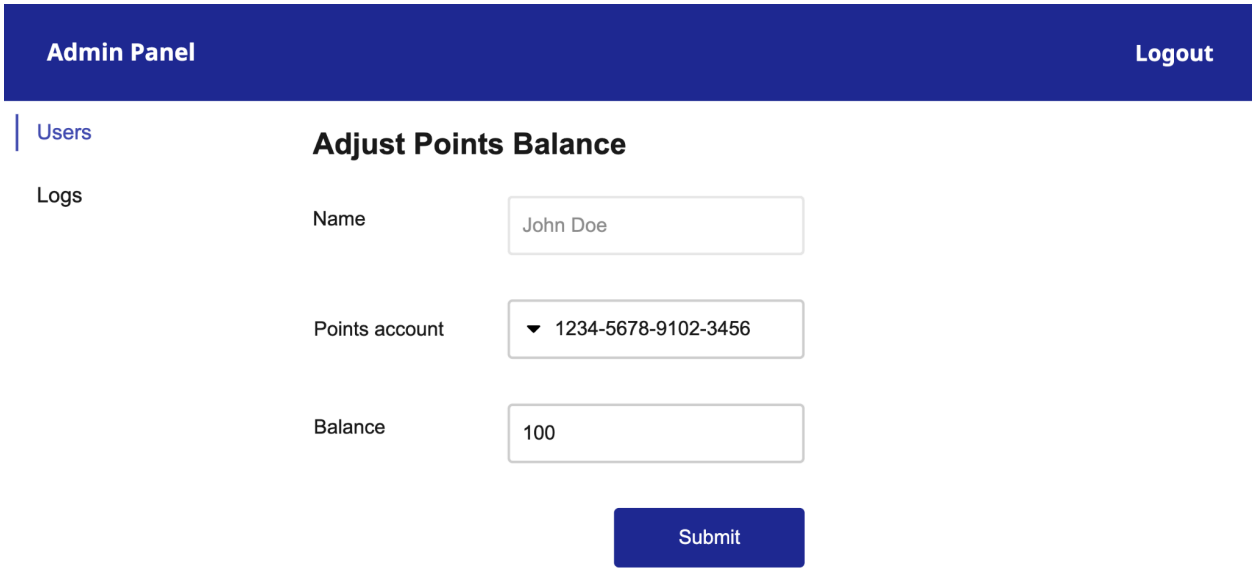
Admin Panel UI - Users view mockup. Note: “Points balance” column here can display the sum of all points balances from points accounts of a given user.



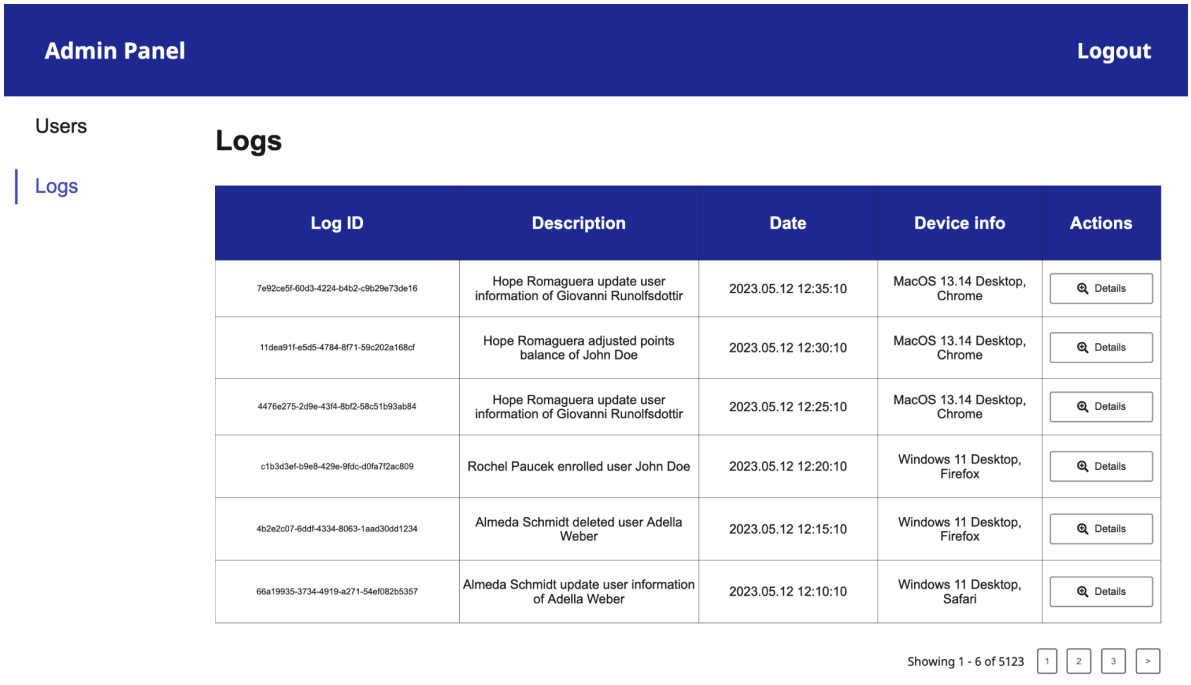
Admin Panel UI - Enroll user form mockup



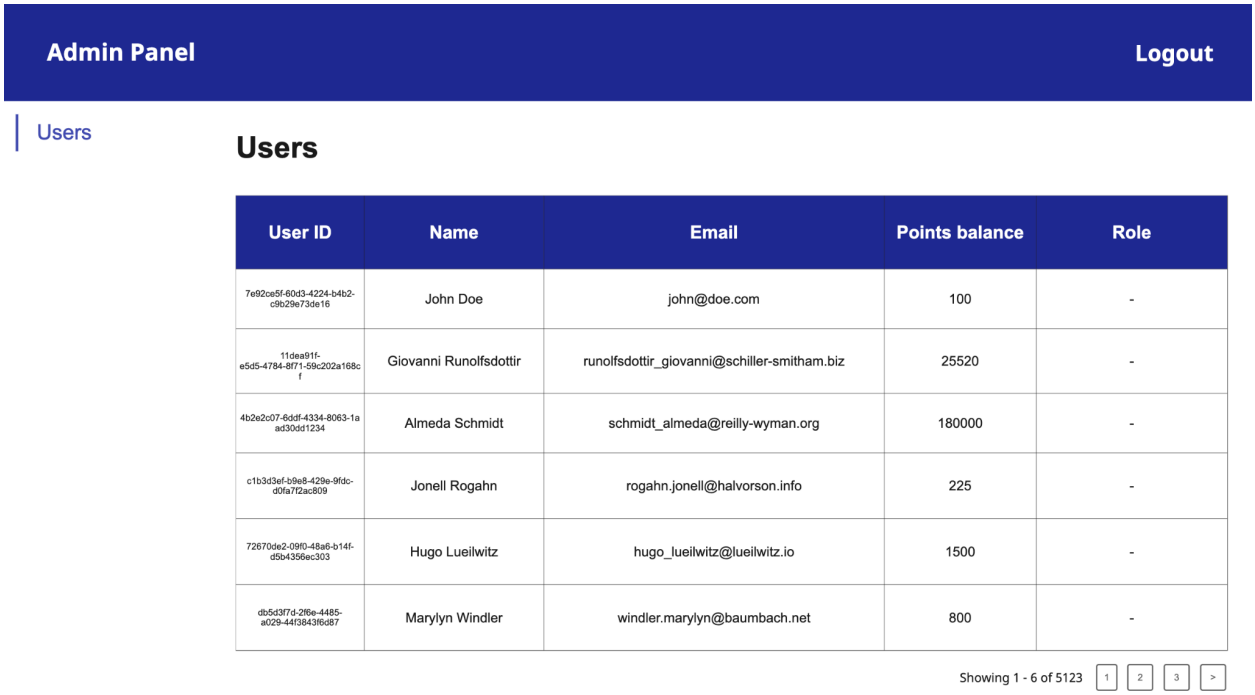
Admin Panel UI - Edit User Details form mockup



Admin Panel UI - Adjust Points Balance form mockup



Admin Panel UI - Logs view mockup



Admin Panel UI - Users view mockup as “Product Manager” role (only read-only access to the list of users).