Purpose

This is an internal design/bootstrap document that has a few purposes

- Explain the feature to a developer that has near-zero knowledge of specific tools and technologies used to create this feature
- Settle on a high-level design for the feature in Affordable
- Provide a framework for said developer to begin work on the feature

This document in particular is focused on implementing an administrative user account in Affordable.

Motivation

In any software system, there are often tasks that should only be performed by internal users/developers that have internal access to the system. If these tasks are one-off, then it usually makes sense for those users to manually modify the database. When these tasks become more frequent, it's often not worth the risk or time to manually modify the database. In these cases, having an administrative account simplifies the process.

Implementation Approaches

There are a few different ways to accomplish this, and none of them are necessarily incorrect. I will be outlining one straightforward way to do this.

One Admin User

One admin user could be created when the database is created, or when the server is launched (I think I would prefer defining this in the server over in SQL). One way this could be done is modifying DatabaseConnection.ts (which is run whenever a database connection is initialized. We could add a createAdminUser function, and run it after the database connection is initialized:

```
class DatabaseConnection {
	public static async initializeDatabaseConnection(): Promise<Connection> {
		connection = await createConnection({...}); // This is the same
		await DatabaseConnection.createAdminUser(connection);
		return connection;
	}

	public static async createAdminUser(connection: Connection): Promise<void> {
		await connection.getRepository(AuthenticationInformationDBO)
		.save({
			id: ADMIN_ID,
			username: ADMIN_USERNAME,
			password: ADMIN_PASSWORD, // this should be hashed before saving
			email: "admin@affordhealth.org"
```

```
            });
        }
}
```

ADMIN_USERNAME and ADMIN_PASSWORD can be defined as environment variables (in the .env file). These are accessed with the dotenv library and using process.env.<varname> (as seen elsewhere in the DatabaseConnection class). ADMIN_ID can be saved as a constant (I think `1` works fine).

Checking Permissions

We should have an Authorization service class (either created for this task or created for the AuthN/AuthZ project in progress). This class simply needs to have a method isAdmin(userId: number) that returns true or false if the ID matches 1. Then, we simply use this method in the business logic of other methods to determine if a user is permitted to do something (putting this logic in each method is not necessary to complete this task, it's straightforward and we can handle it later once we have reduced the feature set).