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

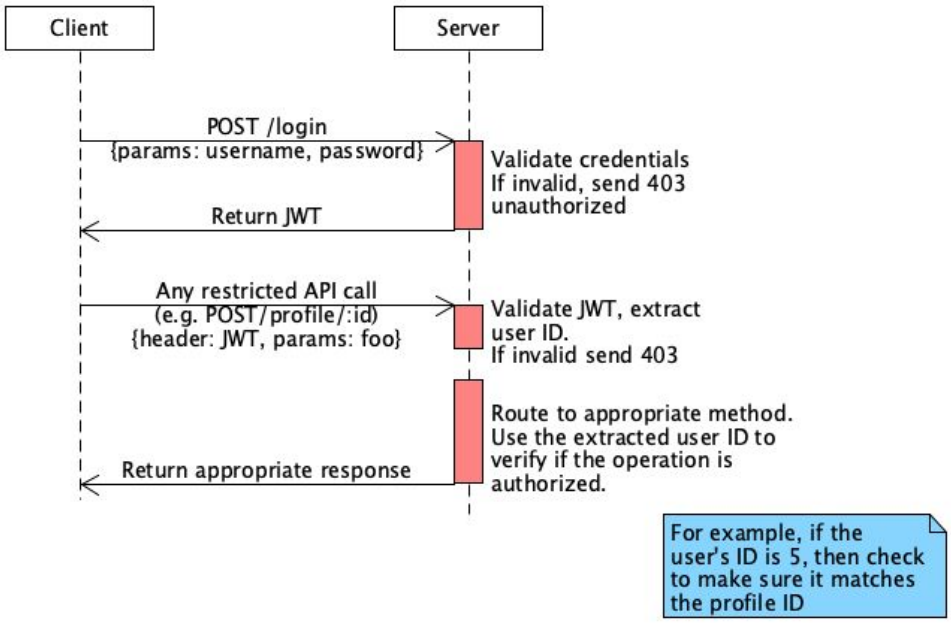
This document in particular is focused on implementing mechanisms for authentication and authorization in Affordable.

Authentication

Here we'll talk about how to authenticate a user at login, and how to persist that authentication.

Here's a decent guide I found to a broad overview of how this should work. It includes some implementation details that can be helpful references, but I would prefer we spend more time hashing out our own design rather than copy-pasting
https://stackabuse.com/authentication-and-authorization-with-jwts-in-express-js/

The sequence diagram below captures the basic flow how a client would obtain and use a JWT to authenticate in Affordable



We can split the work into the two different client-server interactions: the Login flow (which results in a token being generated and returned), and the token-based authentication flow (where a user utilizes the token to complete any API call that would require authentication)

Implementation

JSON Web Token Approach

Using JWTs seems like a good option because in addition to complying with an internet standard, we can capture identity and a secret in one token. We can use the json web token package on (npm/GitHub)

One of the benefits of using JWTs is that we do not need to store the token server-side. The token is signed using an HMAC algorithm, and the token can be verified so long as the signing key is kept secret. (The main drawback of this approach being that JWTs are not revokable). Here is a good summary of advantages and drawbacks to various approaches.

Login Flow

To support generating a session token when we log in we must

- Generate JWTs
    - Libraries can be used to do this (see jsonwebtoken above)
    - We must decide what information goes into the JWTs
- Redesign the API to support returning a JWT

The login method request body should be straight forward; the client provides a username and password

```
class LoginRequest {
	username: string;
	password: string;
}
```

In the response, we return UserInfo so that the client can store/display information about the user. We should now additionally pass the JWT that the client will use to authenticate.

```
class LoginResponse {
	userInfo: UserInfo;
	token: string | JWT;
}
```

In the method AuthenticationService.validateLogin (and likely also the registerUser method), we will want to generate and pass the JWT

```
class AuthenticationService {
	public async validateLogin(request: LoginRequest): Promise<LoginResponse> {
		// Code for the following two tasks already exists
```

```
		// Validate username, password, (2FA if necessary, but we will ignore
	this flow for now)
		// Create the UserInfo object

		const loginRequest = new LoginRequest();
		loginRequest.userInfo = retrievedUserInfo;
		// We will delegate to a separate method which uses a library
		loginRequest.token = generateJsonWebToken(...);
		return userInfo;
	}
}
```

Token-Based Authentication

The client will always pass the token in the header of the HTTP request. (The protocol dictates that this is done in form { Authorization: Bearer <token> }. For this work, we will assume that this token is always passed (configuring the React frontend/Affordable JS client to pass this is a separate issue).

We need a way to validate the token and extract a user ID from the request before a request hits the controller/service layers. I think the most straightforward way to do this would be to write an Express middleware function.

This function would look something like (pseudocode/JS)

```
function parseJwt(req, res, next) {
	let token = req.headers.get('Authorization');
	// Parse jwt from this token text, this can probably be done with a library
	// Parse user ID from the token as well
	const userId = parsedToken.iid; // not sure what field
	let isAuthenticated = await authenticationService.validateToken(jwt);

	if (isAuthenticated) {
		res.locals.userId = userId; // see here
	}

	next();
}
```

We can use the middleware by doing

App.js

```
app.use(parseJwt)
```

In the root folder of the app, we can just do `express.use(parseJwt)`, and then this middleware would be called every time an endpoint is called.

Say we get an HTTP request

```
GET url.com/bar/
```

In each controller that calls a service, we can do

```
class FooRouter {
	router.get("bar/", FooController.barControllerMethod);
}

class FooController {
	public async barControllerMethod(req, res, next) {
		try {
			let response = await barService.getBarMethod(res.local.userId,
...);
			res.status.200.json(applications);
		} catch (error) {
			next(error);
		}
	}
}
```

As a logical consequence of this, all service methods that should require authentication will have their method signature changed. For example, `ProfileService.createProfile(profile)` will be changed to `ProfileService.createProfile(userId, profile)`.

Additional Considerations

Users with unverified email addresses

What should they be allowed to do? This may warrant a discussion with sponsors.

Supplemental

While I was working on Synapse.org, one of the Principal Engineers was tasked with revising password policy to be compliant with HITRUST/HIPPA guidelines. We should consider implementing this password policy. This document consolidates his findings and conclusions: https://sagebionetworks.jira.com/wiki/spaces/PLFM/pages/717979671/Synapse+Password+Policy