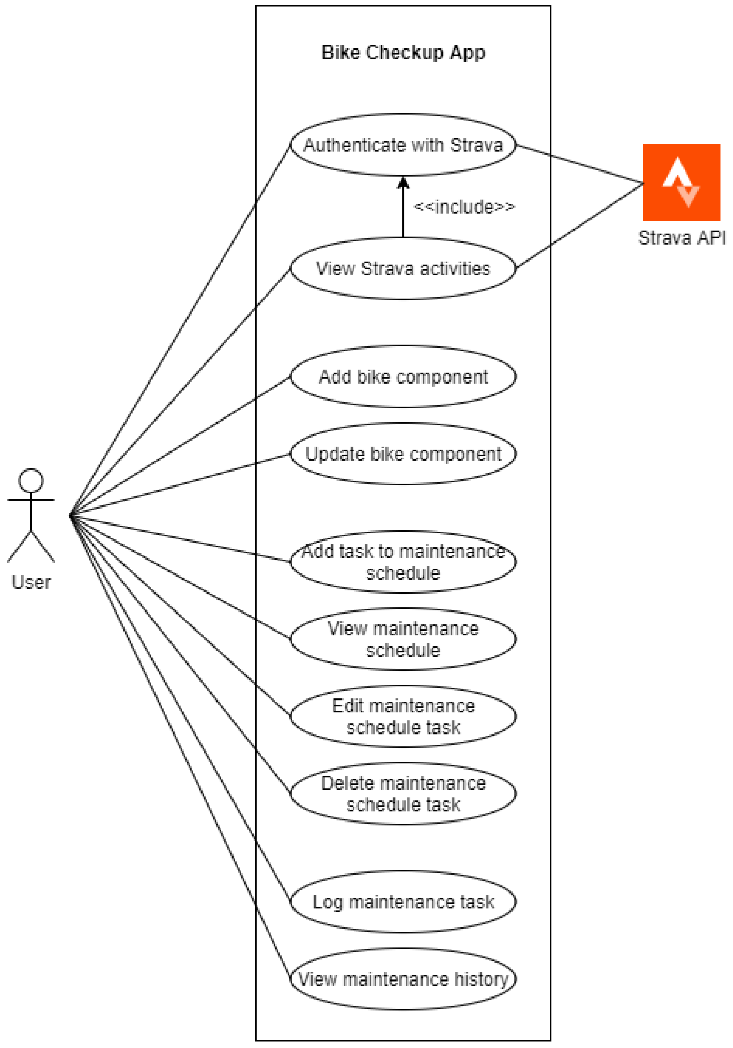
Milestone 3: Requirements

Project Description

Many cyclists are aware of the importance of keeping their bike in good shape for riding. However, without software, it is difficult to keep track of bike/parts usage and time-consuming to update maintenance history lists. Bike Checkup makes these tasks easier by providing cyclists a secure place to keep track of all bikes and their components, as well as create their own bike maintenance schedule.

Use-Case Diagram



Non-Functional Requirements

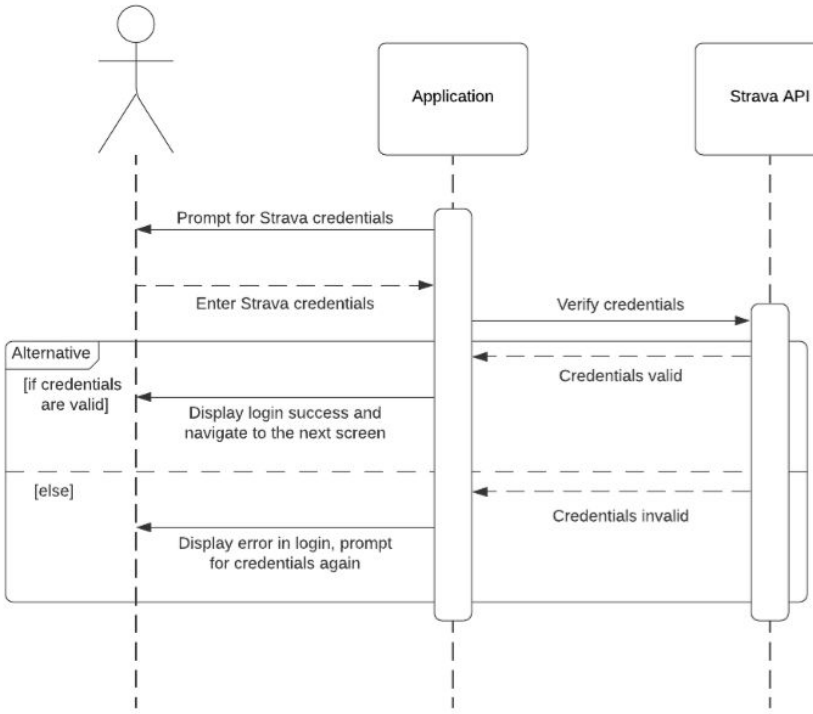
- **Security:** App's Strava API "secret" used for authentication is stored in a file that is encrypted prior to being pushed to the source tree.
    - This non-functional requirement is relevant to the Strava authentication use-case of my app, which requires the developer to obtain a client secret which "should never be shared".
- **Usability:** App design takes accessibility into account by having sufficient contrast, acceptable touch target sizing and spacing, and dynamic text sizing capabilities. The targets I will be using for this requirement are outlined in Material Design's accessibility guidelines, namely choosing a color scheme using Material Design's Color Tool to achieve sufficient contrast, having touch targets be at least 48 x 48 dp in size and separated by at least 8 dp of space, and marking text and their associated containers to be measured in scalable pixels to enable system font size [in an Android app].
    - I believe this non-functional requirement should be essential to any app in order to make it accessible for use by anyone and everyone.

Main System Components

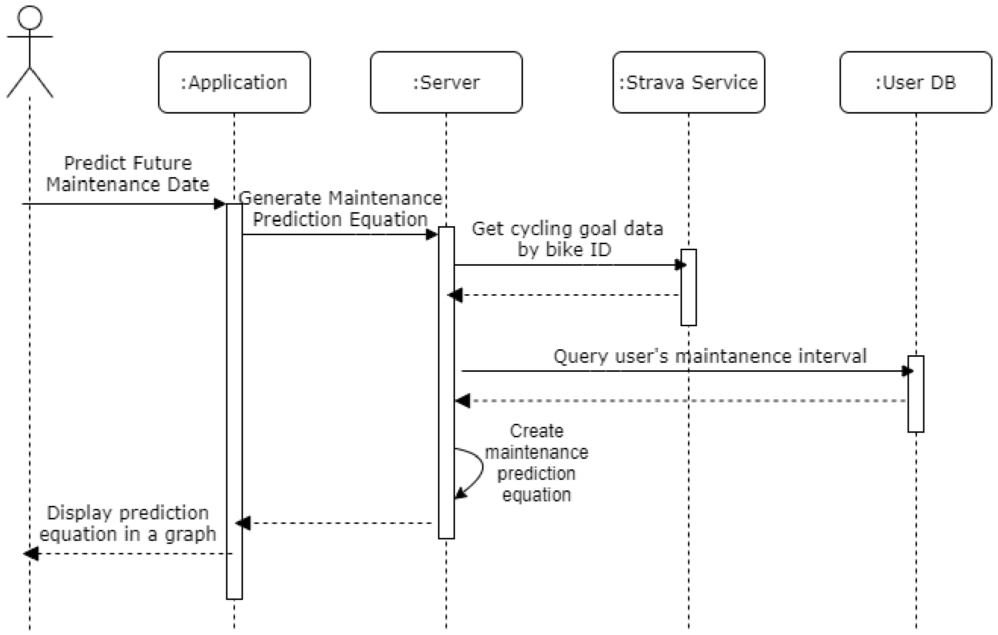
- User Database
    - Will store a backup for each user and their data (including the user's components, maintenance schedule, maintenance history, and Strava information
- Backend Server
- Frontend Mobile Application
- Strava API

Sequence Diagrams

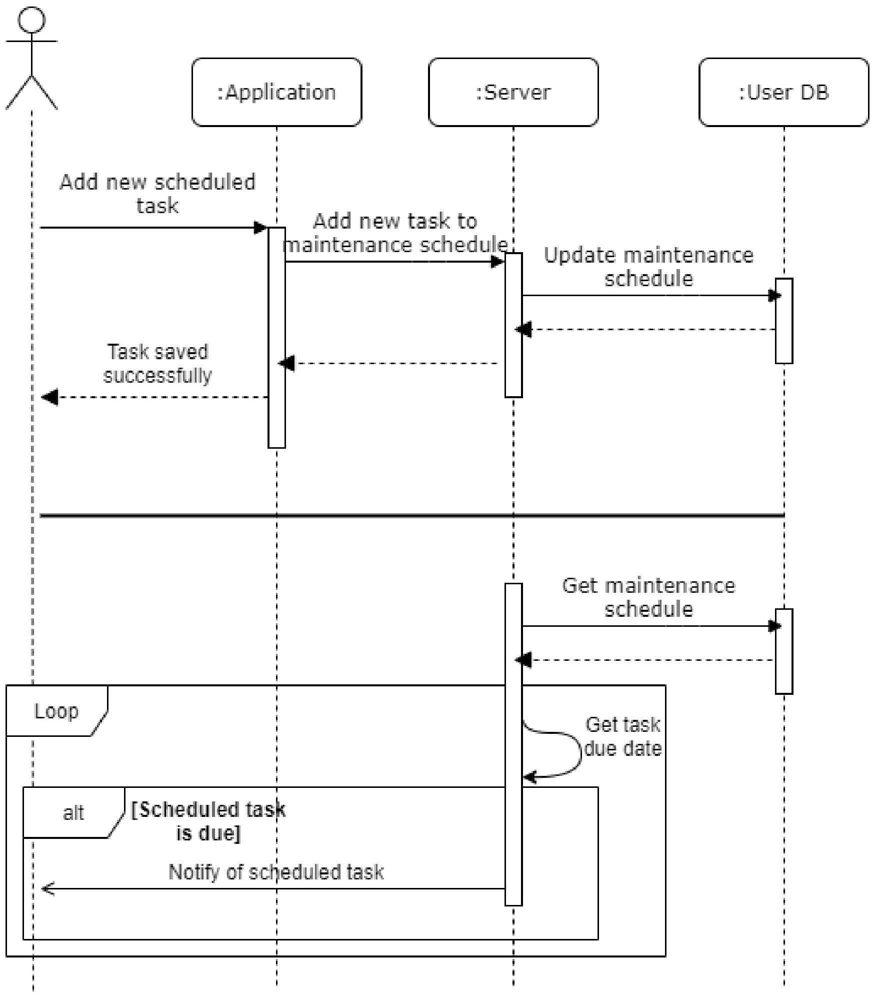
Strava Authentication (External API Call)


Application
Strava API
Prompt for Strava credentials
Enter Strava credentials
Verify credentials
Alternative
[if credentials are valid]
Credentials valid
Display login success and navigate to the next screen
[else]
Credentials invalid
Display error in login, prompt for credentials again

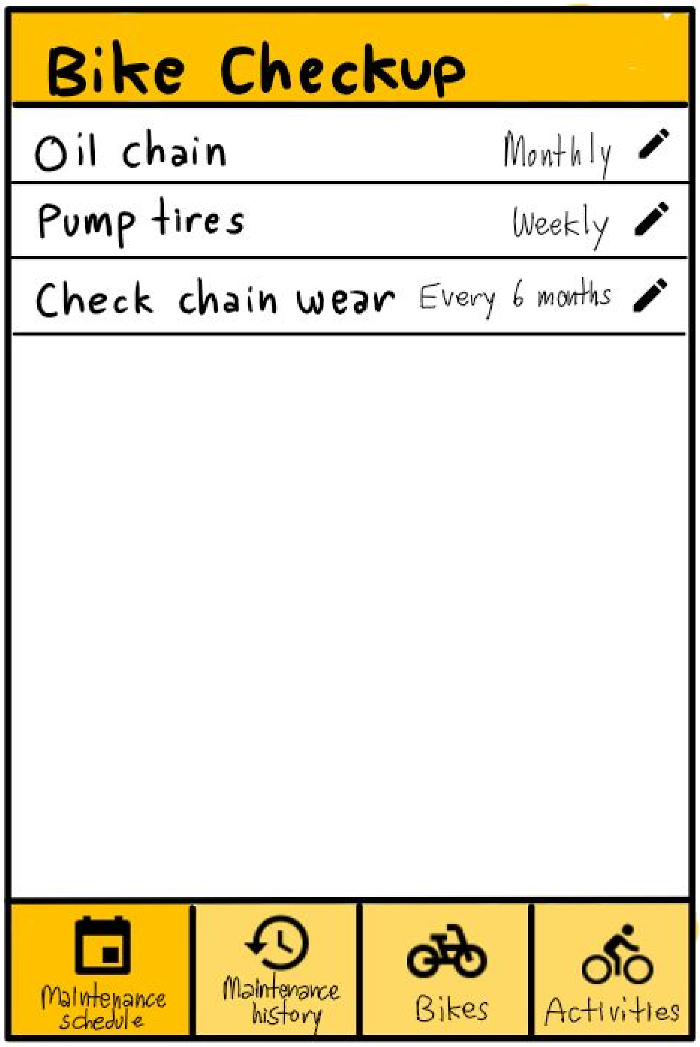
Predict Future Maintenance Date (Complex Logic)


:Application
:Server
:Strava Service
:User DB
Predict Future Maintenance Date
Generate Maintenance Prediction Equation
Get cycling goal data by bike ID
Query user's maintanence interval
Create maintenance prediction equation
Display prediction equation in a graph

Schedule Task Use (Live Updates)



Main App Screen



Milestone 4: Design

Main System Modules and Interfaces

The back-end of our application is composed of the Strava API, a database component, a server storage component and two groups of components that we refer to as “Repositories” and “Services”, that are inspired by the Domain-Driven Design (DDD) principals, and the front-end is a stand alone component referred to as the “Client”.

Database

The purpose of the database is to store all data that the app will use aside from larger resources like images.

Server Storage

The purpose of the server storage is to store all other resources that are larger in size like images, that are not the best use of space in a database.

Repositories

Every repository component has the same purpose, which is to send new data to the database, and to fetch, delete and update data from the database. The repository layer is effectively a layer used to decouple dependencies from the services to the database, and decouple dependencies of unrelated database entities from each other when used in services. Thus there is a repository component that corresponds to each set of related entities stored in the database.

The following are all of the repository components:

- Bike Repository
- Component Repository
- User Repository
- Activity Repository
- MaintenanceHistory Repository
- Task Repository

Services
Each service is similar in that they are responsible for filtering and transforming the entities provided by the repos into the views will be displayed in the client, and transforming the views that are created, or updated in the client into entities to be created or updated. They may also implement other functionalities.

The following services all perform view and entity transformations alone:

- Gear Service
- User Service
- MaintenanceHistory Service

Other Services:

Activity Service
In addition to the standard service functionalities, the activity service will interact with the strava api to get the user’s recent activity so it can be saved within the application to update the usage of each component by other services

MaintenanceSchedule Service
In addition to the standard service functionalities, the MaintenanceSchedule Service will make predictions of when each component related to a particular service task should be replaced based on user settings and the usage of the component.

Resource Service
The resource service does not interact with any repos or with the database, instead it is used to access resources from the server storage.

Standard Repository Interfaces

Each repository will have the same 4 interfaces:

Get Interface

The get interface takes a json object as a parameter which will be used to filter entities from the database, and will return a list of database entities that match the given filter.

Update Interface

The update interface takes an entity as a parameter, and returns true or false value to indicate whether or not the entity was properly updated in the database.

Create Interface

The create interface takes an entity as a parameter and returns a true or false value to indicate whether or not the entity was created in the database.

Delete Interface

The delete interface takes an entity as a parameter and returns a true or false value to indicate whether or not the entity was deleted from the database.

Standard Service Interfaces

Update, Create, Delete Interfaces

Each service will use the same 3 update, create, and delete interfaces as each of the repos, but instead of antaking an entity as a parameter, the service will take a view object as a parameter.

Get Interfaces

Each service will also have a number of interfaces similar to the repo get interface, but instead it will return views, and will still take json objects as a filtering parameter.

Resource Service Interfaces

The resource service has three interfaces:

Get

The get interface takes a string representing an hdd path for the back end server that points to the resource that is being requested, and returns that resource as a byte string representing its bytes.

Create

The create interface takes a byte string representing the resource, as well as a string representing the back end server hdd path that the resource should be stored at.

Delete

The delete interface takes a string representing an hdd path for the back end server that points to the resource to be deleted, and returns a true or false value indicating if the resource was deleted successfully.

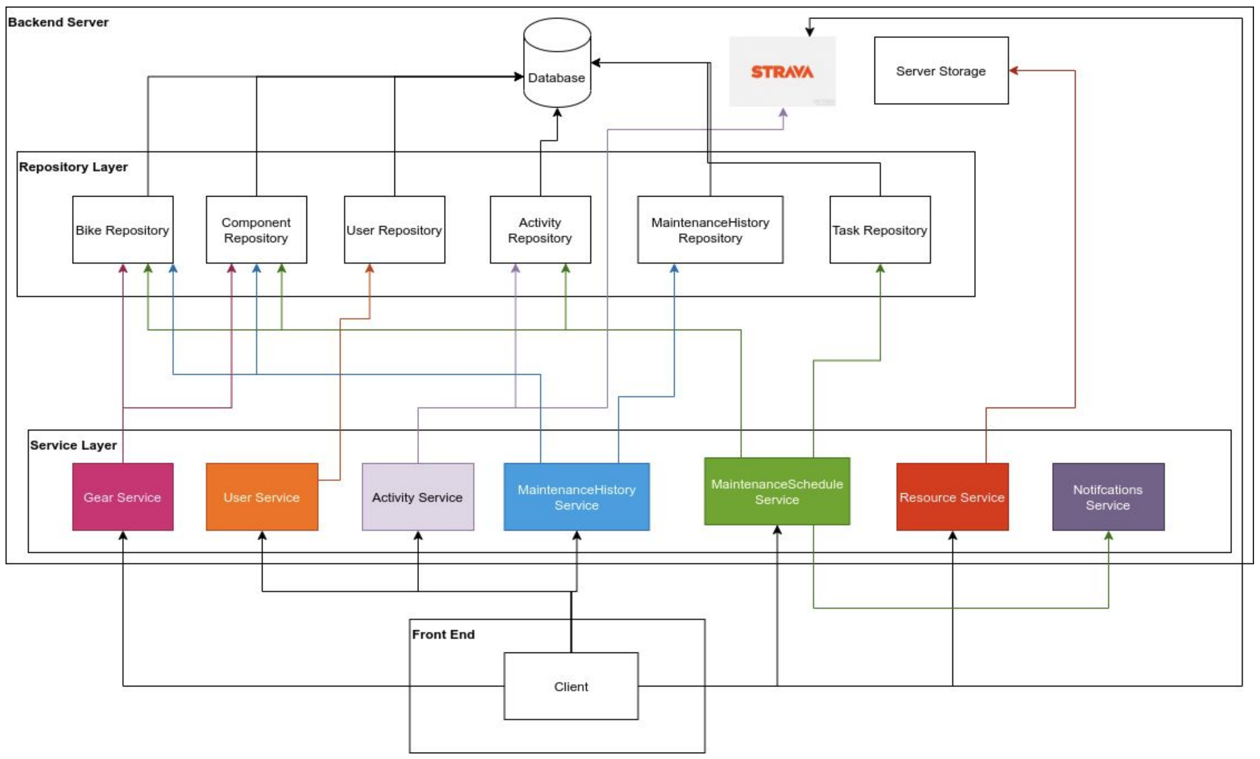
MaintenancePrediction Interface

The MaintenanceSchedule service has one additional unique interface that is unrelated to the others, as it does not simply translate views into entities and call repo functions. This interface will take a task id as a parameter, and will run a maintenance prediction algorithm in order to predict when the component related to the given task should be replaced. The interface will return an equation of a line in the form of a json object with the properties m and b.

FetchNewActivities Interface

The Activity service will also have an unique interface, which will take no parameters, and will return a list of newly logged activities from strava (in json format) that did not previously exist.

High-Level App Architecture



Architectural Patterns

- Client-Server
    - We chose this pattern because it is inherent to the architectural design of the project, being that we have a mobile application acting as our client and our backend serving this frontend.
- Model-View-Controller (MVC) / Domain Driven Design (DDD)
    - These patterns aid in separating data from views, and decoupling low level database interactions from higher level manipulations. In this specific arrangement, the client and the services act as the controller, while the views returned by the services are the same as views in MVC, and the entities passed between the repositories and services are the models. The views incorporate business logic related to the application, and the entities are identical to the database data models.

Language and Framework Choices

- **Database choice:** MongoDB
  - We chose to use a non-relational database over a relational database because we value the flexibility it provides, allowing us to make any modifications during the software development process easily. In addition, our team has less experience with MongoDB compared with MySQL, so we thought this would be a great learning opportunity decision.
- **Front-end choice:** React Native
  - We chose React Native over native Android because prior research into our app idea revealed that similar apps do exist, however no one app supports both iOS and Android. Therefore by going with React Native, we will be giving ourapp a competitive advantage if we end up releasing to the app stores.
- **Cloud provider choice:** AWS (Amazon)
  - We chose AWS over other leading cloud platforms such as Azure (Microsoft) and Google Cloud because it has an established reputation as the oldest and most experienced player in the cloud market. As one of the oldest cloud providers, AWS has established a larger customer base, as well as larger trust and reliability factors. Across multiple detailed comparisons of cloud providers, Amazon tends to edge out the rest.

Plans to Realize Non-Functional Requirements

- **Security:** App's Strava API "secret" used for authentication is stored in a file that is encrypted prior to being pushed to the source tree. This can be accomplish using the “git-secret” tool.
- **Usability:** App design takes accessibility into account by having sufficient contrast, acceptable touch target sizing and spacing, and dynamic text sizing capabilities. To accomplish this, the app will be developed in accordance with the targets outlined in Material Design's accessibility guidelines, namely choosing a color scheme using Material Design's Color Tool to achieve sufficient contrast, having touch targets be at least 48 x 48 dp in size and separated by at least 8 dp of space, and marking text and their associated containers to be measured in scalable pixels to enable system font size [in an Android app].

Complex Logic Algorithm

- **Input:**
    - Bike ID
- **Output:**
    - Predicted due-date of all maintenance tasks for the given bike
- **Main computational logic:**
    a. Acquire activity data for the given bike from Strava
    b. Normalize data (for example, into "km per day")
    c. For each maintenance task in the user's maintenance schedule for the given bike:
        i. Get the task's user-chosen recurrence interval (for example, "every 100km" for the "oil chain" task)
        ii. Normalize task recurrence interval into complementary units to normalized activity data value
        iii. Using the normalized activity data and recurrence interval values, calculate the expected amount of time until the task is due
        iv. Based on the current time/day, convert the above value to an actual due-date for the maintenance task