Design Documentation

for

Rune

Version 2 .0

UCLA CS 130

27 May 2016

Brandon Ly (304-136-729)
Justin Hou (204-155-681)
Roland Zeng (204-150-508)
Alex Wang (103-889-014)
Brendan Sio (804-150-223)
Kevin Tong (704-161-137)
Kevin Zuo (104-201-160)

1. Introduction

This document is designed to be a reference for anyone wishing to implement or any person interested in the architecture or any person starting to develop for Rune. This document contains information about the system architecture, all of its subsystems, the database schema, application controllers, and decision making. Both high level and low level designs are included in this document.

This document should be read by an individual with a technical background and has experience reading data flow diagrams, experience with event-driven programming and API-driven development.

The design document accompanies a software requirements specifications document which outlining the requirements set for this application. This document includes, but is not limited to, the following design details of Rune: system overview; design considerations; architectural strategies; policies and tactics; and detailed system design.

2. System Overview

Rune allows users to create and manage profiles, projects, and tasks. It takes inputted data and manages them through a system with various nodes, routes, and endpoints. The following figure gives a visual representation of Rune's system.

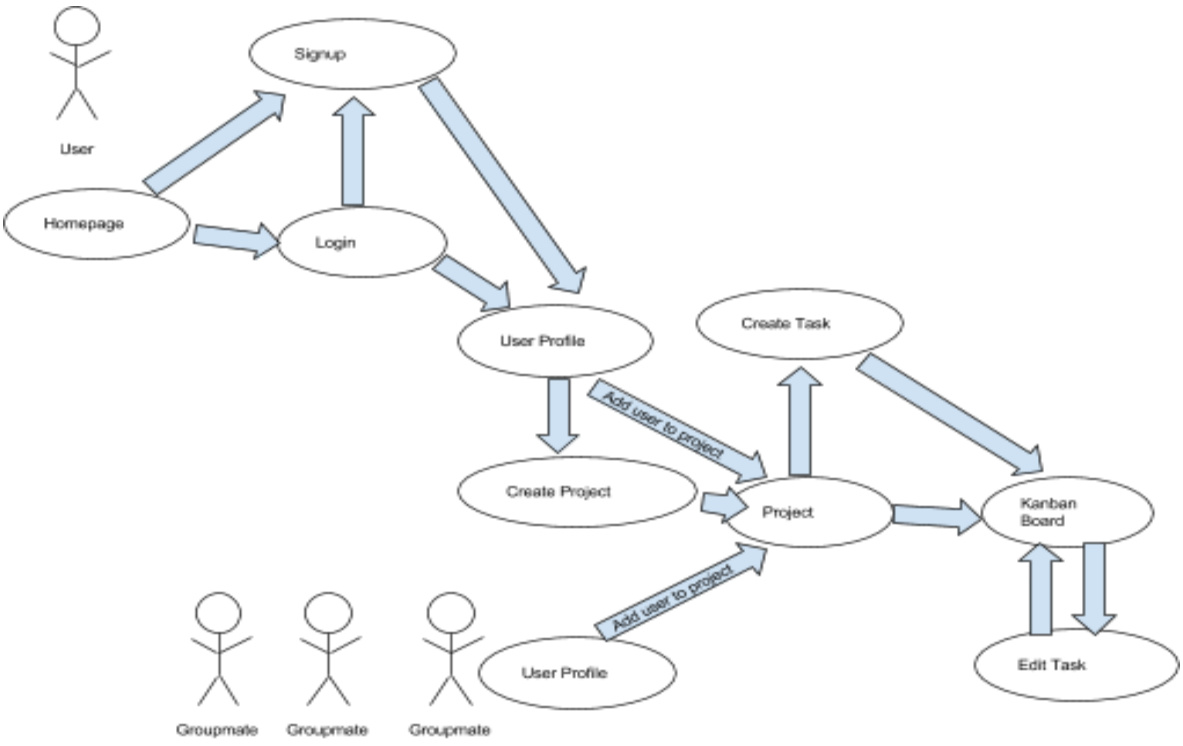

Figure 1: A visualisation of a system overview of Rune.

3. Design Considerations

The objective of Rune is to provide a service for computer science students working in large software project groups. There are various design considerations in regards to this objective, including assumptions and dependencies, general design constraints, design and development goals, and development methods.

3.1. Assumptions and Dependencies

The client application makes the following assumptions about its operating environment:

- The client machine runs supported operating system (Window, Mac OS, Linux).
- The client machine has a TCP/IP internet connection.
- The client machine has a supported browser (Chrome, Firefox, Opera, Safari, not Internet Explorer) installed.
- The client machine has Javascript enabled on aforementioned browser.
- The client machine allows both the application and mLab, a remote database service, through its firewall.

The server must makes the following assumptions about its operating environment:

- The server machine must have a valid operating system installed (Unix or Windows).
- The server machine must have access to mLab's MongoDB database service from it's network.
- The server machine must have Node.js installed, and all packages for the application (found in package.json) installed.
- The server machine must have at least one port open for server to listen on.

The software application makes the following assumptions about the end users:

- Users have a mouse and keyboard and valid email address.
- Users are UCLA students enrolled in specified classes, or professors for those classes.
- Users have a general understanding of Agile development concepts.

3.2. General Constraints

Design and implementation of Rune shall include the following general constraints:

- Rune shall be a web application. It shall have the standard developmental constraints of commercial moderate-traffic web applications (i.e. development of front-end features of the application is limited to modern web browser capabilities).
- Rune shall focus on student-level projects. The web application shall have several strong core features taken from a few of many features from commercial task-tracking systems (i.e. JIRA). Most student-level projects tend to last at most a year, especially course projects, and do not necessarily follow strict deadlines. Furthermore, most students have little to no actual industry-level software engineering experience, and as such will not take as much advantage of the more involved features of commercial task-tracking systems. Therefore, Rune development will focus only on the core features rather than overwhelm students with a plethora of features.

- All server-side interactions with the client-side features on the application shall perform as well as most commercial web services. This means that, on optimal internet connection speeds, Rune shall run server-side operations (i.e. database queries) with a runtime of ~0.1 s average, <5.0s maximum.
- All client-side interactions on the application shall perform as well as most commercial web applications. This means that, on optimal internet connection speeds, Rune shall take ~0.5 s on average, <8.0s at worst to load new pages and dynamic elements on the page. This cascades if certain client-side interactions involve server-side operations; time to perform client-side operations includes the time that each server-side operation takes. Also, all dynamic elements of any page shall run smoothly on at least 60 Hz (the user shall not experience any "lagginess" when using the application).
- Rune shall be as safe to use as any other web application (i.e. safety is therefore up to the users themselves). The web application shall use standard security technology and protocols (i.e. HTTPS, SSH, OpenSSL, etc) to ensure that the application and it's data will most likely not be compromised.

3.3. Goals and Guidelines

Development of Rune involves keeping the following goals and guidelines in mind:

- The KISS principle ("Keep It Simple, Stupid") - It keeps development and construction of features and modules efficient and relieves effort and stress on developers.
- Front-end Design - The development of front-end/client-side modules and features of Rune should have a focus on general ease of use and intuitiveness for student users.
- Core functionality based on JIRA - Rune should work very similarly and as well as JIRA, a proprietary issue tracking system for commercial organizations and their products, projects, services, etc. With student users in mind, Rune will also have extra functionality catered to student users, but developers shall develop Rune with the mindset of creating robust core functionality.
- Security - Project details should be private by default; should students want their projects visible to the public, there are many other viable web services out there suitable for this. Rune shall be developed with security in mind.
- Scalability - The design, architecture, and construction of Rune shall incorporate scalability. Rune is, for now, intended to support the computer science student community at UCLA; in the event that Rune becomes popular and used worldwide, however, having sound design and architecture that is future-proof will enable Rune developers to efficiently scale Rune to serve a drastically larger number of users.

3.4. Development Methods

Development of Rune involves the kanban software development method. The kanban method emphasizes just-in-time delivery of features. Unlike scrum, another popular agile development method, kanban does not utilize time-pressured sprints to accomplish completion of features. Work is divided into tasks that are either added to a backlog or assigned to developers. Actual

development happens on a rolling basis; each developer works on a task at his or her own pace, and upon completion of one task, can check the backlog to pick up other tasks.

4. Architectural Strategies

Rune development employs the use of various architectural strategies to make sound, secure, and scalable software for Rune. Rune development follows general strategies, mainly the model-view-controller architectural pattern, to form the basis of Rune and contribute to producing effective and clean code. Rune development also employs various technologies and frameworks to streamline the design and architecture of Rune.

4.1. Model-View-Controller Architectural Pattern

Rune's architecture follows the Model-View-Controller (MVC) architectural pattern. The architecture has Models for data and how they are stored. Each Model corresponds to a database page. When a user accesses a page or makes a GET/POST request, the Controller handles the request. It parses the request and then manipulates the data Models, either requesting data from the server-side database or inserting/modifying existing data. The application then manipulates the View and ends up with a specific web page output to the user. Section 5 specifies the components in the application's architecture and how it follows one of the three paradigms of MVC.

4.2. Other Strategies

Rune development also saw the use of several miscellaneous architectural, construction, and development strategies.

4.1.2. Code Reuse

Development of Rune involves code reuse in order to simplify and keep consistency of code and technologies. Code reuse consists of a large range of actions, from consistent application of object-oriented programming and design to use of popular software packages, frameworks, and technologies.

4.1.3. Coding conventions and consistency

Consistent coding conventions help keeps code clean and easy to read. It helps developers to minimize on possible errors, and can potentially make debugging a lot easier when needed. See section 6.1. for more information.

4.3. Technologies and Frameworks

Rune builds on the following technologies and frameworks with particular reasons for using them.

4.2.1. JavaScript

JavaScript is a powerful and popular high-level, dynamic, untyped, and interpreted programming language. Because of its proliferation in websites, web application, and with general web software development, JavaScript is ideal for web application programming and for Rune development in particular. The technologies and frameworks involved in Rune development build on JavaScript (Node.js and its packages, jQuery, and Bootstrap, in particular) (more details on why this is so is expanded later in this section).

4.2.2. Node.js

Node.js is a popular and powerful runtime environment for developing server-side web applications. It follows an event-driven architecture but can also implement applications following a model-view-controller architectural pattern. Implementation of Node.js has a powerful package manager called the Node Package Manager (NPM), which easily manages various Node.js software packages provided by a healthy community. Node.js has fewer dependencies and a faster interpreter, providing generally better performance than other popular server-side frameworks.

4.2.3. MongoDB

MongoDB was chosen over relational databases such as MySQL because the database was expected to scale vertically a lot more than horizontally. Nonrelational databases are superior to relational databases when it comes to simple queries and changing schemas, which comprises the majority of Rune's database interactions.

4.2.4. Jade

Jade is an HTML templating engine that synergizes well with Node.js, in particular as a package for NPM. Its strong compatibility with Node.js lends to the simplicity and ease in designing web pages with intuitive user interfaces and the ability to pass data back and forth between the front end and back end.

4.2.5. jQuery and jQuery UI

jQuery is a JavaScript library designed to simplify the client-side scripting of HTML. JQuery is used in Rune to implement dynamic and interactive client-side web pages and to make AJAX calls to Rune's server stack. jQuery UI is a sub-component of jQuery that focuses on the user interface aspects of client-side scripting of HTML.

4.2.6. Bootstrap

Bootstrap is a HTML, CSS, and JavaScript framework for developing web pages with clean design and intuitive interfaces. It streamlines the design process and makes it easy and efficient to apply modern web design principles to the web application.

4.2.7. Compass

Compass is a CSS3 framework. It uses Sass, a CSS extension that gives it many declarative programming features. Compass lets developers easily create web page styles and apply CSS properties.

5. System Architecture

Rune's architecture has various properties, details and patterns. As the architecture follows the model-view-controller (MVC) architectural pattern, Rune mainly has components that make up each of the models, views, and controllers for the architecture. Each of the subsections for section 5 is an aspect of the system architecture and corresponds to either the Model, View, or Controller aspects of the MVC pattern.

5.1. Data Models

Rune's architecture represents data stored in its databases as three main models of objects. They include the User model, the Project model, and the Task model.

5.1.1. User Model

The User model defines and models the properties and attributes of the representation of a user in Rune. The following table lists the properties for the User object in Rune's architecture.

Attribute	Type	Description
First name	String	User's first name.
Last name	String	User's last name.
Email	String	User's email.
Password	String	Salted and hashed password string.
User ID	String	User's unique identifying string.
Project	Array of Project ID and Project Key	List of all the Projects User is in.
User Color	String	Chosen color used for profile and icons.

5.1.2. Project Model

The Project model defines the properties and attributes of the representation of a Rune project. The following table lists the properties for the Project object in Rune's architecture.

Attribute	Type	Description
Project ID	String	Project's unique identifying String
Project Name	String	Project's name
Project Key	String	Project's shortened identifying name
Members	Array	List of Users associated with project
Counter	Number	Number of Tasks in project
Tasks	Array of Task	List of Task model objects inside project
Public	Boolean	Indicates if a project is visible on the Project Finder
Pending Users	Array of String	List of usernames of applicants to a Project.

5.1.3. Task Model

The Task model defines the properties and attributes of the representation of a task in Rune. The following table lists a specification for the Task object in Rune's architecture. Tasks are not stored as individual MongoDB documents; instead they are stored inside each project's "Tasks" array.

Attribute	Type	Description
Project ID	String	Unique identifying String for associated Project
Task Name	String	Name of Task
Task ID	Number	Unique identifying number for Task in Project
Task Description	String	A description of the Task given by a User
Created By	String	Email of User that created Task
Assigned To	String	Email of User that Task is assigned to
Status	String	Current status of Task. Options are: Backlog; Selected for Development; In Progress; Completed
Date Created	String	Date of which the Task was created
Priority	String	Priority of the Task. Can be High, Medium, or Low
Issue Type	String	Issue type of Task. Options are: Core; Feature; Bug

5.2. Client and Server Controllers

Rune's architecture consists of several systems that add, remove, and manipulate data stored in databases and views outputted by the application. These systems that add and edit data are represented as controllers, with three main controllers and several miscellaneous controllers. The three main controllers are the User Account System, The Project Controller, and the Task Controller.

5.2.1. User Account System (Home and Profile Controllers)

The User Account System manages the creation of User accounts and the authentication, authorization, and management of User accounts. Users can create an account through the home page or through the signup page. From these pages, users are prompted to enter their name and email, choose a profile picture color, and create a password for their account. The login system salts, hashes, and then stores the password in Rune's databases, such that if the databases are ever compromised, all users' password information remain relatively private. This is all managed by the Home and Profile controllers.

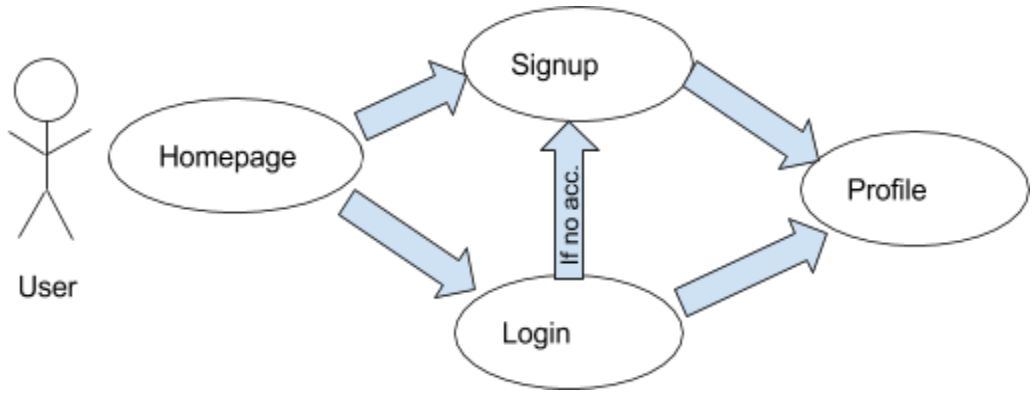

Figure 2: A visualization of the account system in Rune.

5.2.2. Project Controller

The Project controller manages how users create projects. The Project controller enables project creation and project management of its settings through a client-side requests. The Project controller also manages the Kanban Board and the Task Browser.

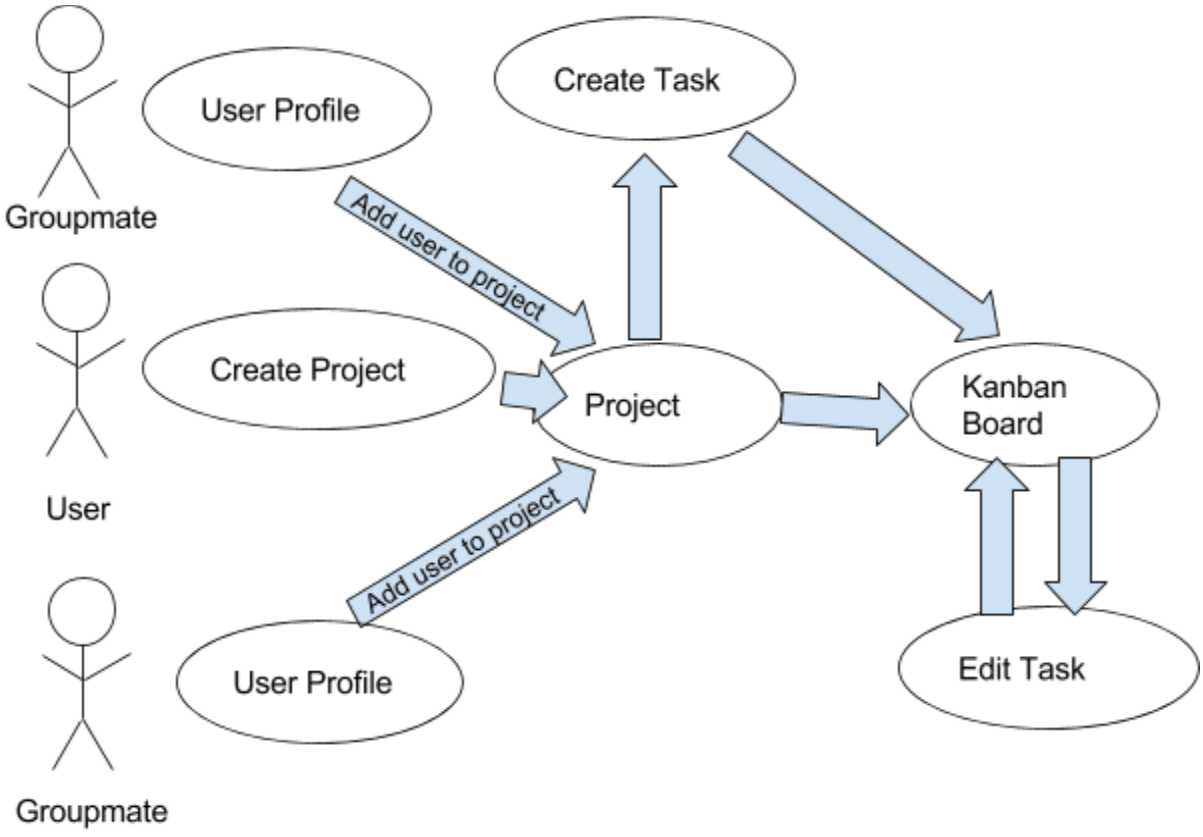

Figure 3: A model of the Project controller component of Rune.

5.2.3. Task Controller

The Task controller manages how users create and edit tasks. It parses user requests and updates the changes to the Task View and Task Browser.

5.2.4. Project Finder Controller

Rune's Project Finder allows users to apply to join other projects. The purpose of the Project Finder is to provide an easier way for students to find group members for a class project, or any project in general. It works by listing every project that has been made public, with an adequate description of what the project entails. The descriptions will display Project name, Project Leader, Members (number), Project description, and skills. If a user is interested for a project, the user can click apply and wait for the project leader to approve. Once approved, the project leader and the applicant will be added to the project group like all other students involved in the aforementioned project.

The Project Finder Controller manages the Project Finder. The Project Finder takes client-side requests and sends them to the Project Finder Controller. The controller then parses the request and updates its associated views.

5.3. Interface and Application Views

The interface and application views represent what the Users see when they use Rune. It represents all visual and interactable aspect of the application, including individual pages, forms, etc. There are separate views for various parts of the application and what the Controllers manage, including the following:

- User Profile View
- Project View
- Kanban Board
- Task Browser
- Task View
- Project Finder View

The User Profile view is defined mainly by the Profile page. When new Users sign up for Rune, they will each be given a unique page with the purpose of adding personal descriptions as well as showcasing past projects. The Profile page will enable Users to see what other Users have done in the past, and Project Leaders could determine whether an applicant is qualified for the proposed project. On the Profile page, Users could edit their own personal description, and customize the page to choose the color of the Rune icon the User desires.

The Project view details all progress of a particular project. The Project view incorporates the Kanban Board and task browser. It also manages display of the project group members, tasks, and settings.

The Kanban Board allows Users to view all Tasks in a Project on a single, interactable 4-column dynamic virtual sticky board. Each of the four columns corresponds to a different Task status.

The Kanban Board has drag and drop capability, where dragging each task block to a different column will change that Task's status to the corresponding status under the column.

The Kanban Board is controlled by the Project controller. The Kanban Board takes client-side requests and sends them to the Project controller, in which the Project controller then parses the request, changes the Project and Task data and other data as it sees fit, and sends the Project controller

The Task Browser enables Users to browse all the tasks of a given Project on a single page. Users can filter and sort tasks by different categories, including by priority, status, users, and date. When filtering by priority, Users could choose to show tasks of an interested priority: if Users only want to see tasks of high priority, only tasks of high priority will be shown. When filtering tasks through status, Users could choose to show tasks of a given status, such as 'in progress' or 'completed'. Tasks that are filtered by users will only show the tasks that are assigned to the user. When sorting by date, the tasks could be sorted from newest to oldest and vice versa. Also, Users can sort the priority from highest to lowest as well.

The Task view encompasses all of the task functionalities, including task page and task browser. The Task view Upon creation, tasks can be viewed either on the kanban board on the project page or on the task browser. On the individual task page, users are given the option to edit task details or to delete the task.

The Task page allows Users to edit tasks, delete tasks, and add comments to the tasks. Users can access the task page by clicking on a task name on the Kanban Board. While on the task page, Users can find details of the task, including priority, status, and who the task is assigned to. Final edits of the task could be fabricated through the 'submit' button.

The Project Finder view manages the Project Finder and all of its details. It lists projects that visible to the public and enables Users to apply to, or opt to be a part of, the project

6. Policies and Tactics

6.1. Coding Conventions

The major frameworks that Rune builds on use JavaScript as the programming language. The application's construction, therefore, follows popular programming style and conventions for JavaScript. In particular, all JavaScript code follows the Google JavaScript style guide (https://google.github.io/styleguide/javascriptguide.xml).

In addition, JavaScript code in Rune's construction adheres to the following conventions:

- Indentations shall consist of four spaces.
- Statements shall end in a semi colons.
- Variable and function names shall use camelcase.
- One true brace style shall be used, in which the opening curly bracket of a block is placed on the same line as its corresponding statement or declaration.

6.2. Version Control and Git

Rune development uses Git for version control system due to the very modular architecture and relatively large developer base of Rune. It also uses GitHub to host the remote repository for Rune. Changes to modules and implementations of features come in the form of Git commits, which contain changes done to the application codebase and are pushed individually from developers. Use of Git also helps easily visualize changes in Rune development. Furthermore, in the case of errors that get accidentally committed, Git easily allows developers to roll back to an earlier version. Overall, Git and GitHub provide a straightforward and efficient version control system to develop Rune.

6.3. Development Strategy

Development and construction of Rune involves the kanban method. The development team holds weekly meetings where tasks are decided on and the only hard deadlines are the ones enforced supervisors and other individuals of higher authority. The development team works continuously to add features while coordinating with the kanban task board and collaborating with teammates at meetings and through online communication methods (instant messaging, etc).

7. Detailed System Design

The detailed system design section elaborates the system architecture components of Rune explained in section 5 in greater detail.

7.1. Data Model Descriptions

7.1.1. Detailed User Model Specification

Classification Model - MongoDB Schema

Definition The User model defines and models the properties and attributes of the representation of a User in Rune.

Responsibilities The User object stores the attributes of each individual user and allows controllers to search for users with said attributes.

Constraints Each User must represent a single person, who is using the Rune application. The person representing each User must have a valid email address and

Uses/Interactions The User class is used for authentication, each person using the Rune application must create and account and log-in using a password stored as a salted hash in the User class. Each User may have a list of Projects they are a member of, and may modify and add tasks to said project. Each User also has a profile page that acts as a public facing interface for other Users to see.

Resources and Processing User objects are stored in a MongoDB database, and thus requires hard disk memory.

Interface/Exports

Attribute	Type	Description
First name	String	User's first name.
Last name	String	User's last name.
Email	String	User's email.
Password	String	Salted and hashed password string.
User ID	String	User's unique identifying string.
User Color	String	Chosen color used for profile and icons.
Project	Array	List of all the Projects User is in.

User.findOne([criteria], [projection], [callback])

- Purpose
  - Finds one user with the given criteria
- Parameters
  - criteria – mongodb selector
  - projection – optional fields to return
  - callback – function to use the found models

User.find([criteria], [projection], [callback])

- Purpose
  - Finds a number of users with the given criteria
- Parameters
  - criteria – mongoDB selector
  - projection – optional fields to return
  - callback – function to use the found models

User.remove([criteria], [projection], [callback])

- Purpose
  - Removes a user with the given criteria from the database
- Parameters
  - criteria – mongodb selector
  - projection – optional fields to return
  - callback – function to use the found models

7.1.2. Detailed Project Model Specification

Classification Model - MongoDB Schema

Definition The Project model defines the properties and attributes of the representation of a Rune project.

Responsibilities The Project class stores the attributes of each individual Project and allows controllers to search for users with said attributes.

Constraints Each User must represent a single person, who is using the Rune application. The person representing each User must have a valid email address and

Composition The Task model is a subcomponent of the Project model.

Uses/Interactions The User class is used for authentication, each person using the Rune application must create and account and log-in using a password stored as a salted hash in the User class. Each User may have a list of Projects they are a member of, and may modify and add tasks to said project. Each User also has a profile page that acts as a public facing interface for other Users to see.

Resources and Processing Project objects are stored in a MongoDB database, and thus requires hard disk memory.

Interface/Exports

Attribute	Type	Description
Project ID	String	Project's unique identifying String
Project Name	String	Project's name
Project Key	String	Project's shortened identifying name

Members	Array	List of Users associated with project
Counter	Number	Number of Tasks in project
Tasks	Array of Task	List of Task model objects inside project
Public	Boolean	Indicates if a project is visible on the Project Finder
Pending Users	Array of String	List of usernames of applicants to a Project.

7.1.3. Detailed Task Model Specification

Classification Model - MongoDB Schema

Definition The Task model defines the properties and attributes of the representation of a Rune task.

Responsibilities The task class stores the attributes of each individual Task and allows controllers to search for Tasks within Projects with said attributes.

Constraints Each Task must represent a single task within a project.

Uses/Interactions Each Task is created by a User and has a User assigned to it. Each task is also enclosed within a Project. Users may edit tasks and change the status of tasks within Project.

Resources and Processing Task objects are stored in a MongoDB database, and thus requires hard disk memory.

Interface/Exports

Attribute	**Type**	**Description**
Project ID	String	Unique identifying String for associated Project
Task Name	String	Name of Task
Task ID	Number	Unique identifying number for Task in Project
Task Description	String	A description of the Task given by a User
Created By	String	Email of User that created Task
Assigned To	String	Email of User that Task is assigned to
Status	String	Current status of Task. Options are: Backlog; Selected for Development; In Progress; Completed
Date Created	String	Date of which the Task was created
Priority	String	Priority of the Task. Can be High, Medium, or Low
Issue Type	String	Issue type of Task. Options are: Core; Feature; Bug

7.2. Data Controller Descriptions

7.2.1. Detailed User Controller Specification

Classification	Controller - Node.js Module
Definition	This controller handles account creation by new Users, login attempts and subsequent session creation, to handle account creation, User login, session handling
Responsibilities	The User controller handles account creation and login. The controller accepts POST requests from forms and creates accounts, then logs Users in and creates a session.
Constraints	The User controller must accept inputs as POST requests only from forms on the front-end. The User controller is limited by inputs from the front-end views of our project. The login and account creation is limited by the Node package “passport”.
Composition	The User controller composes of subcontrollers that manage login, account creation, and sending data to the User View. ● The Login Controller creates a new session for users who log in through the authentication front end. ● The Account Creation Controller creates new user accounts. It stores passwords in salted and hashed formats, along with the new user information. ● The Profile Controller dynamically generates User information and sends it into the User View to generate a user profile page, giving users a front-facing interface to provide to other users.
Uses/Interactions	The User controller receives input from the login and account creation views. These views have forms that send POST requests to the controller, allowing it to
Resources and Processing	The User controller requires session creation from the client side. The session is stored as a cookie in memory on the client’s machine. The controller processes all require server processing power and memory.
Interface/Exports	*Endpoint* app.get(‘/’)
	Purpose Generates a home page with signup form for new users and returning users to enter application from
	Parameters N/A

Endpoint app.get(‘/profile’)

Purpose Redirects user to own profile

Parameters N/A

Endpoint app.get(‘/login’)

Purpose Generates login form for users to log in with

Parameters
- Email - user email address
- Password - plaintext password that will be hashed and checked for match

Endpoint app.post(‘/login’)

Purpose Accepts POST request from login page and creates session if user is authorized with correct login and password

Parameters N/A

Endpoint app.get(‘/u/:userid’)

Purpose Generates unique user profile page

Parameters
- :userid - Unique user ID

Endpoint app.get(‘/signup’)

Purpose Generates a sign up page for users to submit their information and create a new account

Parameters
- Name
- Email
- Password
- Validation check

Endpoint app.post(‘/signup’)

Purpose Takes POST request from signup form and creates a new User account with hashed password

Parameters N/A

Endpoint	app.get(‘/u/:userid/adduser/:projectid’)
Purpose	Generates a home page with signup form for new users and returning users to enter application from
Parameters	● userid - unique ID of User being added to project ● projectid - unique ID of Project
Endpoint	app.post('/u/:userid/edit’)
Purpose	Form request that changes User profile details and information after account creation
Parameters	● userid - unique ID of User
Endpoint	app.post('/u/:userid/edit/delete’)
Purpose	Endpoint for user to delete their own account if they so choose
Parameters	N/A

7.2.2. Detailed Project Controller Specification

Classification Controller - Node.js Module

Definition This is a controller that manages the data for Projects

Responsibilities The Project Controller is responsible for creating, editing, and deleting projects, along with serving data the project views.

Constraints Each project must be created by a user. The project themselves have tasks within them.

Composition The Project Controller composes the following functions:

- Each project has a “admin” user, or user in charge of the project.
- Additionally, each project has a system of tasks within it.
- The Kanban Board is managed by the Project Controller and serves as its own subsystem, getting data from the Project Controller and passing POST requests back asynchronously.
- The Task Browser and managing data to be parsed for the Task Browser in the Project View.

Uses/Interactions The Project Controller has several uses and interactions with other parts of the application’s architecture:

- Each Project is created by a User and is managed by Users.
- Projects have Tasks within them. Tasks may be moved around inside the project and edited.
- Kanban Board View information is generated by the Project Controller, and passes POST requests back asynchronously to the Project Controller, allowing for dynamic editing.

Resources The controller processes all require server processing power and memory.

Interface/Exports

Endpoint app.get('/createproject)

Purpose Generates a page with a form to create a new project

Parameters N/A

Endpoint app.post('/createproject')

Purpose Take the POST request generated by the create project form and adds a new project to the databae

Parameters

- projectname - name of new project to be created
- projectdescription - description of new project to be created
- useremail - email of user who created project

Endpoint app.get('/p/:projectid/')

Purpose Generates project page with all tasks and users and project description

Parameters

- projectid - unique id of project

Endpoint app.post('/p/:projectid/edit/')

Purpose Allows users to update description of project and add and/or edit the GitHub repository for the project

Parameters

- projectid - unique id of project
- projectdescription - description of new project to be created
- github_url - URL of GitHub repository

Endpoint app.post('/p/:projectid/deleteproject/')

Purpose Deletes project after confirmation and removes from database

Parameters N/A

7.2.3. Detailed Task Controller Specification

Classification Controller - Node.js Module

Definition This controller manages Tasks that are members of Projects.

Responsibilities The Task Controller is responsible for creating and editing tasks for individual projects, and for deleting them if necessary.

Constraints The Task Controller is constrained by the Projects. Each Task must be a member of a Project.

Composition Each task has its own attributes, but no smaller subsystems.

Uses/Interactions Each Task is a member of a Project, and thus its attributes are tied to the Project's.

Resources The controller processes all require server processing power and memory.

Interface/Exports

Endpoint app.get('/p/:projectid/createtask/')

Purpose Generates a page where users can create a new task

Parameters

- projectid - unique identifier for project

Endpoint app.post('/p/:projectid/createtask')

Purpose Takes the information from the createtask endpoint and creates a new task within the given project

Parameters

- Parameters
- taskname - name of new task
- taskdescription - description of new task
- createdby - ID of user who created the task
- assignedto - ID of user task is assigned to. Null if unassigned
- status - current status of new task

Endpoint app.get('/p/:projectid/t/:taskid/')

Purpose Generates a task page with all of the information and history of the given task

Parameters
- taskid - unique identifier for task
- projectid - unique identifier for project

Endpoint app.get('/p/:projectid/t/:taskid/edit/')

Purpose Generates a page for users who are a member of the project to edit a given task

Parameters
- taskid - unique identifier for task
- projectid - unique identifier for project

Endpoint app.post('/p/:projectid/t/:taskid/edit/')

Purpose Takes a POST request and edits a task with the given information

Parameters
- taskname - name of new task
- taskdescription - description of new task
- createdby - ID of user who created the task
- assignedto - ID of user task is assigned to. Null if unassigned
- status - current status of new task

Endpoint app.post('/p/:projectid/t/:taskid/archive/')

Purpose Changes a task's status to "Archived"

Parameters
- taskid - unique identifier for task
- projectid - unique identifier for project

Endpoint app.post('/p/:projectid/t/:taskid/unarchive/')

Purpose Changes an "Archived" task's status to "Completed"

Parameters
- taskid - unique identifier for task
- projectid - unique identifier for project

Endpoint app.post('/p/:projectid/t/:taskid/delete/')

Purpose Deletes a given task

Parameters
- taskid - unique identifier for task

- projectid - unique identifier for project

Endpoint	app.post('/p/:projectid/t/:taskid/movetask/')
Purpose	Edits a given task's status after moving it on the Kanban board
Parameters	• taskid - unique identifier for task • projectid - unique identifier for project • status - new status of task
Endpoint	app.post('/p/:projectid/t/:taskid/comment/')
Purpose	Saves a comment into the given task
Parameters	• taskid - unique identifier for task • projectid - unique identifier for project • comment - comment to be added to task • githubcommit - github commit, if there is one, to be added to comment
Method	getProject()
Purpose	Returns the Project that the task is a member of
Parameters	• Parameters • taskid - unique identifier for task
Method	setProject()
Purpose	Assigns a Project to the given Task
Parameters	• taskid - unique identifier for task • projectid - unique identifier for project
Method	getUsersList()
Purpose	Returns a list of all the users that are members of the project that this task is a member of
Parameters	N/A
Method	redirectToTask()
Purpose	Redirects user to page generated for that Task

Parameters
- taskid - unique identifier for task
- projectid - unique identifier for project

7.2.4. Detailed Project Finder Controller Specification

Classification Controller - Node.js Module

Definition This controller handles the functionalities of the project finder. In order to do this, the Project Finder controller must keep track of which projects are listed as public and handle requests by students to apply for selected projects.

Responsibilities The user can click “Apply” when viewing selected public projects in the project finder. This generates a POST request to the Project Finder controller, which must handle the request by adding the user to a list of applicants for the project. The controller must also allow for the removal of users from this list, which occurs if they are accepted (added to the project) or declined, after which in both cases the user is removed from the pending applicants list for the respective project.

Constraints The Project Finder must only accept inputs as POST requests from front-end input forms. The Project Finder controller is constrained to only inputs from the front end views of our project.

Composition The Application controller handles when the user clicks “Apply” for a selected listed public project. This will add the user to the list of pending applicants for the project selected.
The Response controller handles when a project member “Accepts” or “Declines” a pending application for his/her project. The controller will be notified of the choice--if the choice was “Accept,” the controller will add the user to the list of members for the selected project; if the choice was “Decline,” the controller will not add the user to the list of members for the selected project. Both occurrences will then be followed by the prompt removal of the specified user from the pending applicants list of the selected project.

Uses/Interactions The Project Finder controller will only receive input from either the project finder page (from users applying to projects) or the pending applicants section of the project page (from users accepting / declining pending applicants for their project). These interactions are handled via POST requests.

Resources and Processing The Project Finder controller requires server processing power and memory.

Interface/Exports	*Endpoint*	app.get(‘/projectfinder’)
	Purpose	Generates a list of all public projects
	Parameters	N/A
	Endpoint	app.post(‘/projectfinder/:projectid/apply)
	Purpose	Sends a POST request with an application to the given project to apply as a member of that project
	Parameters	projectid - unique identifier for project
	Endpoint	app.get(‘/project/:projectid/applications)
	Purpose	Generates page with all current applications from users for a project
	Parameters	projectid - unique identifier for project
	Endpoint	app.post(‘/project/applications)
	Purpose	Rejects or denies applications to projects. Accepted users will be added to the project. All users responded to will be removed from the application
	Parameters	projectid - unique identifier for project

7.3. Application View Descriptions

7.3.1. Detailed User View Specification

Classification	View - Jade HTML Templates
Definition	Manages the HTML web pages that appear to the Users in relation to viewing and editing their User profiles.
Responsibilities	The User View renders the login and signup page and the User Profile Page, and for displaying proper information received from Models and Controllers.

Uses/Interactions The User View interacts with the User Controller and the User Model to correctly display User Profile information and to correctly update Views when information is changed.

Resources and Processing User View processing requires server processing power and memory.

7.3.2. Detailed Project View Specification

Classification View - Jade HTML Templates

Definition This view is the central hub of each project the user is a member of. It has multiple tabs, including Overview, Kanban Board, Users, Tasks, and Settings.

Responsibilities The Project view is responsible for rendering the Project Page and its subviews, and for displaying proper information received from Models and Controllers.

Constraints The Project view is constrained by the existence of projects in which the user is a member.

Composition The Project View composes of a few subviews:

- The Overview tab shows general information like the project name, github repository, description, and a detailed, most-recent sorted project history. The user is able to click the link to the project repository, if one exists, and is also able to click on links in the task history to directly see the task pages of each recent change.
- The Kanban Board shows a dynamic view of all tasks of the project. Every task is displayed in a small box, with information like task name, description, assignee, and priority listed clearly. They are listed in the categories: Backlog, Selected for Development, In Progress, and Completed. The user is able to click on tasks to view their respective task pages, or drag-and-drop tasks from one category to another in order to update its status.
- The users tab lists all users who are members of the project, with clickable links (to their respective profile pages) and their icons shown. The email of each user is also listed here.
- The Task Browser tab displays, in a browsable and navigable window, detailed task information as explained in section 7.3.3.
- The Settings tab allows the user to edit project details like description, github repository, project finder visibility (public/private), and relevant project skills. The user is able to update these fields via the “Update Project Settings” button. The user is also

able to delete the project via the “Delete Project” button. At the bottom of this tab is the Project Applicants section, which lists all pending applicants for the project (discovered via the project finder). The user may click “Accept” to add the user into the project or “Decline” to remove the user from the pending applicants list.

Uses/Interactions The Project View interacts with the User and Project Controller and the User, Project, and Task Models to correctly display Project View information and to correctly update Views when information is changed.

Resources and Processing The Project View requires client-side processing power and memory to load.

7.3.3. Detailed Task View Specification

Classification View - Jade HTML Template

Definition This view shows all tasks for the current project at hand, allowing users to filter, view, add and delete comments, and link GitHub commits from the repository

Responsibilities The detailed task page serves as an interface to view details about a project and to edit tasks. The task page has a modal to edit the project along with buttons to archive completed projects.

Composition The Task View composes of the following subsection, some in regards to the Task’s associated Project:

- An edit task function that generates a modal to edit the task
- An archive/unarchive button that shows when a task is completed/archived
- And a comments section with all the comments curre

Uses/Interactions The user is able to view the details of the task at the Task View. User may archive or unarchive tasks that are completed or archived, respectively. Users may also add comments from the task view, and view all previously added comments. When adding comments, the Task Controller sends over a list of all GitHub commits with the GitHub repository associated with the project, and the user may link one of these commits along with the project.

Resources and Processing The Task View requires client-side processing power and memory to load.

7.3.4. Detailed Project Finder View Specification

Classification View - Jade HTML Template

Definition	This page lists all public projects seeking new members and allows users to apply for projects of their choosing.
Responsibilities	The project finder page must list all public projects, including each one's name, leader, number of members, description, skills, and an apply button.
Composition	The user is able to click the project leader's name to visit his/her profile page. The users should also be able to read all of the data in each project's description and skills tabs. Upon clicking the "Apply" button, the user is then successfully added to the project's pending applicants list. This will allow the project's members to accept or decline this application from the project page.
Uses/Interactions	The Project Finder View interacts with the User and Project Finder Controller and the User and Project Models to correctly display Project Finder information and to correctly update Views when information is changed.
Resources and Processing	The Project Finder View requires client-side processing power and memory to load.