RAPDB: The Rapid Social Application Deployment Service

Lance Tyler, Matthew Morris
California Polytechnic University, San Luis Obispo
Advising: Dr. Gene Fisher

December 14, 2014

Contents

1 Introduction

1.1 Problem Statement

Many application developers today experience the frustration of wanting to rapidly deploy some new social media application, or essentially an idea, but are quickly faced with the daunting task of developing a back-end server and designing the associated database that are necessary to accommodate such an application. Given the fact that these services have been developed countless times before for other applications, it would make sense that there would exist some generic platform of services that provides these services to casual app developers. With our RAPDB service, we aim to provide these services in a generic fashion in an effort to reduce the amount of development time necessary to deploy an application by eliminating the need to reinvent the programmatic wheel or spend countless hours developing the parts of an application that is neither visible to end-users, nor unique to the application itself.

1.2 What is a Social Application?

A social application is a way that multiple users can interact with each other with the use of technology. The types of software in the category are endless, and immensely popular and profitable. The most profitable and successful social application is Facebook, which makes roughly 22M dollars a quarter from ads, and has over one billion users.

1.3 Patterns In Social Applications

When developers look over the examples of Social Applications you may notice that certain patterns appear. One of the most prominent patterns is that of connected or associated users, which is a key aspect that defines the application in question as a social one. One major privavy issue with publicly accessible social applications is that of authenticating users, validating their permission(s) as a specific user, and enforcing social application logic based on user roles or categories (i.e. not making one user's "private" data accessible to the public).

1.3.1 Associations

In most social applications, all users typically have the same user role or user abilities. However, users often want to associate with other users on differing levels. An example could be as user doesn't want to show their co-workers their late night drinking habits, so any pictures of the user in a bar should be excluded from users they consider their co-workers. By specifying an association level between users, associated users can be categorized into groups that mirror real-life social groups, which then allows social applications to provide customized experiences between users on the basis of their association level.

Associations are mutual, so the process to become 'friends' should consist of a friends request, and then the request receiver accepting the request. Some applications take this concept further, and provide different types of relationships between users, or even groups for users to belong to. Our goal is to provide this functionality in a generic fashion, so that social application developers of all types can make use of user associations.

1.3.2 Sharing

Once a user is linked to another user, they will want to start sharing information with one another. The most common example of information sharing between users is the exchange of text and pictures, which can take a large variety of forms (Facebook posts, Twitter tweets, and other variants). This could be a professional note taker sharing PDFs of the notes he takes with a group of paying students, and no other users. In other cases, a user might want to share some information with all of his associations, regardless of their association level.

Given these requirements, users also need to be able to set association levels or visibility levels on a "per-data" basis, so that users have the granular ability to control who sees their social content. We aim to provide this functionality in a way that allows social application developers to upload and serve data that can be set to specific visibility levels as they please.

1.3.3 Authentication

A user's personal identity is very valuable to them. Once you've forged associations with other users on a social application and shared or stored private data, it is essential that users cannot access other user's private information such as their login credentials or their data.

In addition to initial user authentication, we also need to consider the need for message integrity. That is, we need to verify that every API that is accessed is done so by a user that is both who they say they are, and who is allowed to perform the requested operation of the API they are utilizing. This can be a difficult problem for new social application developers who aren't versed in the areas of application security or user permission design. Again, we aim to provide both user authentication and request integrity through a simplified process.

1.4 Difficulties In Developing Social Applications

1.4.1 Reinventing the Wheel

When a developer designs and implements the basic parts of a social application, they are often forced to duplicate work that has been done many times before. This leads to more time being wasted on the basic parts of the social application, rather than the actual parts that make the it unique.

1.4.2 Technical Challenges

Everyone has good ideas, the difficult part is modeling the ideas correctly using a computer. Technical knowledge of web technologies, databases, and security is essential to developing a working version of a social application. Many new developers with good ideas don't have these skills, so their idea can never take off the way they want it to, and then the project never reaches fruition.

1.5 The Solution: The RAPDB

The RAPDB is a generic service that and attempts to provide practical solutions for the patterns described above and serves as a remedy to the technical challenges faced by new social application developers. It abstracts away much of

the complexity in developing back-end services for social application with easy to use http interfaces. By utilizing RAPDB's RESTful API, users can avoid dealing with database configuration and content persistence, back-end server logic and deployment, and the internals of authenticating and associating users by using a succinct set of generic APIs.

1.6 Contents of this Report

This report contains an examination of the defining components of social applications, as well as a high level design describing how a developer could set up a back-end RESTful service to support such an application. This description will cover server and database design, as well as a practical security design responsible for cryptographically securing the service. On top of the design of the service, this report will also cover how a front-end developer could utilize such a service to rapidly deploy a basic social application, along with practical testing methods for RESTful APIs. Lastly, the report will also contain complete API documentation of our example service, as well as a comparison of common frameworks that could be used to implement our back-end services.

2 Scenario Of System Use

This section illustrates how a simple app is created using the RAPDB service. It starts with the motivation of this application, and an overview its requirements documents. After reading the requirements documents the developer will be presented with a small check list to determine if their application concepts could benefit from using the RAPDB service. The section concludes with an example of the amount of work that is saved by using the RAPDB service.

2.1 FoodApp

This section covers the FoodApp implementation details, from comparing to the checklist provided below, to requirements documents, work saved from using the RAPDB, and finally a demonstration of the application.

2.1.1 Introduction to FoodApp

The FoodApp is an example application that makes use of the generic RAPDB service in order to create a customized application. The FoodApp serves as a social website through which users can create and share food recipes, which include step-by-step instructions concerning the recipe they are a part of. Users can create associations between each other (i.e. by sending Friend Requests), and can also specify what subsets of associated users should be able to view their content.

2.1.2 FoodApp RAPDB Checklist

In order to make sure that the RADPB service will be appropriate for an application, the following constraints must be followed:

1. Models must match the ones that are provided by the RAPDB.

2. The application must have a way to create HTTP calls using the JSON format.

3. TLS encryption must be supported by the application.

4. There are only 10 association levels supported, the model must fit with this requirement.

5. Meta data must be first POSTed to the service, followed by a PUT with the data. So any kind of fast data upload, download model would not work correctly.

6. Asynchronous communication works best with the RADPB, any applications requiring instant communication would not work with the system.

Given the open source nature of the RAPDB, the developer may edit any of the RAPDB files to fit the model more effectively. It should be noted that since FoodApp was developed side by side with RAPDB, it meets all six of the criteria listed above perfectly.

2.1.3 Requirements

The sections below describe the requirements documents produced for the FoodApp. The screens below were developed using a simple wireframe tool, and each section indicates which part of the application is being described.

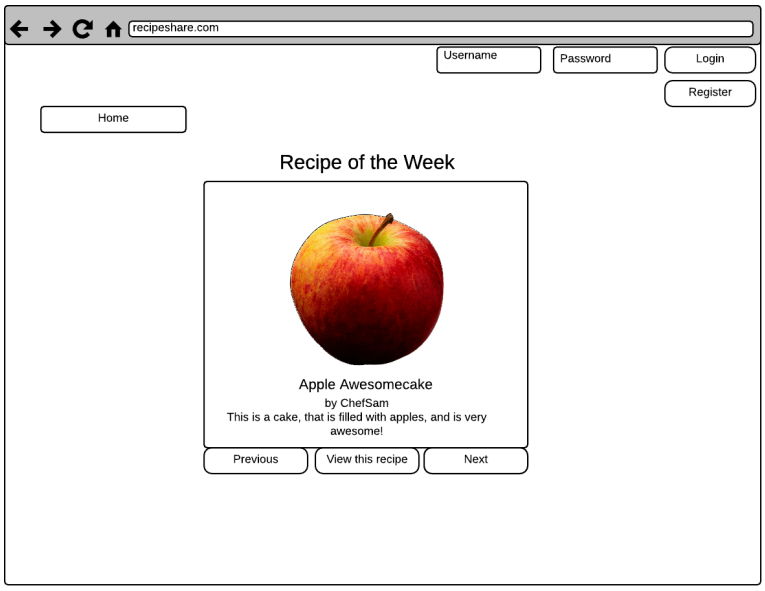

Figure 1: The home page of food app as a non-registered user

Home Page (As an unregistered user) The home page of the Food App is the starting point of any use case. It initially features three main parts, the Login Area, the Navigation Bar, and the Picture Viewer. Until the user authenticates, they are considered to be unregistered. See figure 1

Login Area This area is where the user authenticates with the RAPDB service. Once the user enters a valid user name and password, and then presses the "Login" button an AJAX call is made to the RAPDB service. In this particular instance, FoodApp is using the authentication API, which takes the following form: $POSThttp : // < rapdb_URL > /FoodApp/users/ < userTextBoxValue >, withrequestpayload =< passwordTextBoxVal >$ This submission happens over https using TLS. If the authentication call is successful, a token needed for future API calls is returned, and the user is considered authenticated. The views will then change to their 'logged in' versions.

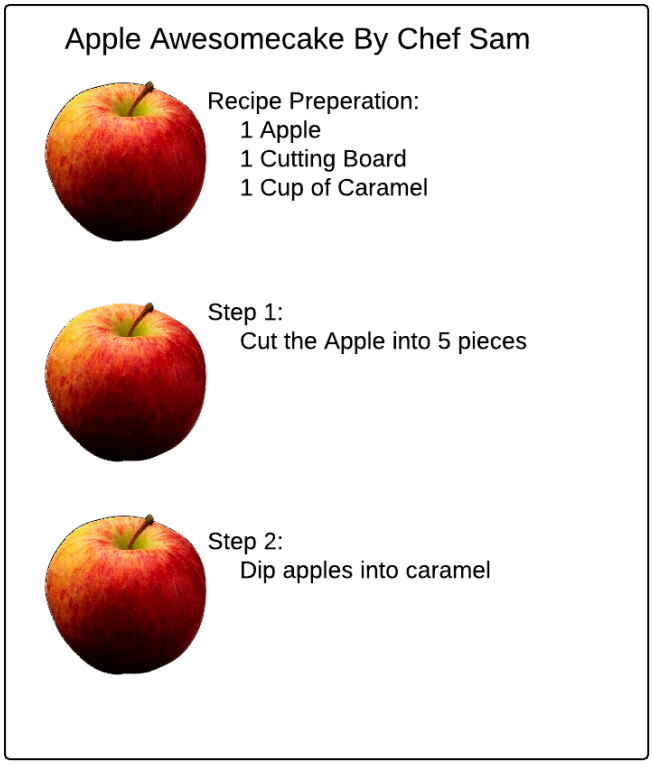

Figure 2: The popup after clicking 'view recipe'

Recipe Viewer The recipe viewer helps the user view the publicly available recipes. The recipes owner, the person who created the recipe, is displayed. The recipe's description, the label for the first image is displayed. The recipe's first picture is displayed as a way to catch another user's attention. When the user clicks the next button, the next recipe will be displayed. When the user clicks the previous button the previous recipe will be displayed. When the user clicks 'view this recipe' the Recipe Popup will display with the steps of the recipe.

Home Page (Authenticated) The friends list is how users of the FoodApp manage their associations with each other. Friends are listed in the top area of the list, and friends requests are listed on the bottom. When a user clicks the

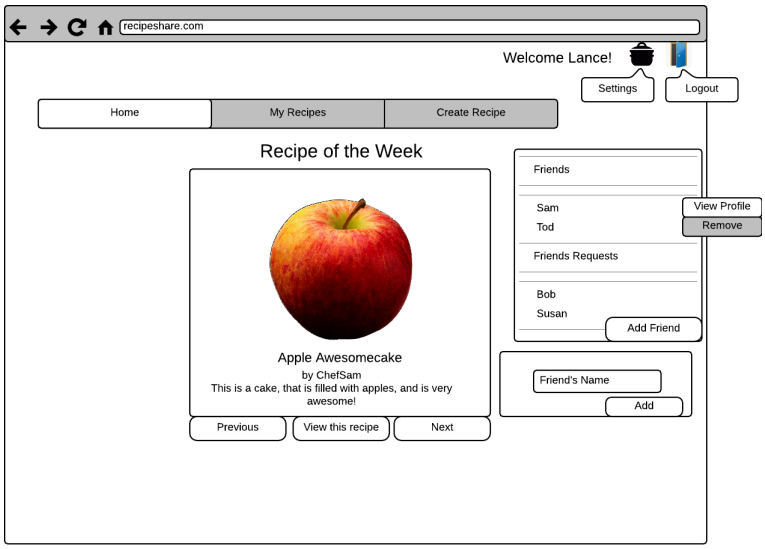

Figure 3: Adding a friend

'Add Friend' button the 'Add Friend' box will display with a text box and a button to complete the action. See Figure 3

Figure 4: Removing a friend

When a user right clicks a friend requests, they can accept it or deny it. The FoodApp will make a request with the RAPDB service to change the level of association to 1, which means the two users are friends, or remove the request. See figure 4

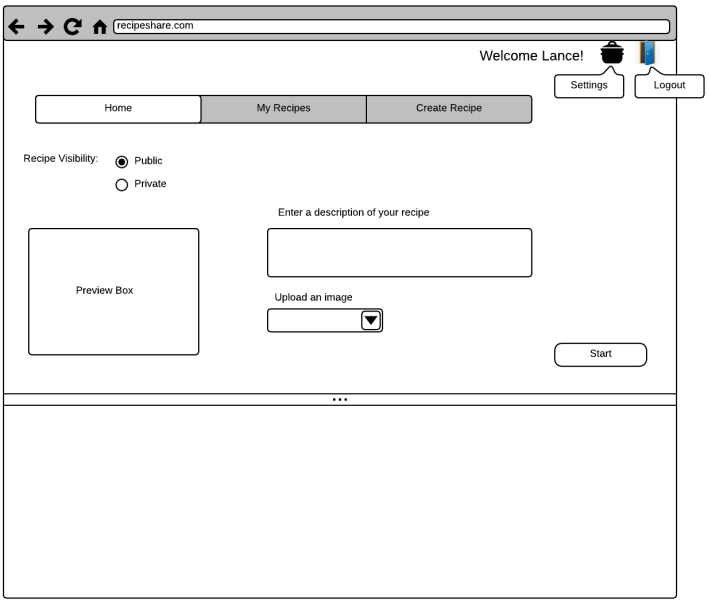

Figure 5: Start page to creating a recipe

Create Recipe Page (Logged In) Once the user authenticates, they are able to see the create recipe wizard. The create recipe page is how the user creates and save their recipes, which internally uses the RAPDB service to store their data and categorize it. Recipes fall under two categories, public recipes which are available to all users of FoodApp, and private recipes, which are only available to the friends of the user who creates the recipe.The user may select to submit the recipe as a private recipe, the default is set to public. The user is then required to enter in a top level description of the recipe, and an 'attention grabber' picture, that is displayed at the top of the recipe. Once the user clicks 'start' the application will then disable interactions on this level, and submit a POST to $http://FoodApp/data$, with all the information gathered from the UI. This will create the recipe data node itself, which is the parent to all the 'step' nodes. If a picture is added for any of the steps involved, the FoodApp will also be making use of another data API for uploading the user-supplied picture at PUT $http://FoodApp/data/ < userName > / < picID >$. See Figure 5

Adding Steps The first step template is automatically created after the recipe has been created, due to the fact that a recipe with no steps is not a recipe. The user must create at least one step. When the user fills in a description, and selects a picture for the step, they may either add more steps by clicking 'Add Step'. To remove a step, as long as it's not the first step, the user clicks 'remove step'. To save the recipe to the RAPDB service, the user clicks 'Save'. See figure 6.

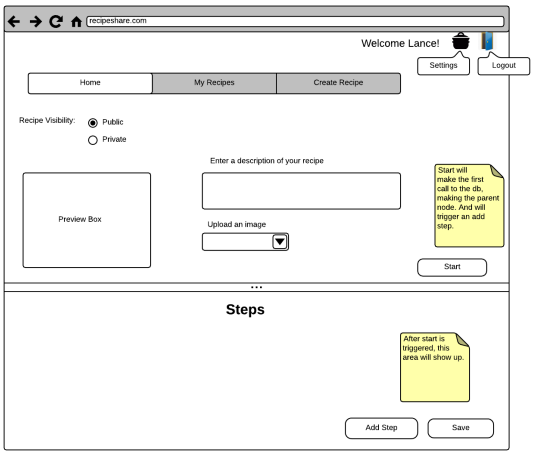

Figure 6: Adding a step to the recipe

2.2 FoodApp Demo

The FoodApp was created using HTML, CSS, and Javascript. The main Javascript library used was Jquery. This library allowed us to make AJAX calls to the RAPDB service. The screens shown in figures 7 through 13 are taken from the completed application itself.

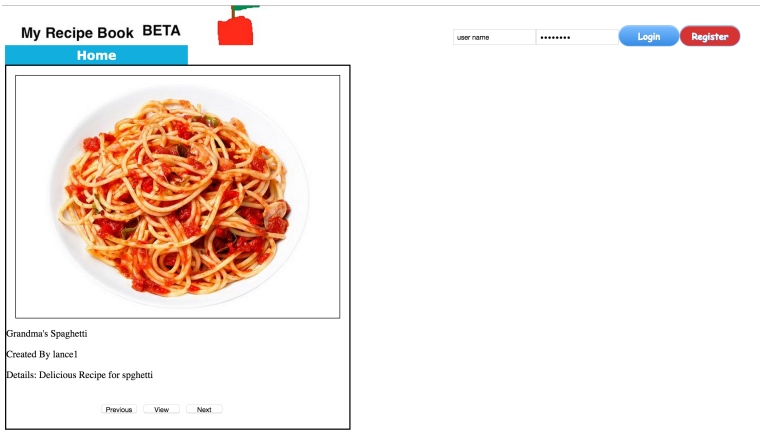

Figure 7: Home page as a un-registred user

Figure 8: Homepage as an authenticated user

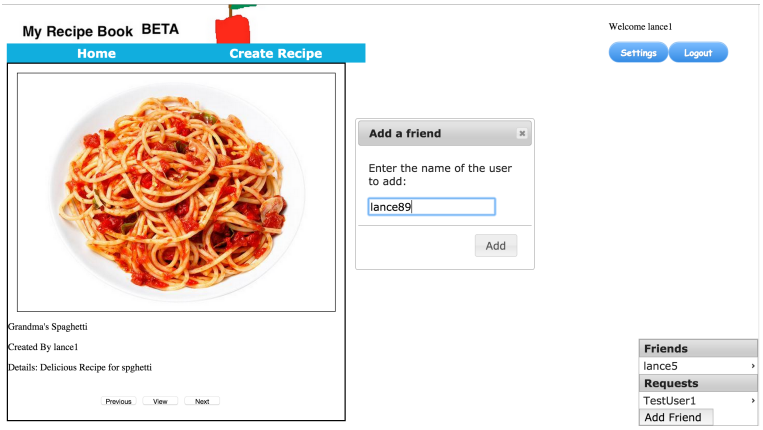

Figure 9: Sending a friend request

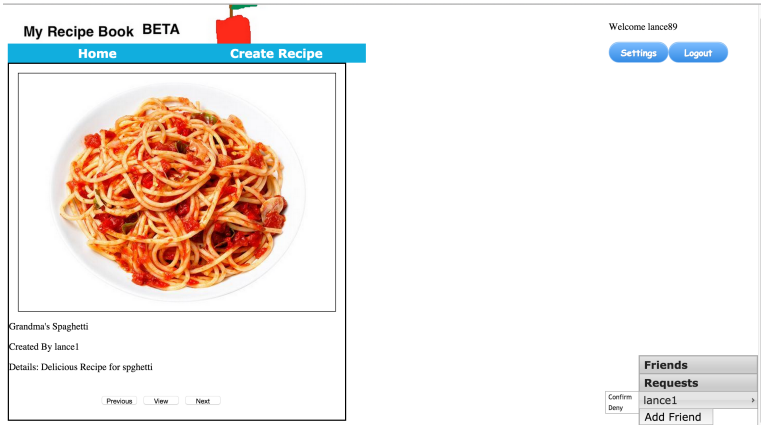

Figure 10: Accepting friend request, logged in as user who received request

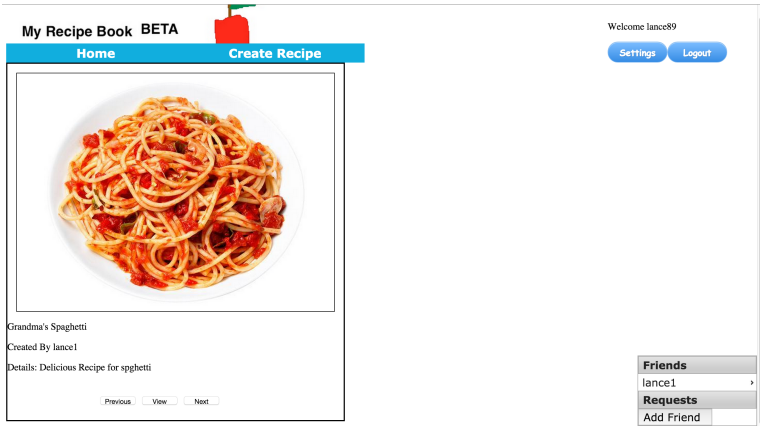

Figure 11: Friend request accepted, now both users will appear as friends

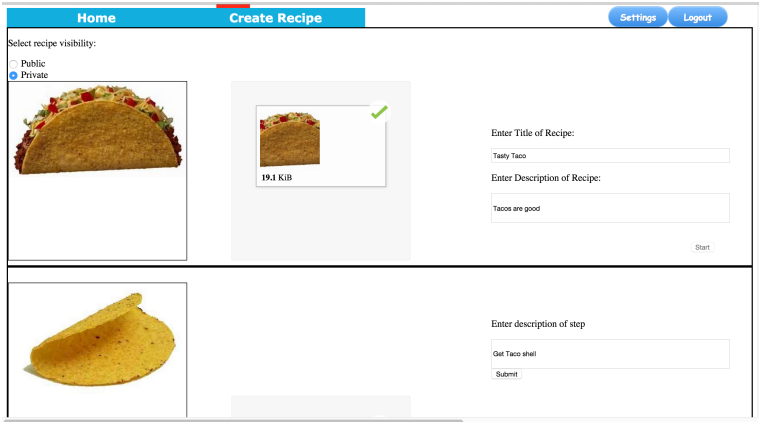

Figure 12: Create recipe page demonstration

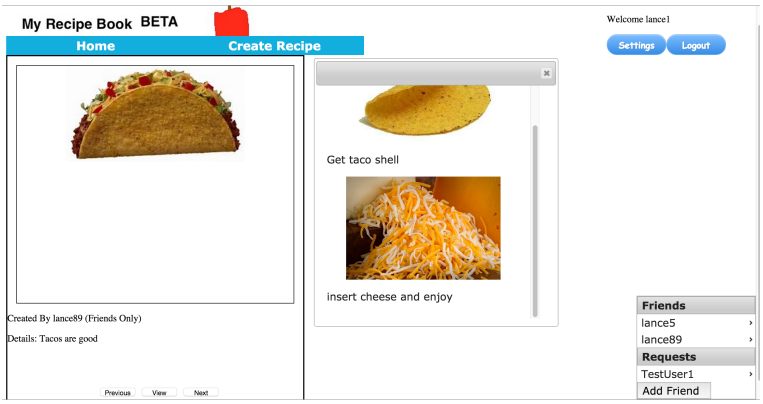

Figure 13: Viewing a private recipe on the main page

2.3 FoodApp Work Saved

In software engineering, the metric of lines of code may be used in measing the amount of work saved. In the FoodApp application there are 12 distinct AJAX calls to the RAPDB service which is almost all of the methods provided by the service. Each of these methods average about 100 lines of code each including the routing methods, and the database layer methods. By these measures, we can estimate that 1200 lines of code were saved by the creators of the FoodApp. Note that this does not include any of the time spent on researching an appropriate framework, setting up the database, setting up the server, or adding a layer of security on the project. See figure 14 for an example on the amount of code saved by using RADPB.

Figure 14: Code saved by using RAPDB

3 High Level Design

This section contains a discussion of the solutions to technical problems faced while developing the RAPDB. These problems range from designing a database that can hold generic types of data, a service that can take in this generic data, and securing the service from threats to message confidentiality and authenticity. The RAPDB server was also was designed using a multilayer software architecture approach as illustrated below in figure 15. This section will start from the bottom of the model, the database, and work its way to the top to the user interface developed as a demonstration for the service. See again 15

3.1 Database Layer

One of the technical challenges we faced during the design of the RAPDB was having to anticipate the needs of of application developers, the most prominent being the need for storage, sharing, and retrieval of ambiguous data from the RAPDB service.

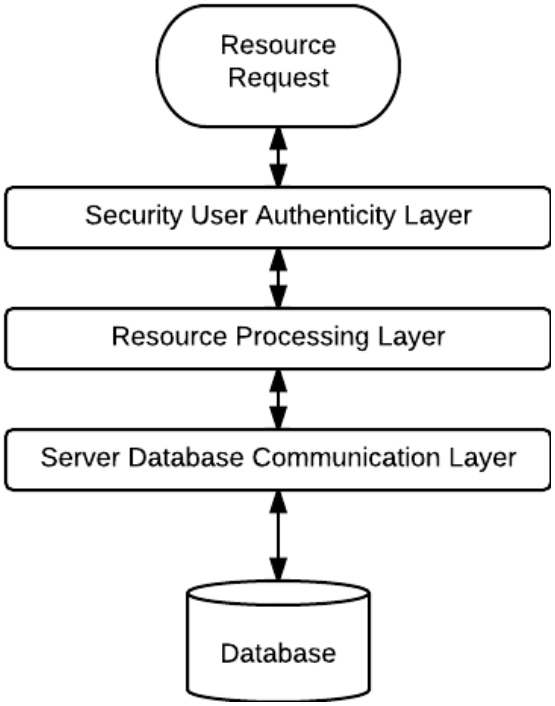

Figure 15: A high level diagram of the RAPDB project

3.1.1 Entities

While analyzing the common needs of social applications in regards to database design, the following entities and relations were discovered.

Application An application is an entity that is registered with the RAPDB service. Each application is identified by a unique name, and the date it was registered with the service. A one time "super" API key is generated for each application, and is used as proof of application ownership.

Users Users register with one application. A user's name must be unique to the specific application that they are registering with. Characteristics, such as the date the user registered with the application, are also commonly collected. Users also need a self-defined password, which needs to be of a reasonable length so as to deter any common means of account predation.

Security Tokens Security tokens are used in authenticating users. They contain a shared secret that is generated by the service, an expiration date, a created date, and the userID and application that the token is bound to.

Data Data in the context of this service is nothing more than an entry of generic data. It has a date that the user uploaded the data to the service. Data

recursively references itself, allowing nodes to be connected to each other.

3.1.2 Relationships

The following relationships were discovered while analyzing the requirements of the RAPDB. Users and Applications

Users and applications are related to each other with a registered date, and a last login date.

Association of Users Users may associate with other users based on a level. This level is an integer from 0 to 9, and its meaning may be defined by the application developer. Information such as when the association was created and the when the association was accepted are recorded.

Sharing of Data Amongst Users When data is initially uploaded it has the sharing level set to 0, which may be interpreted differently by different applications. This level may be changed.

Data relationship to Data Data is related to each other hierarchical means. The root node references null.

See figure 16 for an ER Diagram of the database

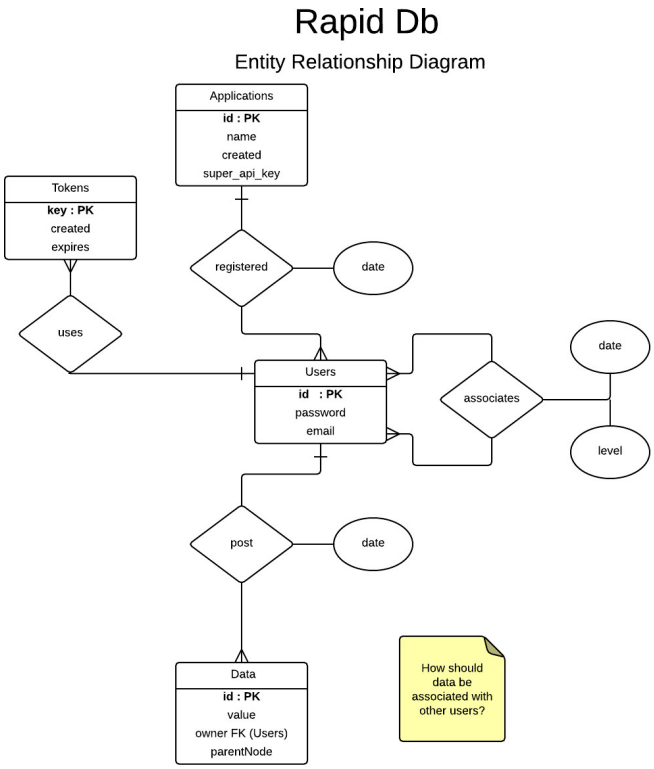

Figure 16: An ER Diagram of the RAPDB database

SQL DDL Below is the DDL to the MySQL database used in the RAPDB project.

```
CREATE TABLE Applications {
id INT, name VARCHAR(30),
created DATETIME,
masterAPIKey VARCHAR(40),
PRIMARY KEY (id) };

CREATE TABLE Users {
id VARCHAR(20),
password VARCHAR(20),
email VARCHAR(20),
dateRegistred DATETIME,
lastLogin DATETIME,
applicationID INT,
PRIMARY KEY (id),
FOREIGN KEY (applicationID)
REFERENCES Applications (id) };

CREATE TABLE Associations {
senderID VARCHAR(20),
receiverID VARCHAR(20), level INT,
dateRequested DATETIME, dateAccepted DATETIME,
PRIMARY KEY (senderID, receiverID, level),
FOREIGN KEY (senderID) REFERENCES Users (id),
FOREIGN KEY (receiverID) REFERENCES Users (id) };

CREATE TABLE Tokens { key VARCHAR(40),
created DATETIME, expires DATETIME,
userID VARCHAR(20), PRIMARY KEY (key),
FOREIGN KEY userID REFERENCES Users (id) };

CREATE TABLE Data ( id varchar(32) NOT NULL DEFAULT '0', created DATETIME,
value longblob, owner VARCHAR(20), applicationName VARCHAR(30),
visibilityLevel int(11), description varchar(512),
contentUrl varchar(128), fileType varchar(16),
name varchar(30), parentNode varchar(32) DEFAULT '0',
PRIMARY KEY (id), FOREIGN KEY (owner) REFERENCES Users (id),
FOREIGN KEY (parentNode) REFERENCES Data (id),
FOREIGN KEY (applicationName) REFERENCES Users(applicationName) );
```

3.2 Server Database Communication Layer

The next layer in the RAPDB architecture is the server database communication layer. This layer wraps up database CRUD operations into methods used by the resource processing layer. See figure 17 for the UML of this layer

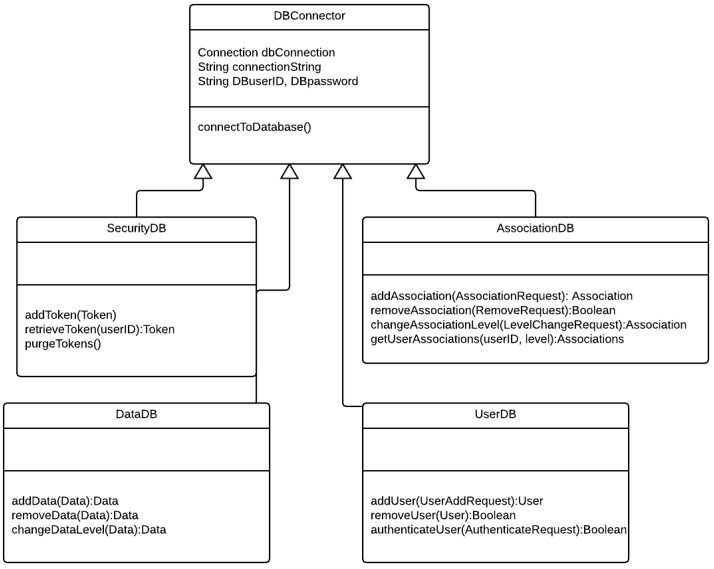

Figure 17: UML of the server database communication layer

3.3 Resource Processing Layer

The resource processing layer is responsible for routing resource requests to appropriate methods. It also extracts the information sent in the parameters, and the payload and creates the appropriate model, and sends that model to the database communication layer for a database CRUD operation. There are four resources made available to the application developer: application, user, associations, and data. Details of these four resources are described in the sections below

Application Applications are containers for users. An application must first register with the service before users can begin registering with it.

Data The rapdb service allows for uploading of arbitrary data. The first step in the process is to "POST" a meta data object, specifying what type of data to upload. From this call a user will receive a url to which they then would put their actual raw data to. By specifying the file type, we can then serve up this raw data in ways that a browser will know how to embed (by setting mime-types appropriately). The data layer also has several APIs for querying for data of a specific user. These queries return URIs which can be used to retrieve the users

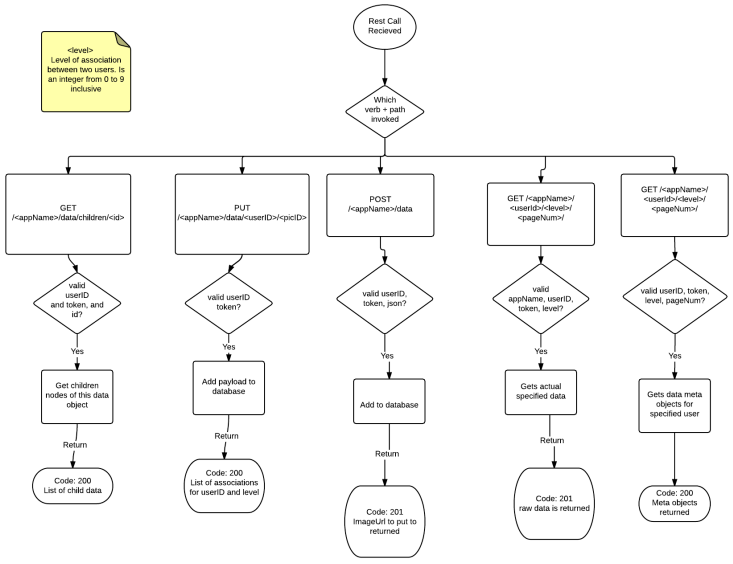

Figure 18: Flowchart of the logic for the data resource

raw data, or be used to embed them directly into a web page's DOM (if the data represents an image, for example).

See Figure 18 for logic on adding data to the service.

User Users register with applications. The /user/ resource is responsible for managing users, as well as user login. See attached document labeled "API Guide" for more details.

Association The association resource is used for the management of associations between two users. An association request will have the user who is requesting the association, and the receiver of the request. This is so that if the application wished to see who made the request, the user id of who sent the association request is preserved. The level of the association is also maintained for use by the application developer. A created time is also recorded by this layer. For more details see figure 19 for the associations flow chart.

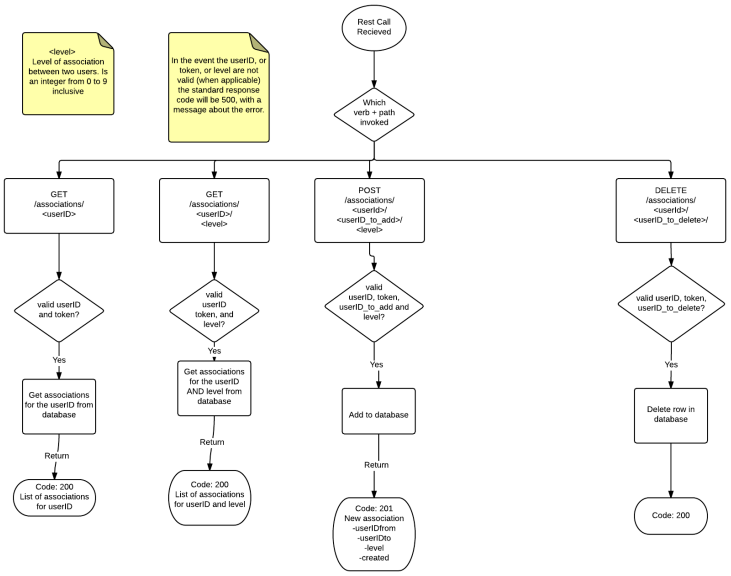

Figure 19: Flowchart of the logic for the Associations resource

3.4 API Guide

See the companion report titled "API Guide" for a list of public API methods avalible.

3.5 Security / User Authenticity Layer

The user authenticating components of this project, as well as the overall security components, are discussed in the paper linked below. This paper assumes an understanding of introductory cryptography principles, and discusses how a TLS initiated HTTPS connection and HMACs are used in combination to provide both authenticity and confidentiality for our service. See the attached report titled "Securing Restful API Services". This report was produced for an advanced course on computer cryptography, taught by Professor Zachary Peterson at Cal Poly.

4 Testing

With the development of any software comes the need for testing or validation of it. In developing a RESTful service, we found that a sure-fire way of quickly producing HTTP requests was highly critical in being able to trigger and test backend APIs, and ultimately in being able to create APIs in a timely manner. Through a combination of JUnit testing and testing via specialized RESTful testing tools, we are able to quickly and efficiently obtain validation for our services as we developed them.

4.1 JUNIT

To simplify testing of our less involved APIs, we utilized the JUnit framework, which automated what would have otherwise required manual testing, and allowed for efficient regression testing. This approach also allowed us to create a testing environment that was decoupled from the actual implementation, which helped to achieve uniformity between our various implementations, since they were responding to the same test cases. From establishing basic connectivity with the server, to validating request responses and their payloads, JUnit proved to be an invaluable tool in regards to this project. See figure 20 for the Junit tests we created.

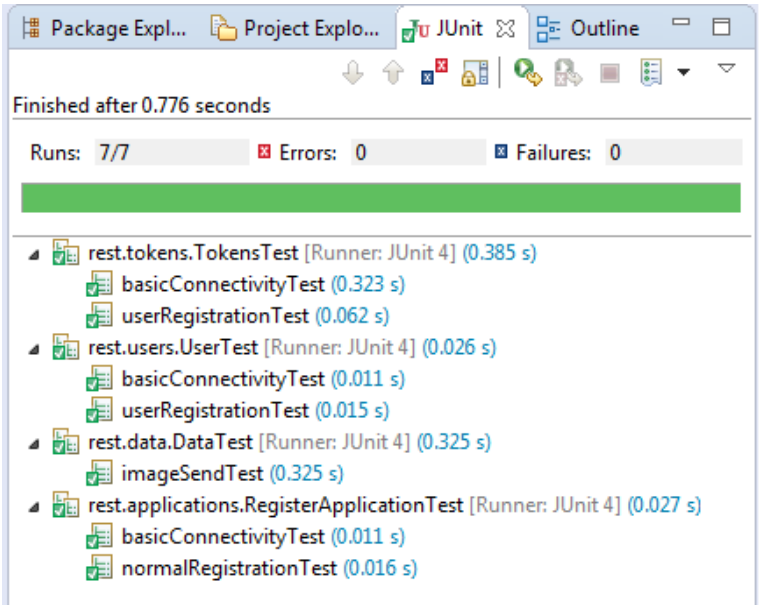

Figure 20: JUnit tests in Eclipse

4.2 Advanced Rest Client

For more involved test cases, we also utilized some more specialized tools created specificly to deal with testing RESTful services. The Advanced Rest Client, for example, is a chrome plugin that has an extensive collection of tools in regards to submitting HTTP requests. With the Advanced Rest Client, not only can you specify the HTTP verb and URI to access, but you can also specify specific query parameters, HTTP headers, and custom payloads of any type. This tool allows for the storing of requests, as well as the ability to replay requests in succession, which is a highly desirable ability when you need instant validation of an API as you iteratively develop it. The Rest Client Tool, which comes in the form of an eclipse plugin, also serves a similar purpose. If you are already developing your server/service in eclipse, this tool is highly valuable in that you can get instant verification of APIs, or trigger your backend services for quick testing/debugging purposes without having to switch contexts to a more heavy weight form of testing. Of course, these requests can ultimately be exported to a more permanent JUnit test, but they serve their purpose especially well for initial testing. See figure 21 for the Advanced Rest Client interface.

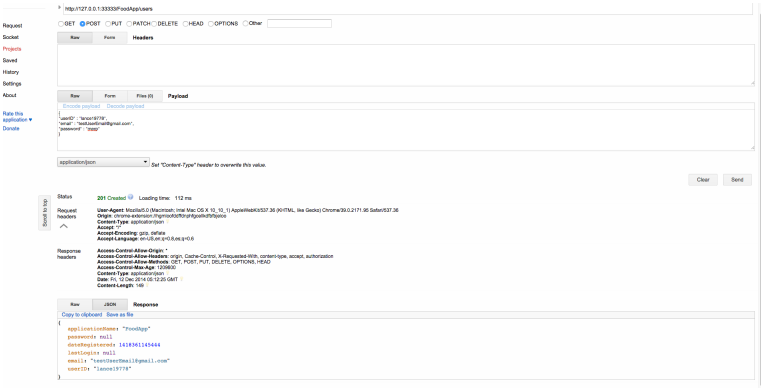

Figure 21: The Advanced Client Interface

4.3 FoodApp Acceptance Testing

This section describes the acceptance testing of the FoodApp as seen by a user of the application. The following tests were created from the requirements specified in the above section. These tests cover normal usage of the FoodApp, from using features such as the recipe viewer, authenticating with the system, adding an association, removing association, and creating a new recipe.

Number	User Inputs	Expected Output	Results
1	Go to FoodApp Homepage by entering URL	Picture viewer and all elements should be present	Pass
2	Click the view button on the recipe viewer	The steps for the current recipe should be displayed	Pass
3	Click the next button on the recipe viewer	The next recipe should be displayed	Pass
4	Click the previous button on the recipe viewer	The previous recipe should be displayed	Pass
5	Enter "lance1" in the user name text field and "taco" for the password, then select the "login" button	The system should change to the logged in view, the settings button should appear. The friends list should also appear. A create recipe tab should be shown.	Pass
6	Click add friend, and enter in a valid user name	That user should now be in the requests section	Pass
7	Click the user name in the requests section and deny the friend request	The name should dissapear	Pass
8	Re-add the user as a friend, login as that user	The previous user name should appear in the requests section	Pass
8	Click the user name in the requests section, and click add	The name should move to friends	Pass
9	Click the user name in the friends section and click remove friend	The name should dissapear	Pass
10	Click the create recipe tab	The create recipe wizard should load	Pass
11	Fill out the recipe information, and upload a photo, then click submit	A green check will appear next to the image, and a new step will load	Pass
12	Go back to the homepage and find the new recipe in the recipe viewer	Recipe should be there with any steps created in previous testing step	Pass

5 Comparison of Different Frameworks

A companion report entitled "Choosing RESTful Framework For Web Applications" outlines the way in which we decided on the development framework to used for RABDB. This paper was produced for an independent study course directed by professor Gene Fisher at Cal Poly.

6 Conclusions

6.1 Does it Work?

Yes, it works in the context of fast development. Our service may not be suitable for a production level environment - however it would be good for applications that require a fast prototyping development cycle. It would also be useful to

anyone trying to spin up some type of application prototype, either by front-end developers, people in the starting stages of a start-up, or by students who are seeking a generic platform to manage their users for a smaller sized application. If someone wished to turn their initial prototype into a fully scalable application, they would need to take additional steps to fine tune their server's performance. For instance, they would probably want to apply some denormalization and indexing to their database to get better response times from the underlying database. The deployment server would also need some fine tuning, like limiting the amount of concurrent connections or setting certain properties that are necessary in anticipation of more users. Our service currently uses default framework settings, so while it might be sufficient for moderately sized user bases, it might not perform at its best compared to a fully configured server.

6.2 Does it reduce development time?

Yes, it reduces development time for a simple prototype of the idea. The need to get their idea into the hands of testers to see if their idea is actually viable is something all aspiring developers have encountered. With this service, developers can do just that, so they can focus on the user-facing components of their application, which often act as a litmus test for whether an application will sink or swim in the real world.

6.3 Implementation Decisions We Regret

Using a relational database - a NOSQL solution such as MONGO could have potentially simplified all of our database operations in terms of developer effort. Since we didn't have any particularly complex queries, we probably could have gotten away with a NOSQL solution. As with all new technologies, however, there would have been a learning curve associated with it, so the tradeoffs are difficult to compare, given that this service would likely never be used at a production level. A NOSQL solution may have also fit better with the goals of our project, which was to simply and rapidly set up a back-end service.

Not following true REST design methodologies - To put it succinctly, RESTful purity is difficult to maintain. While RESTful design calls for statelessness, our service was forced to deviate from this in order to add an increased amount of cryptographic security. By storing our user session tokens in a database (in order to validate requests as they come in), we are technically storing or creating some form of state, which RESTful purists might not particularly care for. The trade off, however, is that we were able to incorporate a high level of both confidentiality and authenticity of all of our user requests by keeping track of these unique tokens.