

Spring 2013

# Development of an Api As a Solution to Data Insularity Amongst Dietary-Need Applications

Raphael LaPuma  
*Regis University*

Follow this and additional works at: <https://epublications.regis.edu/theses>



Part of the [Computer Sciences Commons](#)

---

## Recommended Citation

LaPuma, Raphael, "Development of an Api As a Solution to Data Insularity Amongst Dietary-Need Applications" (2013). *All Regis University Theses*. 242.  
<https://epublications.regis.edu/theses/242>

This Thesis - Open Access is brought to you for free and open access by ePublications at Regis University. It has been accepted for inclusion in All Regis University Theses by an authorized administrator of ePublications at Regis University. For more information, please contact [epublications@regis.edu](mailto:epublications@regis.edu).

**Regis University**  
College for Professional Studies Graduate Programs  
**Final Project/Thesis**

**Disclaimer**

Use of the materials available in the Regis University Thesis Collection ("Collection") is limited and restricted to those users who agree to comply with the following terms of use. Regis University reserves the right to deny access to the Collection to any person who violates these terms of use or who seeks to or does alter, avoid or supersede the functional conditions, restrictions and limitations of the Collection.

The site may be used only for lawful purposes. The user is solely responsible for knowing and adhering to any and all applicable laws, rules, and regulations relating or pertaining to use of the Collection.

All content in this Collection is owned by and subject to the exclusive control of Regis University and the authors of the materials. It is available only for research purposes and may not be used in violation of copyright laws or for unlawful purposes. The materials may not be downloaded in whole or in part without permission of the copyright holder or as otherwise authorized in the "fair use" standards of the U.S. copyright laws and regulations.

**DEVELOPMENT OF AN API AS A SOLUTION TO DATA INSULARITY AMONGST  
DIETARY-NEED APPLICATIONS**

A THESIS

SUBMITTED ON 9 OF APRIL, 2013

TO THE DEPARTMENT OF INFORMATION TECHNOLOGY  
OF THE SCHOOL OF COMPUTER & INFORMATION SCIENCES  
OF REGIS UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS OF MASTER OF SCIENCE IN  
SOFTWARE ENGINEERING AND DATABASE TECHNOLOGIES

BY



---

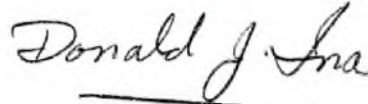
Raphael LaPuma

APPROVALS



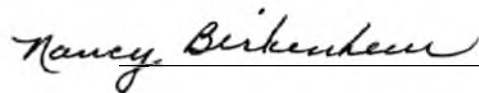
---

Rob Sjodin Advisor



---

Donald J. Ina Ranked Faculty



---

Nancy Birkenheuer Ranked Faculty

## Table of Contents

<b>Table of Contents .....</b>	<b>2</b>
<b>List of Figures.....</b>	<b>8</b>
<b>Part 1: Introduction .....</b>	<b>9</b>
<b>1.1 Introduction to Community-Based Information and Finder-Review Applications</b> .....	<b>9</b>
1.1.1 Dietary need finder-review applications.....	9
<b>1.2 Problem Statement: Data Insularity.....</b>	<b>10</b>
<b>1.3 Thesis Statement.....</b>	<b>11</b>
<b>1.4: Review of Data Insularity in Dietary Need Finder-Review Applications .....</b>	<b>11</b>
1.4.1 Data Models.....	11
1.4.2 Reviews of Applications.....	12
1.4.2.1: Gluten Free Registry application.....	13
1.4.2.1.1 the data model of the application.....	14
1.4.2.1.1 data consumption in application.....	15
1.4.2.2 Happy Cow (Vegetarian).....	16
1.4.2.2 the data model of the application.....	17
1.4.2.2 data consumption in application.....	19
1.4.2.3 Dish Freely.....	19
1.4.2.3.1 the data model of the application.....	22
1.4.2.3.2 data consumption in application.....	23
1.4.5 Summary of Current Applications: Data Insularity.....	23
<b>1.5 Goals of Project .....</b>	<b>24</b>
1.5.1 Criteria of Solution.....	24

1.5.1.1 Solution must act as central repository that will allow clients to consume and provide data.....	25
1.5.1.2 Solution must impose a standard model upon all data that is consumed or provided from the repository.....	25
1.5.1.3 Solution must have a data model that will meet the needs of client applications as well as end users.....	26
<b>1.6 Value of this Project .....</b>	<b>26</b>
<b>1.7 Ethical and Social Impact of Project.....</b>	<b>26</b>
<b>Part 2: Review of Literature and Research .....</b>	<b>28</b>
<b>2.0.1 Review of Possible Solutions .....</b>	<b>28</b>
<b>2.1 Review of API as a Solution.....</b>	<b>29</b>
2.1.1 An Introduction to APIs.....	29
2.1.2 Public vs. Private APIs.....	30
2.1.3 An Introduction to RESTful Web services .....	31
2.1.3.2 REST: pragmatic vs. dogmatic approach .....	32
<b>2.2 Review of Existing Solutions (APIs) .....</b>	<b>34</b>
2.2.0.1 Explanation of API characteristics observed.....	35
2.2.0.2 Explanation of why these APIs were chosen .....	35
<b>2.2.1: Yelp API.....</b>	<b>36</b>
2.2.1.1 Overview of API.....	37
2.2.1.2 Data Model of API .....	37
2.2.1.3 Assessment of use of API for dietary need applications .....	39
<b>2.2.2: FoodSpotting API.....</b>	<b>40</b>
2.2.2.1 Overview of the API .....	42
2.2.2.2 Data Model of API .....	43
2.2.2.3 Assessment of Use of API For Dietary Need Applications.....	45

	4
2.2.3: Food Genius API .....	45
2.2.3.1 Overview of the API .....	47
2.2.3.2 Data Model of the API.....	47
2.2.3.3 Assessment of Use of API for Dietary Need Applications.....	48
<b>2.3 Why None of These Solutions Fit the Requirements.....</b>	<b>48</b>
2.3.1 The need for a new solution.....	51
<b>Part 3: Project Methodology .....</b>	<b>52</b>
<b>3.1 Research Method Chosen: Constructivist Epistemology.....</b>	<b>52</b>
3.1.1 A RESTful API as the Appropriate Solution .....	52
<b>3.2 Project Requirements.....</b>	<b>53</b>
3.2.1 API Solution Requirements.....	53
<b>3.3 Requirements in Depth.....</b>	<b>54</b>
3.3.1 Requirement: Allow Clients to Consume and Provide Data .....	54
3.3.2 Requirement: Scale to current and future dietary needs .....	54
3.3.3 Requirement: Allow for the Two Levels of Data (“Restaurant” and “Menu Item”) .....	55
3.3.4 Developer Support.....	55
3.3.5 API Policies of Usage.....	55
Data Usage Policies: .....	55
Security Policies: .....	56
Quotas and Limitations:.....	56
User-Submitted Data:.....	56
<b>3.4 Methodology: Implementation of Project.....</b>	<b>56</b>
3.4.1 Process Model Chosen .....	56
<b>3.5 API Component Design and Implementation .....</b>	<b>58</b>

3.5.1 Documentation Website.....	58
3.5.2 Sample Client Application .....	59
3.5.3 Data Model and Central Data Repository .....	60
Place Resource .....	60
Item Resource .....	61
Sighting Resource .....	62
Report Resource.....	62
Explanation of Data Model.....	63
Implementation of Data Model.....	63
Important Additional Entities .....	64
Categories.....	65
Permissions .....	67
Application, User, and API Access Log.....	68
3.5.4 RESTful API.....	69
Access to the API.....	70
Requests to the API.....	71
Responses from the API.....	72
Metadata .....	72
Error Message .....	73
Pagination .....	73
ID.....	74
Data.....	74
3.5.5 Implementation of API.....	76
<b>3.6 Process Flow .....</b>	<b>77</b>
<b>3.7 Summary .....</b>	<b>78</b>
<b>Part 4: Analysis of Results .....</b>	<b>79</b>

<b>4.1 How the API meets the requirements for a solution.....</b>	<b>79</b>
4.1.1 Data at Two Levels .....	79
4.1.2 Allow Clients to Consume and Provide Data.....	80
4.1.3 Client Support and Documentation.....	80
4.1.4 Scale to Current and Future Dietary Needs .....	81
4.1.5 API: Equipped to Enforce Policies of Usage.....	82
4.1.5.1 Security.....	82
Authentication.....	82
Authorization.....	82
Validation .....	83
Character Escaping.....	83
4.1.5.2 Usage Limitations and User-submitted Data .....	84
<b>4.2 How the API Is A Solution to the Problem of Data Insularity in This Field of Applications .....</b>	<b>85</b>
<b>4.3 Project History .....</b>	<b>86</b>
4.3.1 How the Project Began .....	86
4.3.2 How the Project Was Managed .....	86
4.3.3 Project Milestones .....	87
4.3.4 Changes to the Project Plan.....	87
4.3.5 Project Timeline .....	87
Phase 1: December 2011 – April 2012 .....	87
Phase 2: May 2012 – August 2012 .....	87
Phase 3: September 2012 – January 2013.....	87
Phase 4: February 2013 – April 2013 .....	88
4.3.6 Ethical and Social Impact of the Project.....	88
<b>Part 5: Conclusions and Lessons Learned .....</b>	<b>89</b>



<b>5.1 Conclusions .....</b>	<b>89</b>
5.1.1 The Problem of Data Insularity.....	89
5.1.2 Data Insularity Causes Incomplete and Fragmented Data.....	89
5.1.3 A Prototype Solution Is Needed.....	89
5.1.4 Prototype RESTful API as a Solution .....	90
5.1.5 The Project Successfully Satisfies These Requirements .....	90
<b>5.2 Summary of Contributions.....</b>	<b>90</b>
5.2.1 The Documentation of a Prototype Solution to Data Insularity .....	90
5.2.2 An Analysis of Potential Solutions to the Problem of Data Insularity.....	90
5.2.3 A Brief Review of Applications in the Field that Represent the Problem of Data Insularity.....	91
<b>5.3 Lessons Learned .....</b>	<b>91</b>
<b>5.4 Recommendations / Future Research .....</b>	<b>92</b>
<b>Sources .....</b>	<b>93</b>

## List of Figures

Figure 1: Table Comparing Possible Solution APIs .....	49
Figure 2: Flowchart of Spiral Prototyping Software Development Lifecycle Used.....	58
Figure 3: ERD of Solution Data Model .....	64
Figure 4: ERD of Item Category Implementation .....	65
Figure 5: ERD of Place Category Implementation .....	66
Figure 6: ERD of Permissions Resource .....	67
Figure 7: ERD of User and Application Authentication.....	69
Figure 8: Flowchart of Application Authentication.....	71
Figure 9: Flowchart of API Access.....	78

## **Part 1: Introduction**

### **1.1 Introduction to Community-Based Information and Finder-Review Applications**

With the rise of Web 2.0 technologies and social media, consumer applications are being created for many industries. One particular type of consumer application that has become popular is the user-generated review application, which allows the user to submit a review to a community of other users who may then benefit from reading this review. The travel and tourism industry, business location services, and the food and restaurant industries are just a few examples of areas for which many applications have been developed that utilize user-generated reviews. The restaurant industry in particular has seen many applications that allow users to submit reviews to a community of users, and then allow users to find a restaurant at which to eat based on reading other users' reviews.<sup>1</sup>

#### **1.1.1 Dietary need finder-review applications**

While finder-review applications may be used for many entertainment-related and informational purposes, one important use of finder-review applications in the restaurant industry is assisting people with special dietary needs to find appropriate places to go to dine, and appropriate menu items to eat. Today, many people are choosing to avoid certain types of food, and some people must avoid these types of food altogether. Three of the most popular types of dietary need applications are for gluten-free, vegetarian, or vegan diets. There are many finder-review applications that have been developed to assist

---

<sup>1</sup> The author will refer to these types of applications as “finder-review” applications throughout the paper.

people with different dietary needs.<sup>2</sup> By focusing on a specific dietary need, users of these applications are able to find and share restaurants and dishes for their needs. The advantage of using this type of application is that users have a way to find ideas of where to go and what types of food they can expect there that will fit their dietary need. However, the issue with this type of application is that typically there are many competing applications with only very limited and inconsistent data.

One of the biggest challenges that any finder-review application faces is obtaining enough data to become useful to end-users. Without a significant amount of data, users will find little value in the application. Paradoxically, without being able to attract a large base of users, the application cannot obtain enough user data to become useful. Furthermore, because each application has its own small set of data, users will often need to use multiple dietary need finder-review applications in order to obtain a more complete sense of the data that is available.

### **1.2 Problem Statement: Data Insularity**

When multiple competing applications in the same field use only their own sets of user-generated review data and do not collaborate with other applications, data is used in isolation. This factor, which the author will refer to as “data insularity”, segregates each application’s community of users from other applications’ communities, thus creating a limited and fragmented set of data, which discourages the creation of new applications in the industry, and adversely affects the end user.

---

<sup>2</sup> Throughout this paper, the term “dietary needs” will be used in particular to refer to the needs of gluten-free, vegetarian, or vegan diets.

### 1.3 Thesis Statement

This study will propose a prototype API for applications in the restaurant industry as a solution to the limited and insular use of data amongst dietary need applications.

### 1.4: Review of Data Insularity in Dietary Need Finder-Review Applications

In order to better understand the problem of data insularity amongst dietary need finder-review applications, a brief observational study will be conducted on several of the most popular applications in this area. Characteristics of the application will be examined such as its data model, its policies on cross-application data collaboration, its policies on data caching, and the existence of some method of data collaboration, such as a public API.

#### 1.4.1 Data Models

In this paper, the term “data model” is used to refer to the organization of key data entities. This term is borrowed from Peter Rob and Carlos Coronel:

*“A data model is a relatively simple representation... of more complex real-world data structures. In general terms, a model is an abstraction of a more complex real-world object or event.” (Rob, 2009)*

This concept of a data model must be differentiated from the application’s underlying “database model”, which is the term that Rob and Coronel use to refer to the actual *implementation* of a data model to a specific database system. In other words, the author does not attempt to understand the database models—how the data is actually stored on these applications’ database systems—but rather an abstraction of how the real world data structures that the data itself represents is conceptually organized in that application. Furthermore, in a data model there exists entities, attributes, and relationships. Entities

are the things about which data is collected (person, place, thing, or event), an attribute is a characteristic of an entity, and a relationship describes the association between entities (Rob, 2009).

In most finder-review applications data is structured into two main entities: Place and Review. The Place entity represents the establishment that is being reviewed, and the Review entity represents a person's individual review of this establishment. There is a one-to-many relationship between the Place (1) and the Reviews (many).

#### **1.4.2 Reviews of Applications**

While there are numerous finder-review applications in the restaurant industry for special dietary needs, the author has chosen three applications from this genre to explore the typical functionality. Both the "Gluten Free Registry" website and application and the "Happy Cow" website and application were chosen because of their popularity and extensive data compared to other applications in this genre that adhere to the restaurant model. The "Dish Freely" iPhone application was chosen because of its unique position as the only application in this genre that uses the "menu item model". What follows is a brief analysis of these three applications, based upon certain characteristics, such as: the type of application it is; the type of data that is available to users; the type of data contributed by users; the method of users submitting data; the types of parameters (if any) the application's search capability is based upon (if any search functionality exists); the method through which users register to be able to utilize the application; who owns the data that is submitted to the application (the user that submitted it, or the application's proprietor); and its data collaboration policies or API status (if it exists).

#### ***1.4.2.1: Gluten Free Registry application***

The Gluten Free Registry is a popular website and application that helps users find restaurants that offer gluten free options and reviews for those restaurants from other Gluten Free Registry users. Below is a summarization of a brief observational study on this application:

- Type of Application:
  - Website and mobile application
- Type of data:
  - Gluten-free restaurants
- User-contributed data:
  - Users suggest restaurants and review a restaurant experience.
- Method of submitting data:
  - In both the Web and mobile application, the user submits an electronic form.
- Method of filtering data:
  - Users may flag a review as inappropriate. Users may report a restaurant as closed.
- Search parameters:
  - Users may search by location and proximity. Users may also filter searches to “chain” establishments and restaurants that will only accommodate gluten-free dishes (not specializing in these dishes)
- Registration:
  - None: anyone can suggest a restaurant and submit a review

- Extras:
  - Optional tags for popular items such as pizza, bread and pasta. Optional link to restaurant’s menu. Optional link to restaurant’s website.
- Data collaboration or API:
  - Data collaboration is not facilitated, and there is no public API.
- Data ownership:
  - The Gluten Free Registry asserts ownership over all user-submitted content in their Terms and Conditions.
- Restrictions:
  - There are strict policies against caching data or reusing data, and crawling the website.

#### *1.4.2.1.1 the data model of the application*

The Gluten Free Registry Website falls into the category of the “restaurant-review” data model, in which the application consists of restaurant listings and reviews of those restaurants. In this model, the data available to users can be separated into a restaurant object, and a review object. A restaurant may have many reviews, but each review is associated to one restaurant. The following data model represents a conceptual collection of objects and properties that can be inferred after an exhaustive summary of all available elements and their properties available to end users of the Web application.

#### *Restaurant (object)*

- Restaurant name (*property*)
- Date added (*property*)
- Location (*property*)



- phone number (*property*)
- Restaurant description (*property*)
- Online menu URL (*property*)
- Restaurant website URL (*property*)
- Tags:
  - e.g. Pizza, Pasta, Bread
- Overall Rating (0-5) (*property*)

#### Review (*object*)

- Date posted (*property*)
- Name of reviewer (*property*)
- Rating (0-5) (*property*)
- Text review (*property*)
- Number of likes / dislikes (number of people who found this review useful or not useful) (*property*)

Based on this data model, it can be seen that users learn of gluten-free items through reading the reviews on a restaurant.

#### *1.4.2.1.1 data consumption in application*

All data must be obtained through the Gluten Free Registry's website or mobile application. There is no public API available, nor is there an option for companies to act as partners to use their data. In its Terms and Conditions, the Gluten Free Registry asserts ownership of all data provided by users, and explicitly prohibits caching data in any way.

### ***1.4.2.2 Happy Cow (Vegetarian)***

Happy Cow is a Web and mobile finder-review application for vegetarian and vegan restaurants (both exclusively vegan/vegetarian, and vegan/vegetarian-friendly restaurants). Similar to the Gluten Free Registry application, Happy Cow allows users to submit and update restaurant listings, and submit reviews of these listed restaurants.

- Type of application:
  - Website and mobile application
- Type of data:
  - Vegetarian and vegan friendly and exclusive restaurants.
- User-contributed data:
  - Users suggest restaurants, submit reviews of an experience at a restaurant, and respond to other users' reviews.
- Method of adding data:
  - In both the Web and mobile application, the user submits a web form.
- Method of filtering data:
  - Users can flag a review as inappropriate, and they may comment on reviews.
- Search Parameters:
  - Users may search for relevant reviews based on location, keywords, and diet type (vegetarian or vegan).
- Registration:
  - Registration is necessary in order to submit a restaurant or review
- Data collaboration or API:

- Happy Cow does not offer any type of data collaboration or API.
- Data ownership:
  - In the Terms and Conditions, Happy Cow asserts ownership of all user-submitted data.
- Restrictions:
  - In the Terms and Conditions, Happy Cow asserts strict policies against caching data, crawling, or reusing data in any way.

#### *1.4.2.2 the data model of the application*

Like the Gluten Free Registry, the Happy Cow data model also falls under the category of the restaurant-review data model, since its information is separated into restaurants, and reviews associated to that restaurant. The Happy Cow also adds a layer of comments that can be made on a review. The data available to users can be broken down into a restaurant object and a review object. The following data model represents a conceptual collection of objects and properties that can be inferred after an exhaustive summary of all available elements and their properties available to end users of the Web application.

#### Restaurant (*object*)

- Restaurant name (*property*)
- Location (*property*)
- Phone number (*property*)
- Website URL (*property*)
- Restaurant email (*property*)
- Category (*property*) – *options*:

- American
  - Chinese
  - Indian
  - International
  - Italian
  - Japanese
  - Thai
  - Western
  - Vegan Options
  - 100% Vegan
  - 100% Raw
  - Mostly Organic
  - Uses Eggs
  - Uses Dairy
  - Macrobiotic
  - Beer/Wine
  - Juice Bar
  - Salad Bar
  - Buffet
  - Fast Food
  - Take Out
  - Delivery
- Has outdoor seating (*property*)

- Reservations required (*property*)
- Wheelchair accessible (*property*)
- Accepts cash only (*property*)
- Hours open (*property*)
- Price range (1-3) (*property*)
- Description of restaurant (*property*)

#### Review (*object*)

- Rating (0-5) (*property*)
- Date posted (*property*)
- Review text (*property*)
- Pros text (*property*)
- Cons text (*property*)

#### *1.4.2.2 data consumption in application*

All data obtained from the Happy Cow application must be obtained through their website or web application. All data must be viewed at the time of requesting the data—meaning that data cannot be cached or saved in any way. Like the Gluten Free Registry, the Happy Cow application asserts ownership of all user-submitted data, and explicitly prohibits the caching or use of their data in other applications. There is no API available.

#### *1.4.2.3 Dish Freely*

The Dish Freely application was chosen for this study because it is unique in that it falls outside of the parameters of typical finder-review applications in the restaurant industry. A relatively new trend is to list menu items and their reviews in the user's vicinity, rather than just listing restaurants and their reviews. This data model (what the

author will call the “Menu Item” data model) allows for many possibilities for helping people with special dietary needs find particular dishes and menu items that they can (and want to) eat.

Dish Freely allows users to log in with their Dish Freely or Foursquare account, search for previously listed restaurants around them, and add gluten free menu items to these listed restaurants. When an a menu item is added to a restaurant, the menu item will come up in users’ searches for menu items based on search parameters. When a user adds a menu item, the app asks what type of dish it is (eg. appetizer, main entre, etc.), and any helpful tags for that item. Also, it allows users to optionally add a review for this menu item. Images are not supported for a menu item review (unlike applications with similar functionality, such as FoodSpotting or Forkly).

In order for a restaurant to be listed within the application, the restaurant must request to be listed via email to the Dish Freely support team. Because users cannot suggest or add restaurants, every restaurant that is listed must be aware of the application, and must actively pursue a partnership with Dish Freely. This scheme inevitably limits the amount of restaurants listed within the application, and thus limits the usefulness of the application to the end user. At the time of writing this paper, there were only 24 restaurants listed in the Denver metro area, and only 12 of these were tagged “gluten free” within the app. There were no gluten free dishes listed in the Denver Metro area (the author submitted the first).

- Type of application:
  - Website and mobile application.
- User-contributed data:

- Users contribute menu item data, as well as menu item reviews.
- Method of submitting data:
  - Normal users cannot submit restaurant data. Users can submit menu item data by first finding a previously listed restaurant, and then submitting a form on the mobile application to submit a new menu item. Users may also submit a form in the mobile application for submitting a review of a menu item.
- Method of filtering data:
  - Only restaurants may submit data about that restaurant, through a partnership with Dish Freely.
- Search parameters:
  - Users may search for restaurants and dishes by location.
- Registration:
  - Users must register with the application in order to use it.
- Data collaboration or API:
  - The application does not have a public API, nor do they facilitate data collaboration in any way.
- Data ownership:
  - In the Terms and Conditions, Dish Freely states that users who submit data retain ownership of that data, but that it will be shared in any multitude of ways.
- Restrictions:

- In the Terms and Conditions, Dish Freely asserts strict policies against caching data and crawling.

#### *1.4.2.3.1 the data model of the application*

As mentioned, Dish Freely uses a “Menu Item” data model. In this model there are three objects: Restaurant, Item, and Review. A restaurant may have one or more items, and an item may have one or more reviews. The following data model represents the conceptual collection of objects and properties that can be inferred after an exhaustive summary of all available elements and their properties available to end users of the mobile application.

##### *Item (Object)*

- Name (*property*)
- Overall Rating (*property*)
- Number of Reviews (*property*)
- Photo (*property*)
- Tags (*property*)
- Price range (1-3) (*property*)
- Restaurant (*property*)

##### *Restaurant (Object)*

- Name (*property*)
- Address (*property*)
- Phone number (*property*)
- Type of establishment (*property*)
- Number of Items submitted (*property*)



- Tag (*property*) - *options*:
  - GFRAP participant<sup>3</sup>
  - 100% Gluten Free
  - Has GF Menu
  - Separate equipment
  - Tagged gluten-free

#### Review (*Object*)

- Rating (0 – 5) (*property*)
- Tip/Comment text (*property*)

#### *1.4.2.3.2 data consumption in application*

Dish Freely does not offer a public API, nor does it facilitate data collaboration in any way. Like the other applications reviewed, Dish Freely prohibits caching their data, or reusing their data in other applications.

#### **1.4.5 Summary of Current Applications: Data Insularity**

While the Dish Freely application is a little different in its functionality and organization, all three of these applications share the characteristic of data insularity. While each application has its own level of popularity and richness of data, it is inhibited by its own limitations. In other words, allowing external developers to use their own creative talents to develop new applications that make use of the existing data in these

---

<sup>3</sup> GFRAP means that this restaurant participates in the Gluten Intolerance Group's

“Gluten-Free Restaurant Awareness Program”

applications, as well as contributing to this data, ensures that the end user benefits as much as possible from the available data.

### **1.5 Goals of Project**

As stated, the goal of this project is to put forth a solution to the problem of data insularity in one particular area: dietary-need finder-review applications.

To solve this problem of data insularity, there must be a way of allowing these applications to consume relevant and standardized data as clients. Furthermore, in order to ensure that the data available to client applications is as relevant and up-to-date as possible, the solution must allow the client to optionally provide data as well. Therefore, the solution must act as a central repository from which clients can consume and provide data. The challenge of creating such a solution is that the data consumed and provided by clients must meet certain criteria in order to ensure that other clients can use the data, and that the data is useful to the user.

Therefore, the general goals of this project will be to create a prototype solution that implements the following requirements:

1. The solution must act as central repository that will allow clients to consume and provide data.
2. The solution must impose a standard model for all data in the repository.
3. The solution must have a data model that will meet the needs of client applications as well as end users

#### **1.5.1 Criteria of Solution**

Following is a more detailed look at the criteria of the solution stated above. The term “client” is used to refer to any application or developer seeking to utilize the

solution. The term “community” is used to refer to the combined groups of end users of the client applications.

***1.5.1.1 Solution must act as central repository that will allow clients to consume and provide data.***

The solution must act as a central repository of data in order to foster a community of shared knowledge in this field. As the brief analysis of applications in this field shows<sup>4</sup>, it is common for an application not to allow other applications to use or cache their data in any way. There needs to be a solution that will allow clients to get their information without having to worry about ownership of the data, and where they know that the data has been submitted by users of the target community. Therefore, there needs to be a central repository in which data is owned and contributed by the community as a whole.

***1.5.1.2 Solution must impose a standard model upon all data that is consumed or provided from the repository***

Because the numerous client applications that will potentially utilize and provide the data in this repository will inevitably have various data models themselves, it is necessary to impose a standard data model to which all applications must adhere. Clients will need to format their requests for data in a specified way. Similarly, the data returned in the response will need to be formatted in a specified way too.

---

<sup>4</sup> See section 1.4: Review of Current Applications

### ***1.5.1.3 Solution must have a data model that will meet the needs of client applications as well as end users***

Because client applications may have varied data models that will utilize data obtained from the repository differently, it must be assumed that some applications will seek restaurant data, and some applications may seek menu item data. Also, to provide as much useful data as possible to the end user, the data model of the repository should store not only restaurant data, but menu item data as well. The data will need to be factual information about the menu items, rather than opinion-based. While the data model should support an overall community rating of this product, the primary data should be informational in nature, so that consuming clients may use this information as a base on which to build their application.

## **1.6 Value of this Project**

Allowing client applications to consume data will make it possible for applications in this genre to expand their current data, and to provide an expanded data set to the user. It will also allow new applications entering this market to not have to start from scratch with their data.

## **1.7 Ethical and Social Impact of Project**

The ultimate goal of this project is to help people with Celiac disease or other special dietary needs to eventually have a more complete set of information. With this goal, there is hope that with this solution there will be more applications appearing in this field, and that these applications (as well as existing applications) will work toward establishing one large community of users with special dietary needs. By having a more complete data set, and by having applications cooperate to establish a larger community

of users that use and contribute to this data set, people with these special dietary needs will be able to find the data that they need, thus improving their quality of life.

## Part 2: Review of Literature and Research

### 2.0.1 Review of Possible Solutions

In order to examine some possible solutions that might meet the requirements specified in the previous chapter, it is first necessary to understand what type of solutions will not work.

Because the first requirement is that the solution will allow clients to consume and provide data, we will first examine methods of consuming and providing data. First, there is the possibility of sending files back and forth between a client and the server. While this type of “batch” processing is appropriate for some types of data and in some scenarios, it is not ideal for a situation in which small client applications are making many requests for getting a single item or place. In the batch enroll scenario, the server needs to parse the client’s file, process this request for data, then assemble a file as a response. Because this scenario is demanding on the system, and because the solution will need to impose a rule against the caching of data on the client’s side (due to many applications prohibiting caching of their data), this solution is not appropriate.

A second potential implementation is a website that allows visitors to search for data and submit data to the repository. While this situation is ideal if the consumers of data were the actual end users, it is not ideal for this situation in which a client application obtains and submits data on the end user’s behalf. In other words, this solution essentially “cuts out the middle man” that this project is intended to help. Furthermore, client developers will be wary of submitting their data to just another application because it drives end user traffic away from their application. Therefore, a website for the data repository is not an appropriate solution. Other methods of “pushing”

data such as RSS feeds are not appropriate for this solution either because the client also needs to submit data.

Because the solution needs to allow client applications to easily make requests to get and submit data, the best solution for this situation is a public API. A public API will allow client applications (on behalf of end users) to get data from the repository, as well as to submit data to the repository by making simple requests through the HTTP protocol. Furthermore, an API will allow the data consumed by clients to be utilized in the moment, rather than caching the data in any way on the client, thus controlling the way in which data is being used by the client.

## **2.1 Review of API as a Solution**

### **2.1.1 An Introduction to APIs**

An API (Application Programming Interface) is a general term for any kind of interface that allows one application to use the functionality of another application. In other words, it is a way for two applications to communicate with one another. In the sense that this paper uses the term, an API is the public interface of a Web service. A Web service is just one form of an API: it accesses an API through a Web protocol (HTTP).

There has been a recent rise in the number of public APIs being used today. As one source estimates, at the time of writing this there were over 7000 published and public APIs actively being used on the Internet, a number that has almost doubled over 2011-2012 alone (DuVander, August 2012). While there may be many reasons why companies are choosing to implement APIs into their infrastructure, one of the main proponents of this popularity is data collaboration (Jacobson, 2012).

It is common for companies to have vast amounts of data that while used internally, is also useful and desirable to other companies and their applications. In Web applications, screen scrapers often crawl through a public-facing website and store its data, thus forcing many of these web applications to define strict policies against crawling and against the caching of data (Jacobson, 2012). For this reason, it behooves any company with this kind of desirable data to consider collaboration policies and a medium of data collaboration. An API provides a medium for data collaboration; it provides a standard, common interface for a client application to obtain (or provide) data.

### **2.1.2 Public vs. Private APIs**

The numbers and trends in “public” APIs must not be confused with the vastly larger number of private APIs that are in existence (Jacobson, 2012). Many companies use APIs internally to allow their applications to reuse the same data and functionality. Companies with private APIs may also choose to expose their services and data to only partner companies. In fact, much of the functionality we see in public APIs is only a fraction of the functionality that these same companies offer privately (Jacobson, 2012). Private APIs are not examined in this paper as a solution for the obvious reason that they are private, and not available to be used.

Public APIs, on the other hand, publish their functionality to the world via the Web. Typically, a public API will define in its terms usage rules and limitations, and will tell the developer the options for using the API. These options usually come in several types of business models for public APIs: tiered, pay as you go, unit-based, and freemium (Jacobson, 2012). In the tiered model, there are different levels of usage. For instance a lower tier may cost less than a higher tier, but the lower tier will allow a much smaller



number of API requests in an allotted timeframe. In the Pay as You Go model the API provider charges the client application based on the number of requests made to the API. In the Unit-based model, the client pays according to specific units of computing or service. And in the freemium model the client may use the API for free, but must pay for various types of additional services. For example, Google Maps uses a freemium model, in which they allow up to 10 thousand calls per day for free, and then charge for exceeding the terms of the plan (Jacobson, 2012).

### **2.1.3 An Introduction to RESTful Web services**

In RESTful Web services, the client makes an HTTP request to a specific URL, along with certain parameters set in either the URL, or in the request header. These requests are typically limited to four operations: GET, POST, PUT, and DELETE, as opposed to SOAP, which allows the developer to define any number of methods (Vaswani, 2010). Responses to the client take the form of a standardized HTTP response, returning a response code, as well as the response data to the client, which typically takes the form of an XML tree, or a JSON array. While initially most RESTful Web services returned only XML, there has been a gradual trend over the past few years toward JSON responses. While there are still many scenarios in which XML is still the appropriate choice, many new RESTful APIs utilize JSON because of its relative ease and simplicity (Jacobson, 2012).

As Jacobson notes, many developers prefer RESTful Web services because they simply do not understand the complexity of using WSDL files and constructing complex SOAP messages, thus making it much easier to get their client applications communicating with an API provider more quickly with REST (Jacobson, 2012). When

comparing REST to other interfacing methods such as WSDL, REST makes it easier for client services to be created to communicate with the provider service because the interface contract is universal for all services (Vinoski, 2007). In other words, the interface contract (the way in which the client application communicates with the API provider) is universal in RESTful services because all RESTful services utilize the same HTTP methods. Developers creating client applications need only to understand how to form these HTTP requests to comply with the provider's interface for invoking its resources. Furthermore, the ability of RESTful Web services (like their SOAP counterparts) to handle large amounts of concurrent requests makes REST a viable alternative to the more traditional SOAP-based Web services (Meng, 2009).

#### ***2.1.3.2 REST: pragmatic vs. dogmatic approach***

There are various interpretations and forms of REST, which was originally developed as part of a PhD dissertation of Roy Fielding (Jacobson, 2012). In this now famous paper, Fielding proposed using the HTTP protocol to allow computers to communicate by dividing URI namespaces into a set of resources (Fielding, 2000). In this model, one can use the standard HTTP verbs (GET, POST, PUT, DELETE) to perform operations on each of these defined resources. There are different camps within the industry which advocate a certain style and interpretation of Fielding's definition of REST.

One group is the REST "purists", who advocate that REST should embody the concept of "Hypermedia as the Engine of Application State" –HATEOAS (Jacobson, 2012). Applications that embody HATEOAS principles dictate that instead of the provider defining a list of resources and actions that the client can use, the client must

discover the functionality that the API provides through an initial request to the root URI of the API, which will then return a list of additional URIs that the client may use (Jacobson, 2012). In this sense, an API that enforces HATEOAS principles reduces the chances of clients requesting stale resources by forcing them to call upon resources dynamically, thus making them more scalable. However, it is very uncommon to see public APIs that adhere to HATEOAS principles. Instead, it is more common to see APIs that utilize what Jacobson coins as “Pragmatic REST” (2012).

The vast majority of public APIs follow *some* REST principles, but not *all*, thus making their adherence to REST more pragmatic than dogmatic (Jacobson, 2012). While there may be many reasons why API designers choose to implement a more pragmatic interface, Jacobson believes that one of the main reasons for this is because HATEOAS is too complex:

*“The HATEOAS principle places such a high bar for the client-side programmer... A pragmatic RESTful approach uses the best parts of the RESTful concept by recognizing that programmers want to understand what they can do with your API as quickly as possible and do it without writing a lot of extraneous code.”*

(Jacobson, 2012, pg. 62).

Jacobson defines a list of pragmatic RESTful characteristics that API designers should adhere to:

- Implement a well-designed URI pattern
- Use standard and obvious sets of optional parameters for each API call

- Make the data format for both requests and responses clear and straightforward to client programmers
- Use standard HTTP return codes
- Hide all other details, such as security, in the HTTP headers
- Establish a clear versioning convention

(Jacobson, 2012)

## **2.2 Review of Existing Solutions (APIs)**

Having established a public RESTful API as a solution for this project, research was conducted on several of the most popular and relevant public APIs in the Restaurant industry as potential solutions: the Yelp API, the FoodSpotting API, and the Food Genius API. Each of these APIs provides restaurant- and/or menu item-related data to consuming client applications, and each follows a type of “review” model (data collected from users is in the form of a review).

The brief observational study that follows is composed of four parts. First, there is a brief explanation and overview of the API, which contextualizes why that particular API was chosen, the application with which it is associated (if one exists), and its main features. Second, there is an examination of a few critical API characteristics. Third, the data model of each API is analyzed, based upon observable patterns of the structure of the API’s available data. This examination is important because it assesses how suitable this data model is as a solution. It is important to point out that the data model is the author’s own interpreted findings based on the data available. And finally, this study presents an analysis of these results, and discusses this API as a suitable solution.

### ***2.2.0.1 Explanation of API characteristics observed***

While there are many important characteristics that should be observed in any API, this study presents those that are most relevant for this project. First, this study examines the API's status as public, private, or semi-public: public being available to use by anyone that registers, while semi-public being only for those who subscribe to the service in some way. Next, the study looks at whether or not the API allows data to be consumed or provided by the client. In many public APIs the client is only allowed to consume data from the API, rather than provide it. The solution chosen in this paper specifies that the API must allow both consumption and provision of the data from the client. Then the study examines the most granular level of data (examined more in depth during the analysis of the API's data model). And finally, characteristics such as the API's policies on data caching, ownership of user-submitted data, limitations and quotas, and developer policies are examined. These characteristics are also important because they show how the APIs specify how the data should be used.

### ***2.2.0.2 Explanation of why these APIs were chosen***

The three APIs examined in this paper were chosen as representatives for their areas. Out of the very few APIs in the restaurant industry for utilizing user-contributed finder-review data, each one of these APIs represents a particular niche: the Yelp API is a quite popular API used by many applications to get basic business and restaurant reviews and information; the FoodSpotting API is unique because of its granularity in its data; and the Food Genius API represents a Data as a Service (DaaS) solution that specializes in providing data for a fee with its partners.

### 2.2.1: Yelp API

Below is a review of the Yelp API, which is based on Yelp’s online documentation<sup>5 6 7</sup>, as well as the author’s experimentation with the API itself. Because it is open to the public, and because Yelp allows developers to access the API in order to learn more about it, the author was able to review its data model in depth.

- API status:
  - The Yelp API is public, with controlled full access. It is based on the “freemium” business model—free for a limited amount of requests.
- API type:
  - It is a RESTful API that returns JSON response data.
- Client actions:
  - Clients may consume data, but not provide data to the API.
- Data granularity:
  - Available data at its most granular is for a business / restaurant.
- Client authentication method:
  - OAuth authentication is required for all access to the API.
- Limitations:

---

<sup>5</sup> API 2.0: Search API (n.d.)

<sup>6</sup> API 2.0: Business API. (n.d.)

<sup>7</sup> API Terms of Use. (n.d.).

- All developers are limited to 100 requests per day, until the application has been officially approved by Yelp, at which time the number of requests increase, and the developer uses a freemium model.
- Ownership of data:
  - Yelp asserts ownership of all data in their Terms and Conditions.
- Caching data policy:
  - Storing, caching, or reusing data in any way other than immediately displaying it to the end user is not permitted.

#### ***2.2.1.1 Overview of API***

The main functionality of the Yelp API is to allow a client to search for local businesses, passing in a number of required and optional parameters, returning a list of businesses within certain geographic information along with basic information about the businesses. The API has two main actions: “search” (a general search for a list of businesses) and “business” (a focused search for a specific business). The Yelp API model only supports the “GET” HTTP method, and does not allow clients to be providers of data.

#### ***2.2.1.2 Data Model of API***

The “search” action of the Yelp API returns a list of businesses relevant to the client’s search parameters, while the “business” action of the API lists all of the details

about a single business. The API's data model can be broken down into two objects: business, and review.<sup>8</sup>

While the "search" action of the Yelp API gives the client a high level description of each business returned in the list of results, by using the "business" action of the Yelp API the client is able to narrow the focus onto an individual business and obtain more specific data.

#### Business (*Object*)

- restaurant id (*property*)
- restaurant name (*property*)
- restaurant categories (*additional object*)
- phone number (*property*)
- is\_claimed (*property*)
- is\_closed (*property*)
- image url (*property*)
- location (*additional object*)
- mobile url (*property*)
- review count (*property*)
- snippet text (*property*)
- business URL (*property*)

---

<sup>8</sup> When describing the data models of these APIs, the author is making an interpretation based on the structure of the response data returned from these APIs. This means that the data models described here are syntheses of the results found.



- deals (*additional object*)
- rating (*property*)
- reviews (*additional object*)

#### Review (*object*)

- review id (*property*)
- excerpt (*property*)
- text review (*property*)
- number rating (*property*)
- time created (*property*)
- user created (*property*)

#### **2.2.1.3 Assessment of use of API for dietary need applications**

Users can include certain keywords in the optional “category\_filter” parameter of the GET request, which filters the results based on these parameters. These category keywords are pre-defined by Yelp. In this list, there are the following keywords that are associated to special dietary needs:

- Gluten Free (gluten\_free)
- Vegan (vegan)
- Vegetarian (vegetarian)

(Category List, n.d.)

By allowing the client to filter search results by the category filter, it is possible to separate restaurants that are known to cater to those with one of the three major dietary needs (gluten free, vegetarian, vegan) from restaurants that do not. When used properly

by users, the category filter is a powerful tool to find restaurants, but not specific menu items. In other words, the client/user is able to determine that the restaurant has a “gluten-free” tag, but not which items are gluten-free. Furthermore, because the Yelp community of users is not primarily focused on special dietary needs like the special dietary need “finder-review” applications, many restaurants listed in Yelp that should have a “gluten free”, “vegetarian”, or “vegan” category do not have it listed.

Similarly, including these dietary need keywords in the “term” parameter return a list of restaurants that have this word mentioned in a review of the restaurant, or that have a category like the search term. This causes the problem of “review pollution”.<sup>9</sup> This means that any time a word such as “gluten-free” is mentioned in a review, that restaurant will be returned in the search results. This poses a problem if a user mentioned that a restaurant was “not gluten-free” in a review, and demands the need for the data model to be at the menu item level, which the Yelp model does not support.

### **2.2.2: FoodSpotting API**

Below is a review of the FoodSpotting API, which is based on its online documentation<sup>10 11</sup>.

- API status:
  - The FoodSpotting API is in a “semi-public-beta” state, and is offered to interested partners.

---

<sup>9</sup> For more information of the concept of “review pollution”, see section 2.3.

<sup>10</sup> Foodspotting | API Documentation. (n.d.)

<sup>11</sup> Foodspotting | Terms of Service. (n.d.)

- API type:
  - RESTful API that returns JSON data only.
- Client actions:
  - Clients may consume and provide data.
- Data granularity:
  - Available data at its most granular is for “sightings” of menu items, and reviews of those sightings.
- Client authentication method:
  - The API uses OAuth for requests that attempt to act upon a FoodSpotting user’s behalf. Otherwise all other requests use an API key to authenticate the application.
- Limitations:
  - Limitations are not specified, and the author was unable to get information from a representative.
- Ownership of data:
  - FoodSpotting specifies in its Terms and Conditions that users maintain ownership of their data.
- Caching data policy:
  - In its Terms and Conditions, FoodSpotting explicitly prohibits storing, caching, or reusing their data in any way.

### *2.2.2.1 Overview of the API*

FoodSpotting is a Web and iPhone application that allows users to share experiences and images of items that they have had at restaurants. When a user submits an entry for a menu item, the application allows the user to check in at a location (on the mobile version), submit a photo that they have taken of the menu item, and enter a description/review of the item. Other users are then able to view this entry based on their location and other search parameters. These user-submitted entries are referred to as “spottings” by FoodSpotting.

Of particular importance, the data model of the FoodSpotting API allows for items to be associated with a restaurant, and reviews to be associated to items. This contrasts with the traditional “restaurant model”, which associates a review with a restaurant (rather than menu item). This not only creates a granularity in the data that is available to users, but it also avoids replication in items submitted by users.

Authentication (via a three-way handshake with FoodSpotting and the end user through OAuth) is required for access to protected resources in the FoodSpotting API (items that contain user-related data). However, authentication is optional for access to certain non-protected resources, such as sightings. Access to the API is controlled by FoodSpotting, and a developer must request access to it.<sup>12</sup> For this reason, this type of API can be considered what many call a “semi-public API”<sup>13</sup>. Even though FoodSpotting

---

<sup>12</sup> The author was denied access to the API when requested for educational purposes.

<sup>13</sup> Companies often make their API and documentation public, yet tightly restrict access to the API to a small number of partners.

has released the documentation for their API publicly, access to this API is limited to partners (DuVander, A. (2011, September 12)).

This API allows the client to access data from six different resources: items, places, sightings, reviews, comments, guides, and people. The client specifies a resource, and then specifies the name of the resource. FoodSpotting organizes its “spotting” data into two separate objects: “sightings” and “reviews”. A sighting is a container that associates an item with a place (restaurant, bar, etc.). A review is simply a user-submitted review (with a photo and optional note) of an item. When a user submits a review of an item, if a sighting has already been submitted for that item, then that review will be associated to an existing sighting; if there has not been a sighting for that item, then a sighting will automatically be created, along with its first review. When searching for data through the API, the client will typically first search for sightings, and then go deeper into the data by searching for reviews for that item (sighting).

#### ***2.2.2.2 Data Model of API***

In order to better understand the specificity of the data available on menu items, it is first necessary to observe what constitutes the data model for a “sighting”, “review” and “item”.

Sighting (*object*)

- sighting id (*property*)
- place (the restaurant associated with this sighting) (*property*)
- count of reviews (*property*)
- date created and date last updated (*property*)
- latitude and longitude coordinates of sighting (*property*)

- item (associated with sighting) (*property*)
- wanted (a tag if someone likes this sighting) (*property*)
- creator id (identifies the user who made this sighting) (*property*)
- current review (*property*)
- distance (*property*)
- nommed (a flag for the application) (*property*)
- last reviewed date (*property*)
- wants count (the amount of times someone has liked this sighting) (*property*)
- ribbons count (a social media counter for the FoodSpotting app specifically)  
(*property*)

#### Item (*object*)

- id (identification of the specific item) (*property*)
- name (*property*)

#### Review (*object*)

- Place (*property*)
- comments\_count (number of comments made on this review) (*property*)
- nommed (a flag for the application) (*property*)
- person id (user who made the review) (*property*)
- date created and date updated (*property*)
- great finds count (count of times this review has been flagged by a user as a great find) (*property*)

- note (the optional text that a user can submit with the review) (*property*)
- item (*property*)
- great shots count (number of times a user has tagged the image associated with this review as a “great shot” (*property*))
- person (additional information about the person who made the review) (*property*)
- image (*property*)
- wanted (a tag if someone likes this sighting) (*property*)
- sighting id (an identifier tying back to the sighting to which this review is associated) (*property*)
- shared to (whether or not this review has been shared to Facebook, Twitter, Flickr, or Foursquare) (*property*)

### ***2.2.2.3 Assessment of Use of API For Dietary Need Applications.***

By looking at the data model, and by understanding how data is consumed by the client, it is evident that the FoodSpotting API does not allow the client to search for menu items based on dietary needs, and does not provide data specific enough to be useful to a client application for providing dietary need data to users. In other words, the FoodSpotting API is only a viable solution if it tracked whether or not the menu item fits special dietary needs, and if the API allowed the client specify this in the search parameters. As it is, the FoodSpotting API does not meet the specified requirements of this project as a solution.

### **2.2.3: Food Genius API**

Food Genius is a Data as a Service (DaaS) company that specializes in providing restaurant and menu item information to client applications for a fee. It currently does not

have an application associated to it, but it encourages the developer community to use its API in a trial state to build applications.

- API status:
  - The Food Genius API is semi-public, and is based on the tiered business model, in which developers may get more access to the API and more requests for a bigger fee.
- API type:
  - RESTful API that returns JSON or XML data.
- Client actions:
  - Clients may consume and provide data.
- Data granularity:
  - Available data at its most granular is for menu items.
- Client authentication method:
  - The client is authenticated through the use of an OAuth key obtained from Food Genius.
- Limitations:
  - After initial approval into the trial stage, the client application is subject to a small number of queries for development purposes. Production access is based on differing pricing options.
- Ownership of data:
  - Food Genius asserts ownership of all its data in its Terms and Conditions.
- Caching data policy:



- Food Genius specifies in its Terms and Conditions that caching data is strictly prohibited.

### ***2.2.3.1 Overview of the API***

Unlike the other companies and applications observed in this paper, Food Genius is primarily focused on its API. In its mission statement, Food Genius states that the purpose of its API is to provide data to applications in the food and restaurant industries (source 3).

### ***2.2.3.2 Data Model of the API***

The data model of the Food Genius API is similar to that of FoodSpotting: data is organized into several distinct objects that are accessed along with certain search parameters. The Food Genius API organizes its data into five main objects: restaurant, place, location, menu item, and interaction. The ability for a client to provide data is limited only to reviews of a location or menu item on a numbered scale.

Menu item (*object*)

- distance (*property*)
- description (*property*)
- location (*property*)
- id (*property*)
- item name (*property*)
- price (*property*)

Interactions (reviews) (*object*)

- Rating (a number scale review) (*property*)

- Created date (*property*)
- Reviewed (the URL / Food Genius ID of the menu item or restaurant that was reviewed) (*property*)
- Range (number of the range for the rating – eg. 10.0) (*property*)
- Units (*property*)
- Identity (user that created the review) (*property*)

### ***2.2.3.3 Assessment of Use of API for Dietary Need Applications***

Like FoodSpotting, the Food Genius API allows the client to include a query string for the name of a menu item in an HTTP request. In other words, the client may query “vegetarian”, but the word “vegetarian” must then be in the name of the item (“vegetarian burrito”). Because in this model a menu item does not include any kind of keyword or category, data cannot be accessed at a granular enough level to determine if an item suits a specific dietary need.

## **2.3 Why None of These Solutions Fit the Requirements**

To summarize the suitability of the above APIs as a solution for this project, it is first necessary to reiterate the three main requirements.

1. The solution must act as central repository that will allow clients to consume and provide data.
2. The solution must impose a standard model upon all data that is consumed or provided from the repository
3. The solution must have a data model that will meet the needs of client applications as well as end users

The table below summarizes how each of the solutions examined in this chapter meet these needs for a solution.

Name	Yelp API	FoodSpotting API	Food Genius API
Data Model Type	Restaurant-level Review	Dish-level review	Restaurant-level and Dish-level
Solution Stores Data	Yes	Yes	Yes
Non-Trial Public API Open to All Client Developers	Yes	No	No
Allows Clients to Consume Data	Yes	Yes	Yes
Allows Clients to Provide Data	No	Yes	No
Enforces Standard Data Model on Submitted Data	N/A	Yes	N/A
Data Model at Both Restaurant and Dish Level	No	No	Yes

**Figure 1: Table Comparing Possible Solution APIs**

While the above APIs meet the first two requirements for a solution (acting as a central repository for clients to consume and provide data, and imposing a standard model on data that is provided and consumed from the repository) they do not satisfy the

third requirement: having a data model that will support the needs of client applications and end users. This third requirement is vitally important in solving this problem of data insularity amongst dietary need applications because without being at a granular enough level the data loses its significance and usefulness to the end user.

In the Yelp API, we see dietary need data being stored as optional tags associated to a restaurant. This is helpful only if users actively include this optional data, and it is only helpful if the end user wishes to know if a restaurant is known to have some sort of menu item that meets that dietary need. However, this data cannot be associated directly to menu items, therefore limiting the usefulness of this feature for end users.

On the other hand, in the FoodSpotting API, we see data stored at a granular level: listing data about menu items. This API has the granularity that the Yelp API lacks, but it still lacks any kind of way to associate dietary need data with menu items. This is because the FoodSpotting data model allows users to enter data about an item only through the item's name, and in a text review of the item. In other words, FoodSpotting's data model does not support tagging a menu item by a specific dietary need type. Thus, using an item's name, or using a text review associated to an item to associate a dietary need with an item opens up the problem of what the author will call "review pollution".

Review Pollution is the result of data at its most granular state being in the "Review" entity. A single review typically contains a rating of the restaurant (or item) and a block of text that constitutes the user's review. This means that user-initiated searches for a particular keyword must look at the entire block of text for the search term. This causes issues if a user specifies in a review that an item is "not gluten free", or when

they say that “I wish this were gluten free”, etc. In other words, the text review pollutes the accuracy of searching for a specific term.

Similar to FoodSpotting, the Food Genius API also does not have a data model that will support the needs of end users. Because data at its most granular is in a text review, this poses the same problem of review pollution we see in the Food Spotting API.

### **2.3.1 The need for a new solution**

Because the APIs examined above do not satisfy the core requirements for a solution to this problem of data insularity, there is a need for a solution to be developed that will satisfy all of these core requirements.

## **Part 3: Project Methodology**

### **3.1 Research Method Chosen: Constructivist Epistemology**

As stated, the goal of this project is to put forth a solution to the problem of data insularity amongst dietary need applications in the restaurant industry. Because an existing solution to this problem does not yet exist<sup>14</sup>, the author will adhere to a constructivist epistemology, seeking to solve the problem of data insularity in the field of dietary need applications through the creation of a prototype solution. This prototype solution will implement a central repository of standardized data that can be accessed and added to by client applications through a public RESTful API.

#### **3.1.1 A RESTful API as the Appropriate Solution**

A public RESTful API is the appropriate solution to this problem for several reasons. First, a public RESTful API is accessible to developers of client applications. This accessibility comes partially from the fact that RESTful APIs are quickly becoming ubiquitous, thus meaning that more and more developers are readily familiar and comfortable with the idea of using an API (Jacobson, 2012). In addition to their popularity, RESTful APIs are accessible because they use a standard set of HTTP methods to request resources.<sup>15</sup> Accessibility is important in getting the developers of these applications to easily adopt this standard model into their own application flow.

The second factor that makes an API the right fit for this solution is that it imposes and enforces a standard data model for the elements that are consumed and

---

<sup>14</sup> The lack of an existing solution is discussed more in the second chapter.

<sup>15</sup> See section 2.1 for more of a discussion of RESTful APIs.

provided. The API is the gateway through which developers submit and request the data of the central data repository. Thus, the central repository acts as the standard data model to which client applications adhere, and the API enforces this data model by forcing submissions to adhere to this model, and formatting response data in this model as well.

And finally, by creating an API that enforces a focused and orderly data model, the needs of both client applications and end users will be met because it will allow a flexibility and granularity in the data available to both applications and users.

Applications will benefit from an API that allows them to access data on several different levels, such as both the “restaurant” level and the “item” level, because each application has its own unique data model and needs. It is likely that some applications may seek to provide users with information about restaurants and not dishes, or vice versa. This flexibility will benefit users because they can obtain information on a restaurant, and then view other data resources associated to that restaurant, such as items and item reviews.

### **3.2 Project Requirements**

With an API established as an appropriate solution to the problem at hand, it is necessary to define general requirements for this solution in terms of the needs of an API.

#### **3.2.1 API Solution Requirements**

Following is a list of the general requirements of the solution:

- The solution must allow clients to consume and provide data
- The solution must scale to current and future dietary needs
- The solution must allow for two different data models: restaurant-review model and menu item-review model

In addition to these core requirements, the solution as an API must consider the following key API needs and characteristics:

- The API must enforce appropriate security policies
- The API must enforce appropriate data usage policies
- The API must enforce appropriate developer policies
- The API must provide support to developers seeking to utilize its service

### **3.3 Requirements in Depth**

In order to better understand these requirements, they will be explored more in depth below.

#### **3.3.1 Requirement: Allow Clients to Consume and Provide Data**

Clients will be able to consume and provide data by adhering to the policies and requirements of the API. Through consuming data, developers will be able to integrate the API's data into their own applications. Similarly, developers will be able to share their application's data with the API community (and hence, other applications). Through the ability to provide data to the API, the goal is that the API's data continue to grow, providing an increasing set of data to consuming applications.

#### **3.3.2 Requirement: Scale to current and future dietary needs**

The API will initially support the three major categories of dietary needs commonly found in applications: gluten-free, vegetarian, and vegan entries. However, to be scalable for future expansion, and to allow the possibility of other dietary needs to be included, the API will also be scalable to include other future categories of dietary needs.



### **3.3.3 Requirement: Allow for the Two Levels of Data (“Restaurant” and “Menu Item”)**

The API will support data on two different levels: the restaurant data model, and the menu item data model. In the restaurant data model, data will be organized by restaurant, and a client will be able to consume and provide data regarding a restaurant. In the menu item model, a client will be able to consume and provide data regarding specific menu items, and a user’s experience with items. Clients will be able to request data at either of these levels, or will be able to request all menu items per restaurant.

### **3.3.4 Developer Support**

The API will have adequate support documentation to assist developers in using the API. Good support documentation will also encourage developers to use the API. This documentation will also include a sample client application that uses some of the features of the API to demonstrate one way that it may be used.

### **3.3.5 API Policies of Usage**

In addition to the main API requirements listed above, the API must also take into consideration certain policies that pertain to the usage of an API. Even though the API will be open to the public, certain policies must be established in order to protect the API against malicious and improper use, and to ensure that the API is used to its fullest potential.

#### ***Data Usage Policies:***

The API will implement and enforce data policies in order to protect itself against abuse and improper usage. The three most important policies are for caching data, rate limiting, and upload policies.

***Security Policies:***

The API will enforce an authentication and authorization scheme in order to control access to the data. Even though this will be a public API available to all external developers, it will implement security measures to enforce the data and other policies by restricting access to those who violate these policies.

***Quotas and Limitations:***

The API will implement a limitation quota on the amount of data and/or the frequency of requests for data in order to ensure that the API continues to function properly.

***User-Submitted Data:***

Users of the API must agree to set of Terms and Conditions, and must agree upon a schema of data ownership and privacy policies. This is to ensure that data being provided to the API is either the property of that application, or if it is user-owned content, that the users of their application have agreed that their data can be provided to a third-party Application.

**3.4 Methodology: Implementation of Project****3.4.1 Process Model Chosen**

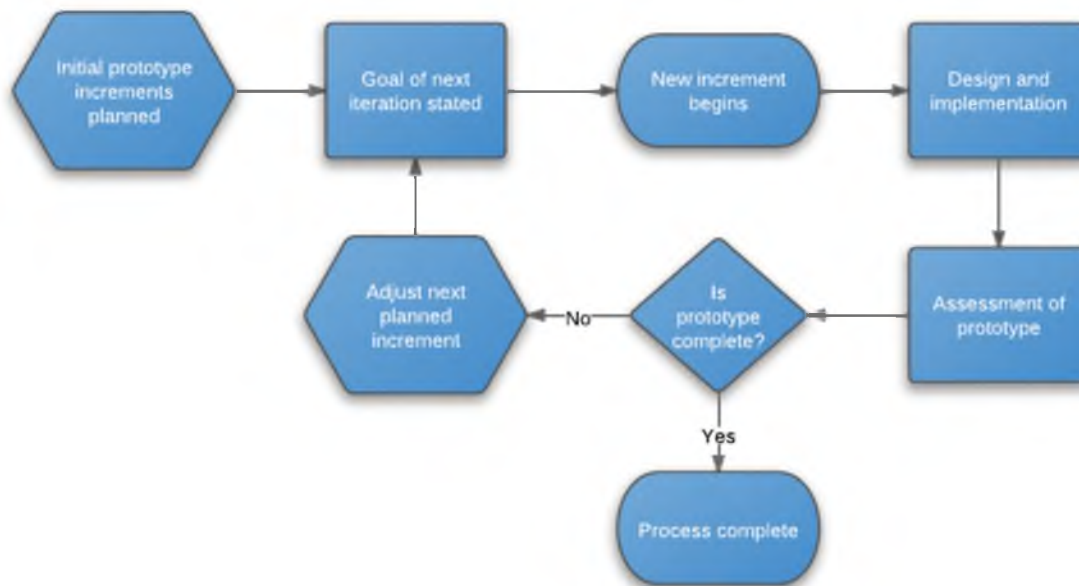
When this project began, it was determined that an incremental and iterative development process model was most appropriate to guide the software development lifecycle, due to the possibility of unknown requirements being introduced at a later time. The goal was that each iteration produce a working prototype that is closer to the desired end product. However, because the author was the only person working on the project, as well as the product owner, a strict adherence to agile team processes such as Scrum or XP

was not appropriate. Therefore, a modified rapid evolutionary prototyping process model was chosen for this project.

At the beginning of this process an initial list of requirements was created based on the stated goals of this project<sup>16</sup>. Having these requirements, an initial increment plan was developed that will attempt to plan evolutionary prototypes into two week increments. This initial increment plan will serve as an initial map to get the project started, and then will inevitably evolve with each iteration. At the beginning of each iteration the goals of that iteration's prototype will be stated, which will then be assessed at the end of that increment. The assessment of how these goals have been met, as well as an assessment of how the overall requirements have been met will guide the planning of the next increment and set of goals for that iteration.

---

<sup>16</sup> See section 3.3.3 for the requirements in depth.



**Figure 2: Flowchart of Spiral Prototyping Software Development Lifecycle Used**

### 3.5 API Component Design and Implementation

The design of the solution and its components are discussed in this section. This begins with a discussion of the intended overall flow of the solution, then discusses the data model of the repository and API, how access to the API is controlled, and how developers are informed about the API.

#### 3.5.1 Documentation Website

The documentation website allows client developers to gather information about the API, see an example of how the API can be used in a client application, and request access to the API. The primary purpose of the documentation website is to be informational, describing in detail the data model used, how requests are to be formatted, how responses from the API are formatted, and how to obtain access to the API. Developers can read the terms and conditions of using the API and fill out a form to

request access to the API. All requests for access must contain the developer's name and email address, the name and URL of the application, how many requests per day are expected, and a short description of the intended use of the API. This information is gathered to understand how the API is being used by client applications, as well as to control access to the API. For example, a client application that intends to use the data improperly, or a client application that expects to have hundreds of thousands of requests per day may be denied access for this version of the API. All requests for access are examined by the administrator of the API (the author) in order to ensure proper usage.

Because the needs of the website were relatively simple and not the primary focus of the project, the documentation website was implemented in PHP and MySQL, utilizing the WordPress CMS framework.

### **3.5.2 Sample Client Application**

As a supplemental part of the API's documentation, and in order to help client application developers visualize some potential ways in which the API can be used, a simple client application was developed in order to demonstrate some of the API's main functionality, such as viewing and posting items, places, and item-reviews. The scope of this application was to give a simple demonstration of the main functionality of the API in the setting of just one potential client application scenario, and not to be an exhaustive embodiment of all of the API's functionality. To benefit client application developers, a link to this sample application, as well as its documentation, is included in the API's documentation website.

The sample client application was implemented in PHP and the jQuery Mobile framework. The JavaScript framework "jQuery Mobile" that was used to create a

standard user interface that can easily be accessed by most of the major current browsers on standard computers and mobile devices. This application does not access the data repository directly, but instead makes REST requests to the API to access the data, as is intended in all client applications that utilize the API.

### **3.5.3 Data Model and Central Data Repository**

A standard data model was designed in order to ensure that data can be used by client applications and end users. The data model of the API was designed with granularity and extensibility in mind in order to be useful to the end user, while still usable and standardized for a multitude of client applications. To accomplish this, a resource-based data model was created that separates key concepts into a Place resource, an Item resource, a Sighting resource, and a Report resource. In this data model a user shares an experience with an item that he or she had at a place. This is called a “sighting.”

#### ***Place Resource***

The Place resource constitutes a “place” that a user can visit. This can equate to a restaurant, coffee shop, bar, pub, taco stand, or any “place” where someone can purchase food or beverage items that are ready to consume there. In most cases a place equates to a restaurant. The Place resource contains useful information about a place, such as:<sup>17</sup>

- The place’s name
- Whether or not that place is a chain
- The place’s address and geo coordinates
- An image URL associated to that place

---

<sup>17</sup> See section 3.5.4 for a complete representation of the Place resource

- The place's phone number

In its implementation, the Place resource takes the form of a “place” table in an RDBMS.<sup>18</sup>

### *Item Resource*

The Item resource equates to a food or beverage “item” that a person can consume at a place. An “item” typically constitutes a dish or special beverage at a restaurant. The Item resource contains all useful information about an item, such as:<sup>19</sup>

- The item's name
- A description of the item
- Dietary “tags” such as “gluten free”, “vegetarian”, or “vegan”
- Ingredient list (when utilized by a user that has the ingredients)
- Category tags that pertain to this item
- Other metadata

The Item resource is dependent upon the Place resource: there can be one or many items for each place, but each item must be associated to one place. In other words, a dish cannot exist without a restaurant. In its implementation, the Item resource takes the form of the “item” table in the RDBMS, with a foreign key dependency on the place table.<sup>20</sup>

---

<sup>18</sup> See figure 3 for an ERD of the data model

<sup>19</sup> See section 3.5.4 for a complete representation of the Item resource

<sup>20</sup> See figure 3 for an ERD of the data model

### ***Sighting Resource***

The Sighting resource was designed to allow users to share their experience with a particular item they had at a place. This resource contains information related to a user's experience with a place or item, such as:

- A review of the place or item, in the user's words
- A rating of the place or item on a scale from 1 to 10

While a sighting is also considered to be a review, the resource itself is designed to be scalable to become a larger concept. For instance, if it is decided in a future version of the API to allow the upload of images associated with a sighting, then this may be added as well without compromising the granularity or regularity of the data.

### ***Report Resource***

The Report resource represents an issue that is reported regarding one of the other resources (Place, Item, Sighting). Because the API is designed to consume user-submitted data, this leaves open the possibility of inappropriate or inaccurate data being submitted. While the API administrator will take preventative methods to weed out inappropriate data, it is still possible that inappropriate data and especially inaccurate data will find its way into the API data. A client application can use the Report resource to “flag” a resource, notifying the API administrator that something is not right about that resource's information. Data that belongs to the Report resource includes:

- The entity (resource) type, such as place, item or sighting
- The entity's number to identify it
- Flag inappropriate
- Flag inaccurate



- The user's note about this resource (why it is inappropriate or inaccurate)
- Other metadata

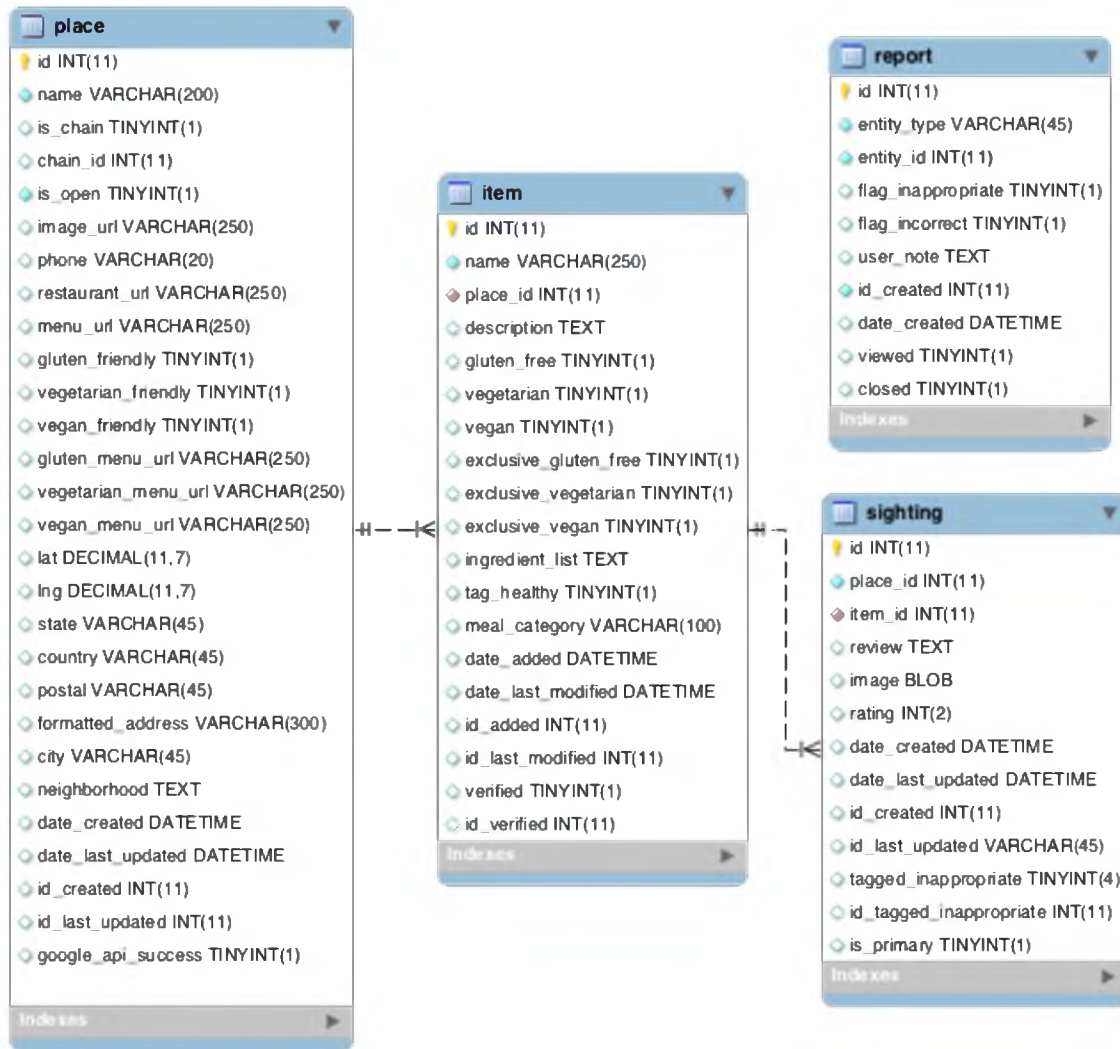
The Report resource must have both a resource type and the entity identification number to be created.

### ***Explanation of Data Model***

As mentioned previously, in the context of this data model a user has an experience with an item, rather than with the place as a whole. While many restaurant finder applications associate a review to a restaurant, the aim of this API's data model is to provide data at a more granular level in order to more adequately assist the end user. In order to be more useful to the end user who is looking for items that they may eat, the data model associates experiences to items rather than places. It is possible that a user may have a good experience with one dish, and a bad experience with another dish at the same restaurant. In this case, a generic review of the restaurant would not provide the user with data that is granular to the item level. Therefore, one or more sightings may be submitted for an item. Similarly, there may be one or many items for a place. A place must exist for an item to exist, and an item must exist for a sighting to exist.

### ***Implementation of Data Model***

The data model was implemented as a MySQL relational database. In the database, each of the data model's resources was implemented as a table. A visual representation of these resources as RDBMS entities can be seen in the following ERD:



**Figure 3: ERD of Solution Data Model**

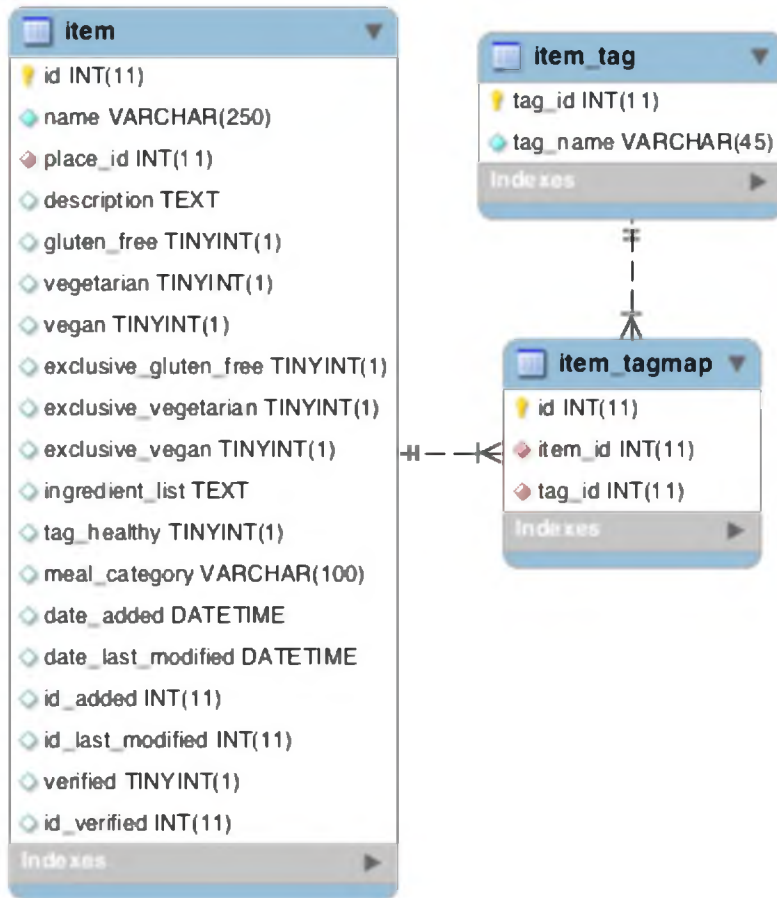
As can be seen in figure 3, there is a one to many relationship between an item and sightings, and there is a one to many relationship between a place and items.

### *Important Additional Entities*

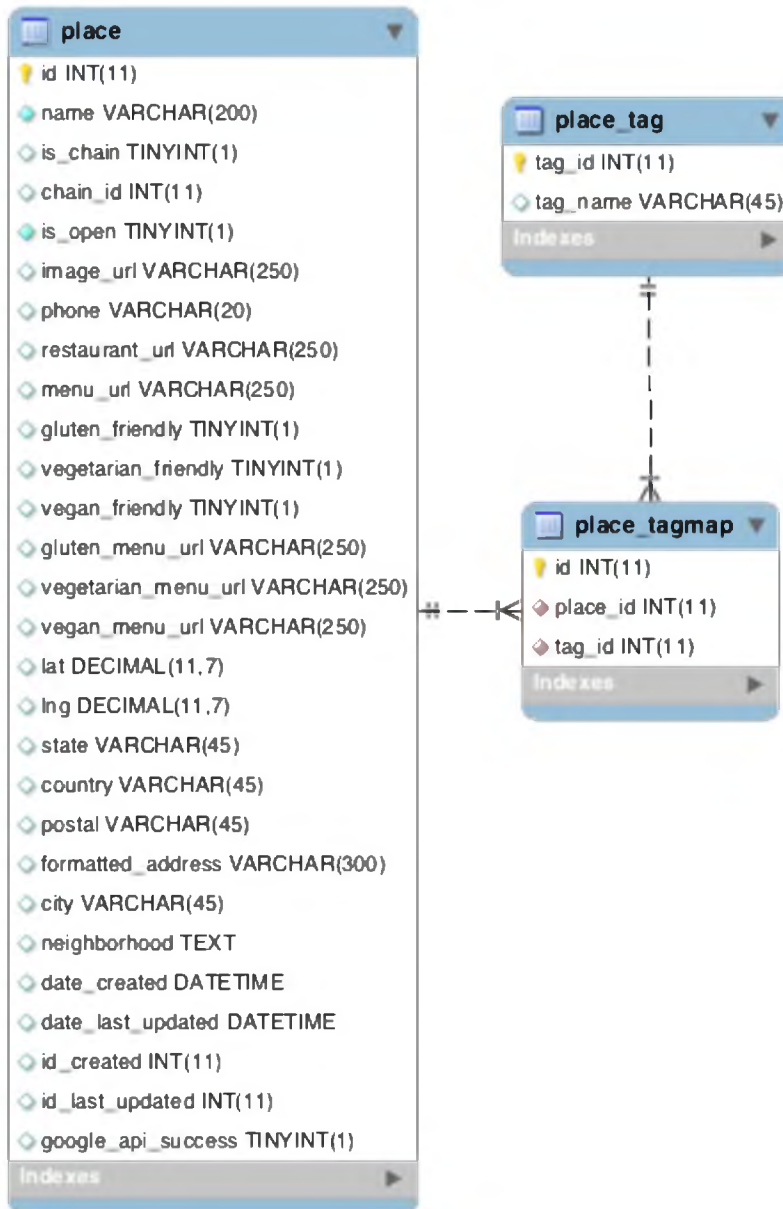
In addition to the main entities that represent the resources previously described, several additional entities were included in this representation because of their importance.

*Categories*

In order to allow one or many categories to exist for an item or place, there exists a “place\_tagmap” and a “place\_tag” table for places, and an “item\_tagmap” and “item\_tag” table. The administrator of the API creates categories in the tag table, which are later mapped to an item or place when they are added.



**Figure 4: ERD of Item Category Implementation**



**Figure 5: ERD of Place Category Implementation**

In this model, the `place_tag` table has a one to many relationship with the `place_tagmap` table, which has a many to one relationship with the `place` table. In other words, there can be many tagmaps that exist for a single place, which maps to a single `place_tag`. The same can be said for the `item`, `item_tagmap`, and `item_tag` tables. Having a separate table

to associate categories to an item or place allows the possibility of these entities to have zero or many categories, without crowding the place and item tables.

### *Permissions*

The permissions entity maps an application to a set of permissions for accessing each of the resources. As the API scales in the future, it is anticipated that access (and types of access) to each resource will need to be controlled on an application level. For instance, if a particular application repeatedly posts incorrect data to the Place resource, the administrator can choose to ban this application from using the POST method on the Place resource through the permissions entity.



**Figure 6: ERD of Permissions Resource**

As of this version of the API, upon a client application's approval for access to the API, the administrator creates a row in the permissions table for the application, assigning it POST and GET access to each of the resources.

### *Application, User, and API Access Log*

The Application, User, and API Access Log entities are all inter-related and used to control access to the API. In this model, a user may be assigned to one or more applications, but an application has only one primary user associated to it. This model was chosen to allow a client developer to have several applications that use the API. For instance, a user may have a development and production version of a client application, each using the API at different levels. In addition, each time an application accesses the API in any way, that particular session is recorded in the API Access Log entity. Through this entity, access to the API can be monitored and controlled. For instance, if a client application uses the API excessively or abusively, the API administrator can set a limit to how many requests may come from that particular application within a given timeframe. While in this version of the API there are not any limitations enforced, as the API is used more in the future it will become necessary to impose limitations. The Application, User, and API Access Log entities were implemented as tables in the RDBMS.



**Figure 7: ERD of User and Application Authentication**

In this model, the user table has a one to many relationship with the application table, which has a one to many relationship with the api\_access\_log table.

### 3.5.4 RESTful API

As mentioned, the API acts as an intermediary between the client application and the central data repository. The goal of the API is to facilitate easy access for client applications to the data in the repository, while also enforcing a standard data model that maintains the usability of the data for client applications and the usefulness for end users of these client applications. The API was designed to accomplish these goals through a simple and standard mode of access to the API, and standard request and response formats.

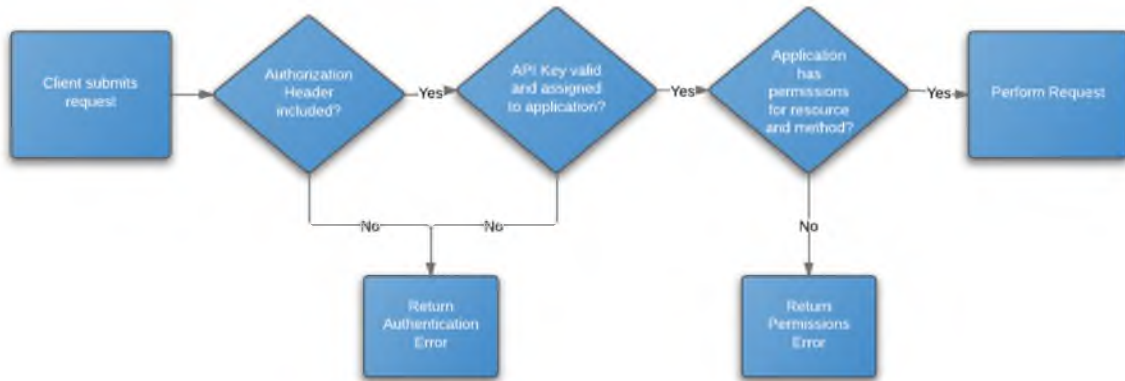
### *Access to the API*

In order to access the API, an application must first obtain an API key.<sup>21</sup> This API key must be included in the HTTP “Authorization” header of all requests. The API analyzes the HTTP request for this header, checks to make sure this API key is valid and associates the API key to a registered application, then checks if this application has access to this particular method on this particular resource. For example, a client application that has previously obtained an API key from the API makes a GET request on the Item resource in order to obtain a list of items, including the API key in the Authorization portion of the HTTP header. The API then validates the API key and checks to see if this application has been granted “GET” access to the Item resource. It sees that the application has access, and then performs the action of getting and returning the requested list of items. If the application does not submit a valid API key, the API returns an error response with the reason of “failed authentication”. If the application submits a valid API key yet does not have access granted to that method upon that resource, the API then returns an error response with the reason of “access to resource denied”.

---

<sup>21</sup> See section 3.6 for obtaining an API key





**Figure 8: Flowchart of Application Authentication**

This process of authentication and authorization of the request acts as a security measure to protect the API from malicious use, allow granular control over access to resources, and allows clients to access the API in a standard, simple and safe way.

### ***Requests to the API***

Clients request resources through the GET HTTP method, and submit resources through the POST HTTP method. As of this version of the API, other HTTP requests, such as PUT and DELETE are not accepted, and the updating or deletion of existing resources is not supported through the API.

GET requests are formatted inside of the URL. The structure of the URL for a standard GET request is as follows: <API URL>/<resource>/<parameters>. In the following documentation of HTTP requests “API URL” will equate to the base URL of the API (which will be included in all API requests), the “resource” will represent the resource being requested, and the “parameters” equate to the values submitted for this resource request. POST requests are different from GET requests in that they are formatted with a combination of the URL and the POST header of the HTTP request. The

resource is specified in the URL, but all other parameters are contained in the POST key-value array.<sup>22</sup>

### *Responses from the API*

For every HTTP request to the API, an HTTP response is returned. The response contains all of the standard HTTP response headers, as well as a formatted string in the response body. This string is a JSON object that contains an array of other objects which contain information about the response, the data requested (if it exists), pagination details and error details where appropriate. Below is a brief explanation of each of the objects returned in a JSON response.

#### *Metadata*

All responses include the “meta” object in them. The meta object represents information about the data returned from the API. This includes the HTTP response code in plain format, a response message to help understand the response, and the authentication status. The authentication status specifies whether or not the client application was authenticated using the API key and HTTP Authentication header. All requests return the meta object, even those that return errors.

#### **Example:**

```
“meta”: {  
  “response_code” : 200,  
  “response_message”: “Request OK”,  
  “authenticated” : “success”  
}
```

---

<sup>22</sup> The API online documentation provides detailed explanations and examples of every kind of request to assist client developers.

### *Error Message*

The “error\_message” object is returned in any request that results in an error, along with the meta object. The meta object’s properties should give a clue as to why there was an error, but the error\_message object will explicitly state what the error was.

Example:

```
“meta”: {  
  “response_code”: 400,  
  “response_message”: “Bad Request”,  
  “authenticated”: “success”  
}  
“error_message”: “The request was improperly formatted: item_id required”
```

### *Pagination*

All successful GET requests return the “pagination” object. The pagination object provides information about the number of results returned, as well as pagination details such as the results per page and the current page—which are useful to client applications utilizing requests with the pagination parameters. The pagination object will not be returned in GET requests that result in an error.

Example:

```
“pagination”: {  
  “current_results_count”: 10,  
  “total_results_count”: 50,  
  “results_per_page”: 10,  
  “current_page”: 3  
}
```

In the above example, the pagination object tells the client that there were 50 total results found based on the search, and that it is returning the third set of 10 results, being results 21 – 30. Utilizing pagination is an optional feature for the client, and if it is not used, the

API will simply return all of the results (up to 1000), along with the pagination object to explain the data set.

### *ID*

The “id” object is returned in all successful POST requests, which represents the identification number of the newly created resource. Returning this identification number is useful to the client because it allows the client to use this number in subsequent requests that will use the newly created resource. For instance, if the client wants to submit a sighting for an item that does not yet exist, the client first creates the item using a POST Item request, then uses the returned Item identification number in the subsequent request to create a sighting for that particular item.

### *Data*

The “data” object is returned on all successful GET requests that find at least one result for the client’s search request. The data object represents an entire resource object, and contains information pertaining to that resource. Because each resource is different, each request for a type of resource (Place, Item, Sighting, Category) will return a data response formatted to that resource type.

### **Example: Place:**

```
"data":[
  {
    "id":"281",
    "name":"Rosati's Pizza",
    "is_chain":1,
    "chain_id":157,
    "is_open":"1",
    "neighborhood":"West Westminster",
    "image_url":null,
    "phone":"3034647477",
    "restaurant_url":"http://www.myrosatis.com/home",
    "menu_url":"http://www.myrosatis.com/menu/store/default.asp?locationId=29",
  }
]
```

```

    "gluten_friendly":"1",
    "vegetarian_friendly":"1",
    "vegan_friendly":"0",
    "gluten_menu_url":null,
    "vegetarian_menu_url":null,
    "vegan_menu_url":null,
    "lat":"39.8775880",
    "lng":"-105.0934990",
    "street_address":"9960 Wadsworth Pkwy",
    "city":"Westminster",
    "state":"CO",
    "country":"US",
    "postal":"80021",
    "formatted_address":"9960 Wadsworth Pkwy, Westminster, CO 80021, USA",
    "distance":"0.304785156818786"}
  ]

```

**Example Item:**

```

"data":[
  {
    "id":"4",
    "name":"Orange Almond Muffins",
    "description":"Warm, doughy, goodness",
    "place_id":"7",
    "gluten_free":null,
    "vegetarian":null,
    "vegan":null,
    "exclusive_gluten_free":"0",
    "exclusive_vegetarian":"1",
    "exclusive_vegan":"0",
    "ingredient_list":null,
    "tag_healthy":"0",
    "meal_category":"breakfast",
    "verified":"0",
    "place_name":"Crumbles Bakery",
    "formatted_address":"Denver, CO, USA",
    "neighborhood":"",
    "g_lat":"39.7375670",
    "g_lng":"-104.9847179",
    "g_state":"CO",
    "g_postal":"80222",
    "g_city":"Denver",
    "distance":"11.2973433876467",
    "tm_id":"13",
    "tag_id":"6",
  }
]

```

```

    "tag_name":"pastry"}
]

```

### Example Sighting:

```

"data":[
  {
    "id":"1",
    "place_id":"150",
    "item_id":"8",
    "review":"Pretty good, but not the best French Onion soup around.",
    "image":null,
    "rating":"6",
    "date_created":"2012-11-15 16:32:29",
    "date_last_updated":null,
    "is_primary":"1",
    "place_name":"Jason's Deli",
    "item_name":"French Onion Soup"}
]

```

### Example Categories:

```

"data":[
  {"tag_id":"1","tag_name":"pizza"},
  {"tag_id":"2","tag_name":"italian"},
  etc...
]

```

The above examples (except for the Categories response) show a single result in the data object. While returning a single result is one type of scenario, the client will more often see a list of results within the data object. In this case, the data object will contain an array of results, each of which is its own array of objects as seen above.

### 3.5.5 Implementation of API

The API was implemented as an MVC (Model, View, Controller) PHP application that accepts HTTP requests and renders a JSON document as the body of the HTTP response. The model layer consists of several main database classes, each

representing one of the API's resources (Place, Item, Sighting, Category, Report, Application). Each of these database classes acts as an interface that the controller layer of the application uses to access the database. The controller layer consists of a PHP script that acts as a request dispatcher that routes requests to the appropriate sub-controller, which then performs the main business logic of the application. In the business logic, authorization to the resource is performed, and then the appropriate model class and methods are utilized in order to access the database and generate data. The controller then renders this data to the view layer, which renders the response in a JSON format.

### **3.6 Process Flow**

While the process of using the API consists of simple HTTP requests and responses, the API cannot be used without the client developer going through the prerequisite steps of viewing the documentation website and registering for access. The different components of the API, therefore, were designed to be utilized in an overall flow. The process flow below is the anticipated interaction of the client developer and client application with the API components.

First, the client developer must visit the API documentation website in order to learn about the API and request access to the API via the web form. The API administrator will grant access to the client developer where appropriate. After the client has access to the API, it is expected that the client developer will consult the API documentation website for information on how to use the API, and likely visit the sample client application in the documentation. After the developer has made the necessary updates, their application will call into the API to get or submit data. At this point the API acts as an intermediary between the client application and the central data repository

that stores the data. As an intermediary, the API enforces the necessary authentication of the client, validation of the request, accessing the data repository, and returning a formatted response to the client.



**Figure 9: Flowchart of API Access**

### 3.7 Summary

The development of the solution described in this chapter was the result of an analysis of the requirements for a solution, the design of the several major components that composed this solution, and the implementation of these designed components. An analysis of the needs of the proposed solution for this project resulted in a list of requirements, which then made it possible to design and implement components that would fulfill these requirements. While this chapter discussed the requirements, analysis, design and implementation of the solution, the following chapter will analyze how this implementation meets the needs discussed in the requirements, and how this acts as a solution to the problems discussed.



## **Part 4: Analysis of Results**

This section analyzes how the implementation of the API outlined in the previous section acts as a solution to the problem outlined in the thesis of this paper.

### **4.1 How the API meets the requirements for a solution**

To assess how the API meets the needs for a solution, this section will examine how each of the requirements for a solution have been fulfilled.<sup>23</sup>

#### **4.1.1 Data at Two Levels**

In order to be useful to a wide array of end-users and different types of client applications it was necessary to make the data model of the solution be as flexible and scalable as possible, while still maintaining a useful amount of specificity. Because of this, a requirement was made that the API will support data on two levels: the restaurant data model, and the menu item data model.<sup>24</sup> As detailed in section 3.5.3, the solution's data model was designed and implemented to accommodate the restaurant data model (which is used by many of the current dietary-need applications) as well as the menu item data model (which is in several recent applications in this genre).<sup>25</sup> This implementation includes separate entities for restaurants (the place entity), dishes (the item entity), experiences with items (the sighting entity), and problems with other entities (the report entity). Through this implementation, users are able to get data more specific and thus useful to them than solely restaurant reviews, and are able to see other users' experiences

---

<sup>23</sup> See sections 2 and 3 for the requirements outlined for a solution.

<sup>24</sup> See section 3.3.3 for this requirement in more depth.

<sup>25</sup> See section 1.4.2.3 for an example of a menu item data model application.

with the actual dishes that restaurants offer. Users and restaurant applications can consume and provide data relevant to a restaurant, and also consume and provide data relevant to dishes served at that restaurant.

#### **4.1.2 Allow Clients to Consume and Provide Data**

Through the RESTful API created in this project, client applications are able to consume and provide data to the central data repository through simple HTTP requests. A client may choose to only consume data and not provide it, or vice versa. All clients consuming and providing data must adhere to the policies of the API, and through the API, client applications are able to form an API community with a set of data that will continue to grow, thus providing an increasing set of data to consuming applications and end-users. Through the implementation of the RESTful interface that accepts simple HTTP requests to provide and consume data, the API fulfills the requirement as a solution to allow clients to consume and provide data.

#### **4.1.3 Client Support and Documentation**

The documentation website implemented as part of this project fulfills the requirement to inform and assist developers in using the API.<sup>26</sup> Through reading the documentation, client developers will be able to learn what the API does, why it exists, and all of the technical details of getting access and integrating their applications with the API. The documentation website will also serve as a place to inform developers of the API's terms and conditions of use, what kind of data the API expects, and what kind of data to expect in return from the API. Through the access request form, developers will

---

<sup>26</sup> See section 3.3.4 for more information on the requirement for developer support.

enter an agreement of understanding of what to expect from the API and the appropriate ways to interact with the API. As part of the documentation website, developers will also be able to use an example client application that utilizes the API.<sup>27</sup> This application provides developers with just one small example of how the API can be integrated into their own projects.

Informing the intended audience on how to use the API is a vital part of its success. The documentation website fulfills the need for an informational gateway that will educate and assist developers, and demonstrate some of its functionality to those interested in utilizing it.

#### **4.1.4 Scale to Current and Future Dietary Needs**

While the current version of the API uses only the gluten-free, vegetarian and vegan dietary types, the data model was implemented in such a way as to allow the future addition of other dietary types as well. When the time comes to add another dietary type to the API, a field will be added to the item entity to represent this new property, and the client documentation will be updated to reflect this addition. In this way, the API fulfills the requirement to be easily scalable for future dietary types.<sup>28</sup> This method of adding new dietary types is both simple for the administrator and also allows the addition to be controlled solely by the API administrator.

---

<sup>27</sup> See section 3.5.2 for more information about the sample client application.

<sup>28</sup> See section 3.3.2 for more information on the requirement to be scalable for future dietary needs.

#### **4.1.5 API: Equipped to Enforce Policies of Usage**

As an intermediary between the client and data, the API is equipped to enforce policies of usage. The following security and usage policies act as a means to protect the API and the central data repository.

##### ***4.1.5.1 Security***

Security is enforced on two levels: access, and data integrity. On the access layer, the client application must be authorized to access the API, and then authorized to have access to resources. On the data layer, all submitted data is cleaned and validated before accessing the data repository.

##### ***Authentication***

In order to ensure the controlled and safe usage of the API, all requests made by client applications are authenticated upon reaching the API. By assigning a unique key to each individual client application, the API is able to authenticate the application, thus allowing the application's usage of the API to be monitored and controlled. Client developers are expected to keep their application's API key hidden from the user in their application's interactions with the API. If it is determined that an application is misusing the API, the API administrator can revoke this API key's privileges.

##### ***Authorization***

After an application is authenticated to use the API, the API then enforces an authorization scheme for access to all resources. The total number of requests, the types of requests, and the resources that may be requested are all controlled through the API's authorization measures. The default maximum number of requests for a 24-hour period is 1000 requests. Client developers needing a higher maximum number of requests can

arrange this with the API administrator. The default is to also allow all new client applications to have GET and POST requests for all of the API's resources. However, these accesses may be modified for individual applications where appropriate.

### *Validation*

It is expected that the client developer will enforce a validation scheme at the application level in order to ensure harmful data is not submitted to the API. However, because this cannot be ensured, the API performs validation upon all data. The client developer, however, should seek to validate data before making a request to the API, in order to not waste API requests and resources, and in order to be more user-friendly.

The API validates all submitted data before inserting it into the database. This means that numerical and string fields are enforced, and fields such as URL or phone number are validated to be a properly formatted URL or phone number. Required fields are also enforced. If a submission is made without the required fields populated properly, the API will not insert the data, and will return an error response. If optional fields are left blank or are improperly formatted (not passing validation) they are simply not inserted into the database, and the API continues to process the other fields. Required fields and the expected and appropriate data types are specified in the client documentation.

### *Character Escaping*

It is possible that some users of the API may attempt to submit malicious data to the API. In order to protect the data repository and combat security issues such as SQL injection, the API escapes and reformats all data before performing any interactions with the database. This step of data sanitation prevents malicious attacks upon the API's data.

Through the authentication and authorization of client applications, and through the cleansing and validation of all incoming data, the solution fulfills the security requirements needed to protect the API and its data.<sup>29</sup>

#### ***4.1.5.2 Usage Limitations and User-submitted Data***

As mentioned before, client developers must agree to the terms and conditions of the API before submitting a request for access to the API on the documentation website. In this documentation, developers agree to the understanding that usage will be limited to 1000 requests per day unless arranged otherwise. Developers must also agree that the data consumed from the API must not be cached or stored in their servers for increments longer than four hours, and that data submitted to the API must not contain inappropriate, vulgar, or harassing language. Failure to abide by these rules results in the suspension or termination of that developer's account.

As part of the goal for many applications to use the API, it is also the goal that this community of applications and end-users will actively participate in enforcing the overall integrity of the data. To facilitate this participation, and to enforce these policies, the API was designed to allow each resource to be “flagged” as either inappropriate or inaccurate. If a resource is flagged as inappropriate or inaccurate, the API administrator will review the validity of this claim and potentially delete the data and take measures with the application in order to prevent this from happening again.

---

<sup>29</sup> See section 3.3.5 for the security requirements.

These measures that were taken to control the usage of data and to allow the community to take an active part in the monitoring and control of user-submitted data fulfill the requirement to establish data-usage policies.<sup>30</sup>

#### **4.2 How the API Is A Solution to the Problem of Data Insularity in This Field of Applications**

The API and the central data repository created in this project are a proposed constructivist solution to the problem of data insularity in the field of dietary need restaurant applications. Requirements were introduced earlier in this paper in order to define certain tangible properties that the proposed solution must possess. As discussed in this section, the project that was designed and implemented in this paper fulfills each of the requirements for a solution.

The data model and central data repository function as the base of the solution, defining how data will be used amongst applications in the industry, and allowing the future scalability of the data as well. The API functions as the gateway to the data, supplying client applications with a secure and simple RESTful interface with which to interact with the data. The client documentation website and sample client application inform the developer of the API, establish a contract of understanding of what the API is and how it is to be used, and encourages the use of the API through the sample client application. Together, these different components form a cohesive solution to the

---

<sup>30</sup> See section 3.3.5 for more information on the requirement to establish data usage policies.

problem of data insularity by allowing client developers to easily create or integrate their applications to utilize community-driven and useful data.

### **4.3 Project History**

#### **4.3.1 How the Project Began**

The concept for this project began two years ago when the author initially realized how fragmented the data of dietary-need restaurant applications was. A friend of the author with Celiac disease was using three different applications to find a place to eat dinner with her family. This person used two different gluten-free restaurant-finder applications in order to find restaurants that were listed as gluten-friendly. Two applications were needed because each had their own sets of data with different restaurants in them. After a restaurant that was gluten-free-friendly was chosen, the person then used a third application (not dietary-specific) to cross-reference this restaurant with reviews of dishes available because the all of the finder applications did not have data at the dish level. This showed the author that the entire data set available to the end-user was fragmented, and that this fragmentation negatively affected the end user.

#### **4.3.2 How the Project Was Managed**

The author was the sole developer for this project, and a rapid evolutionary prototyping development process was used.<sup>31</sup> Larger goals were set for each iteration of the project in order to drive development at a steady pace. For the most part, these goals were adhered to for the entire process.

---

<sup>31</sup> See section 3.4 for a detailed explanation of the software development process used.



### **4.3.3 Project Milestones**

While each iteration was its own milestone, there were several major milestones of the project. The first major milestone was the completion of the requirements phase, during which requirements were defined for the solution. The second milestone was the completion of the design of the data model and API. The third milestone was the implementation of the data model, API, and client documentation website.

### **4.3.4 Changes to the Project Plan**

There were not any major changes to the project plan. The biggest delay in the project was the result of the sample client application taking longer to develop than originally planned.

### **4.3.5 Project Timeline**

The project was completed along with this paper in certain general phases.

#### ***Phase 1: December 2011 – April 2012***

The author initially conceived this project and conducted initial research in the areas of restaurant and meal review applications, dietary-need applications, as well as public RESTful APIs. This phase concluded with the refinement of the project's thesis.

#### ***Phase 2: May 2012 – August 2012***

During this phase, the author conducted detailed research on the field of dietary-need applications, as well as APIs in the food and restaurant industry. Chapters 1 and 2 of the thesis were developed during this phase, which culminated in a project plan.

#### ***Phase 3: September 2012 – January 2013***

During this phase of the project, the author adhered to a project life cycle, developing the project's solution in increments.

#### *Phase 4: February 2013 – April 2013*

The author finished writing this paper and made refinements to the components developed as part of this project.

#### **4.3.6 Ethical and Social Impact of the Project**

While the solution introduced in this paper is in its infancy in terms of its adoption by other developers and its evolution as a product, as stated in section 1.7, it is the hope of the author that the API will eventually be adopted by outside developers, and that this project will eventually assist end users with special dietary needs.

## **Part 5: Conclusions and Lessons Learned**

### **5.1 Conclusions**

This project has been a constructivist prototype solution for a problem in a particular field of applications. In this paper, the author has made the following overall inferences.

#### **5.1.1 The Problem of Data Insularity**

The paper introduced the problem of data insularity amongst dietary need applications in the restaurant industry. The problem was discussed in the context of the intended functionality of this genre of applications. Data insularity is a trait that occurs when the available applications in an area of user-generated content do not make their data available to other applications.

#### **5.1.2 Data Insularity Causes Incomplete and Fragmented Data**

The paper discusses the implications of the problem of data insularity. In the case of user-review dietary applications in the restaurant industry, data insularity is problematic because it presents the end user with a fragmented set of data; a user must check multiple applications in order to get a complete set of all of the available data.

#### **5.1.3 A Prototype Solution Is Needed**

The paper proposed that a prototype solution to the problem of data insularity amongst this group of applications was needed, and that a solution will benefit the end users by making it possible for applications to provide a more complete set of data. In other words, the solution to the problem will allow these isolated sets of data to be used amongst other communities of users.

#### **5.1.4 Prototype RESTful API as a Solution**

The author proposed a prototype RESTful API as the solution to the problem of data insularity. After a brief introduction to RESTful APIs, the paper examined several related APIs in the restaurant industry, and culminated the knowledge gained to define a set of core requirements for the API to be a successful solution.

#### **5.1.5 The Project Successfully Satisfies These Requirements**

The paper then examined how the design and implementation of the project fulfilled the requirements defined for a successful solution. Each requirement was looked at in order to determine that the project's goal was a success: to create a prototype solution to the problem of data insularity in this field of applications.

### **5.2 Summary of Contributions**

#### **5.2.1 The Documentation of a Prototype Solution to Data Insularity**

This paper documents in detail the process of creating a prototype solution to the problem of data insularity amongst this group of applications. The author uses knowledge gained from the investigation of data insularity in this field of applications and potential solutions to explicitly define several core requirements for a solution to the problem. With these requirements, the paper then documents in detail the design of the solution, and discusses the implementation of these designs.

#### **5.2.2 An Analysis of Potential Solutions to the Problem of Data Insularity**

The project offers an analysis of several existing solutions to the problem of data insularity amongst dietary need applications in the restaurant industry. A detailed analysis of several industry APIs reveals that there is not currently an existing solution appropriate for this problem.

### **5.2.3 A Brief Review of Applications in the Field that Represent the Problem of Data Insularity**

To introduce the problem of data insularity the paper initially examines several popular dietary need applications in the restaurant industry. While there are many applications in this genre, three popular applications were chosen as representatives in the areas of gluten-free, vegan, and vegetarian dietary needs, as well as a gluten-free application that attempts to provide data at the dish, rather than restaurant, level. Through this review, the problem of data insularity amongst these applications is revealed.

### **5.3 Lessons Learned**

While there were not any serious problems while working on this project, the most notable difficulty was that some goals took longer to accomplish than originally planned.

The first setback occurred when the author chose to change the language and framework in which the API was to be developed. The author originally developed part of the API in PHP without a framework, and later discovered an appropriate framework that would aid development (called the Slim PHP Framework). While this initially sacrificed about a month's worth of development, the amount of effort saved in the long-term made up for this lost time.

The second setback occurred while developing the sample client application. The author began developing the sample client application without adequately defining the scope. Because the scope of the client application was not adequately defined, the focus changed several times during its development of this development, thus causing it to take more time to finish than originally expected.

In order to ensure that these lessons will be implemented, the author has resolved to spend more initial time researching existing technologies that can aid development, and to always firmly define the scope of every facet of a project.

#### **5.4 Recommendations / Future Research**

While this project acts as a prototype solution to the problem of data insularity in the field of dietary-need applications in the restaurant industry, there are additions to this solution that can be made by future researchers.

Detailed food allergy and dish ingredients can be included in order to supply the end-user with more specific data for dishes. This can benefit users that have severe food allergies, such as peanut, soy or other allergies. The challenge of including food allergies is that only the restaurant proprietor can be trusted to have that detailed information, rather than an end-user casually enjoying the dish. Therefore, in order for food allergies to be included, the API must first allow restaurant-guaranteed items.

Restaurant-guaranteed items (items that are added by the proprietors of restaurants) can provide the data with a level of detail and trustworthiness that the user community cannot. While the user community will ideally strive for accuracy and detail, only the restaurant will know of the exact ingredients of a dish, and thus only items added by the restaurant can be guaranteed to be completely accurate.

### Sources

API 2.0: Business API. (n.d.). Yelp for Developers. Retrieved June 5, 2012, from

<http://www.yelp.com/developers/documentation/v2/business>

API Terms of Use. (n.d.). Yelp for Developers. Retrieved June 5, 2012, from

[http://www.yelp.com/developers/getting\\_started/api\\_terms](http://www.yelp.com/developers/getting_started/api_terms)

API 2.0: Search API. (n.d.). Yelp for Developers. Retrieved June 5, 2012, from

[http://www.yelp.com/developers/documentation/v2/search\\_api](http://www.yelp.com/developers/documentation/v2/search_api)

Category List. (n.d.). Yelp for Developers. Retrieved July 2, 2012, from

[http://www.yelp.com/developers/documentation/category\\_list](http://www.yelp.com/developers/documentation/category_list)

DuVander, A. (2011, September 12). The Dish on Foodspotting's Semi-Public API.

ProgrammableWeb.com - Web 2.0 API Reference Guide. Retrieved July 1, 2012,  
from <http://blog.programmableweb.com/2011/09/12/the-dish-on-foodspottings-semi-public-api/>

Dish Freely. (2012). Grande Labs (Version 1.4) [Mobile application software]. Retrieved

from <http://itunes.apple.com/app/apple-store/id375380948?mt=8>

DuVander, A. (2012, February 6). 5,000 APIs: Facebook, Google and Twitter Are

- Changing the Web. ProgrammableWeb.com - Web 2.0 API Reference Guide.  
Retrieved August 29, 2012, from  
<http://blog.programmableweb.com/2012/02/06/5000-apis-facebook-google-and-twitter-are-changing-the-web/>
- DuVander, A. (2012, August 23). 7,000 APIs: Twice as Many as This Time Last Year. ProgrammableWeb.com - Web 2.0 API Reference Guide. Retrieved August 29, 2012, from <http://blog.programmableweb.com/2012/08/23/7000-apis-twice-as-many-as-this-time-last-year/>
- Farrell, S.; , "API Keys to the Kingdom," *Internet Computing, IEEE* , vol.13, no.5, pp.91-93, Sept.-Oct. 2009. Doi: 10.1109/MIC.2009.100
- Fielding, Roy. 2000. *Architectural Styles and the Design of Network-Based Software Architectures*. Ph.D. Dissertation. University of California, Irvine. AAI9980887
- Foodspotting | API Documentation. (n.d.). Foodspotting - Find and recommend dishes, not just restaurants.. Retrieved September 5, 2012, from  
<http://www.foodspotting.com/api>
- Foodspotting | Terms of Service. (n.d.). Foodspotting - Find and recommend dishes, not just restaurants.. Retrieved July 3, 2012, from  
<http://www.foodspotting.com/terms/>



Jacobson, D., Brail, G., & Woods, D. (2012). *APIs a strategy guide*. Sebastopol: O'Reilly.

Meng, J., Mei, S., Yan, Z.; , "RESTful Web Services: A Solution for Distributed Data Integration," *Computational Intelligence and Software Engineering, 2009. CiSE 2009. International Conference on* , vol., no., pp.1-4, 11-13 Dec. 2009 doi: 10.1109/CISE.2009.5365234

Rob, P., & Coronel, C. (2009). *Database systems: design, implementation, and management* (8th ed.). Cambridge, Massachusetts: Thomson/Course Technology.

Vaswani, V. (2010, May 11). Implement SOAP services with the Zend Framework. IBM - Developer Works. Retrieved August 31, 2012, from <http://www.ibm.com/developerworks/webservices/library/x-zsoap/index.html?ca=drs->

Vinoski, S; , "REST Eye for the SOA Guy," *Internet Computing, IEEE* , vol.11, no.1, pp.82-84, Jan.-Feb. 2007 doi: 10.1109/MIC.2007.22  
URL: <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4061127&isnumber=4061105>