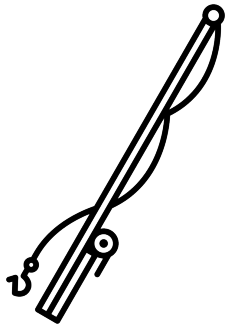

BENEFISHER

Software Design Specification

Version 2.0

10/14/2014

TEAM WAKATI

TABLE OF CONTENTS

1. INTRODUCTION

This document is the Software Design Specification for the Benefisher project, by Team Wakati, sponsored by Code for Sacramento.

Project Team

Team Wakati is comprised of undergraduate students majoring in Computer Science at California State University, Sacramento. The team members are enrolled in a two-semester senior project course required of all undergraduate majors. Successful delivery of the desired software product will fulfill the senior project requirement for the student team members.

Name	Email	Phone
Adrian Chambers	adr510909@gmail.com	(707) 430-3775
Anthony Cristiano	cristiano@csus.edu	(925) 321-7648
James Doan	jhdoan@gmail.com	(949) 690-4212
Daniel Green	djgreensolving@gmail.com	(209) 402-3658
Jesse Rosato	jesse.rosato@gmail.com	(916) 541-5386

Table 1.1 Team Wakati Members

Project Sponsor

Code for Sacramento is a Code for America Brigade, whose mission is "to help government work for the people, by the people" [1]. Code4Sac has aligned the goal of improving access to public services data with their central missions of “connecting citizens and governments to design better services” and “open[ing] civic data” [1]. To that end, the sponsor has proposed the project under discussion.

Name	Role	Email
Brandon Pugh	Brigade Captain	bpugh143@gmail.com
Ash Roughani	Community Organizer	ash@publicinnovation.org

Table 1.2 Code for America Representatives

1.1 Purpose

This Software Design Specification (SDS) exists to establish a baseline for the technical design of Benefisher. This document is primarily a blueprint for Team Wakati to use in implementing the project, but should also provide the project sponsor with insight into Benefisher's technical design for use in later maintenance of the project.

1.2 Scope

This document details the design of Benefisher from multiple perspectives: architectural design, interface design, data design, and component design. This document also includes a performance analysis, resource estimates, and a traceability matrix that ties the contents of this document to the project requirements established in the Software Requirements Specification (SRS) [2].

This is the baseline design specification for Benefisher, and the design described here may change during implementation and testing. Changes should follow the baseline change process described in the Project Charter [3].

1.3 Definitions

Term	Definition
Angular	An open-source web application framework that assists with creating single-page applications, which consists of one HTML page and CSS and JavaScript on the client side [4].
Application Programming Interface (API)	Implemented declarations of how a software component will interact with other software components. A common example of an API is a web service that provides data via a collection of resource addresses.
Client-side	The portion of code in a client-server application that is executed on the client machine.
Continuous Delivery	Continuous delivery (also called continuous integration) is a modern software deployment strategy that allows the source code for a system to be updated causing system downtime.
Express	A Node.js web application framework, designed for building single-page, multi-page and hybrid web applications.
Git	A popular version control system.
GitHub	A web based hosting service for Git repositories
Hyper Text Transfer Protocol (HTTP)	HTTP is one of the foundational protocols of the web, and is generally used to retrieve hypertext from a website (e.g. http://facebook.com). HTTP provides several 'verbs' for making requests, including 'GET' and 'POST', that indicate the nature of the request.
Jade	A templating language and engine focused on enabling quick HTML coding for Node applications [5].
JavaScript	A client-side language for implementing dynamic interactions on a webpage.
JetBrains WebStorm	An Integrated Development Environment (IDE) designed for writing web software languages.
MySQL	An open-source relational database management system.
Neural Network	A computational model and machine learning algorithm with multiple connected input and output units. Each connection is weighted in importance and the weights are adjusted by the algorithm to correctly predict future inputs to outputs [10].
Node	An open-source, cross-platform runtime environment for server-side and networking applications written in JavaScript [6].
Open Eligibility Project (OEP)	An open-source taxonomy to categorize human services and human situations [7].
Representational State Transfer (REST)	A network-based architectural style that “define[s] a uniform connector interface” [8]. In modern practice, REST principles are commonly used to guide design decisions when building APIs for web services.

Term	Definition
RESTful	Jargon to describe software implementations that closely adhere to REST principles.
Server-side	The portion of code in a client-server application that is executed on the server hardware.
SQL Injection	A code injection technique used to attack data-driven applications in which malicious SQL statements are inserted into an entry field for execution.

Table 1.3 Definitions

1.4 References

1. "What we do", [online], Code for Sacramento, http://code4sac.org/what-we-do/, [Oct. 24, 2014]

2. Software Requirements Specification. Chambers, A., Cristiano, A., Doan, J., Green, D., and Rosato, J., Team Wakati, Sacramento, CA, May 1, 2014.

3. Team Wakati Project Charter. Chambers, A., Cristiano, A., Doan, J., Green, D., and Rosato, J., Team Wakati, Sacramento, CA, Mar. 13, 2014.

4. “What Is Angular?”, [online], AngularJS, https://docs.angularjs.org/guide/introduction, [Sept. 23, 2014]

5. “Language Reference”, [online], http://jade-lang.com/reference/, [Sept. 24, 2014]

6. “About Node.js”, [online], http://nodejs.org/about/, [Sept. 23, 2014]

7. “Open Eligibility Project”, [online], http://openeligibility.org/, [Apr. 14, 2014]

8. Fielding, R., “Architectural Styles and the Design of Network-based Software Architectures”, Ph.D., ICS, Univ. California, Irvine, 2000.

9. “JavaScript Style Guide”, [online], jQuery Foundation, https://contribute.jquery.org/style-guide/js/, [Ap. 14, 2014].

10. Han, Jiawei, and Micheline Kamber. *Data Mining: Concepts and Techniques*. Haryana, India ; Burlington, MA: Elsevier, 2012. Print.

11. Segaran, Toby, "Learning from Clicks" in *Programming Collective Intelligence,* 1st ed. Sebastopol, CA. August, 2007. Print. [Oct. 15, 2014]

1.5 Overview of Content of Document

Architectural Design

Architectural design describes the hardware, software, and their interplay within the Benefisher application. The architectural design of Benefisher will include the communication between the interface, server, and data stores.

Interface Design

Interface design establishes the look, feel, and interactivity of the software from a user's perspective. This includes a mapping of Benefisher's features to the functionality they provide to the user, and images from the application depicting those features.

Data Design

The data design represents how Benefisher will store and obtain the data it needs to work. Included is a high level model of where the data will be stored, as well as the SQL statements needed to create the database.

Component Design Specifications

This section details the design of the software and maps the features to their components.

Performance Analysis

This section specifies the implementation of quality attributes detailed in the SRS, as well as any design constraints that have come about as a result of the design and implementation process.

Resource Estimates

This section describes the minimum and recommended system requirements to successfully run Benefisher.

Software Requirements Traceability Matrix

The matrix relates the design components to their associated requirements by listing the subsection numbers in this document and the associated subsection numbers in the SRS.

Approvals

The signatures here indicate that the relevant stakeholders understand the contents of this document, and approve of the outlined software design.

2. ARCHITECTURAL DESIGN

Architectural design describes the hardware, software, and their interplay within the Benefisher application. The architectural design of Benefisher will include communication between the user interface, server, and data stores.

2.1 Hardware architecture

Clients will be able to connect to the Benefisher application using a desktop computer, laptop computer, mobile phone, or tablet. Clients must be connected to the Internet on the device that is being used in order to use the Benefisher application. Benefisher will use the client's location to pull information on the social service providers near them from the Code for Sacramento data source. Information to make the search stronger will be pulled from the Benefisher Statistics Database and this will be combined with the information pulled from the Code for Sacramento database.

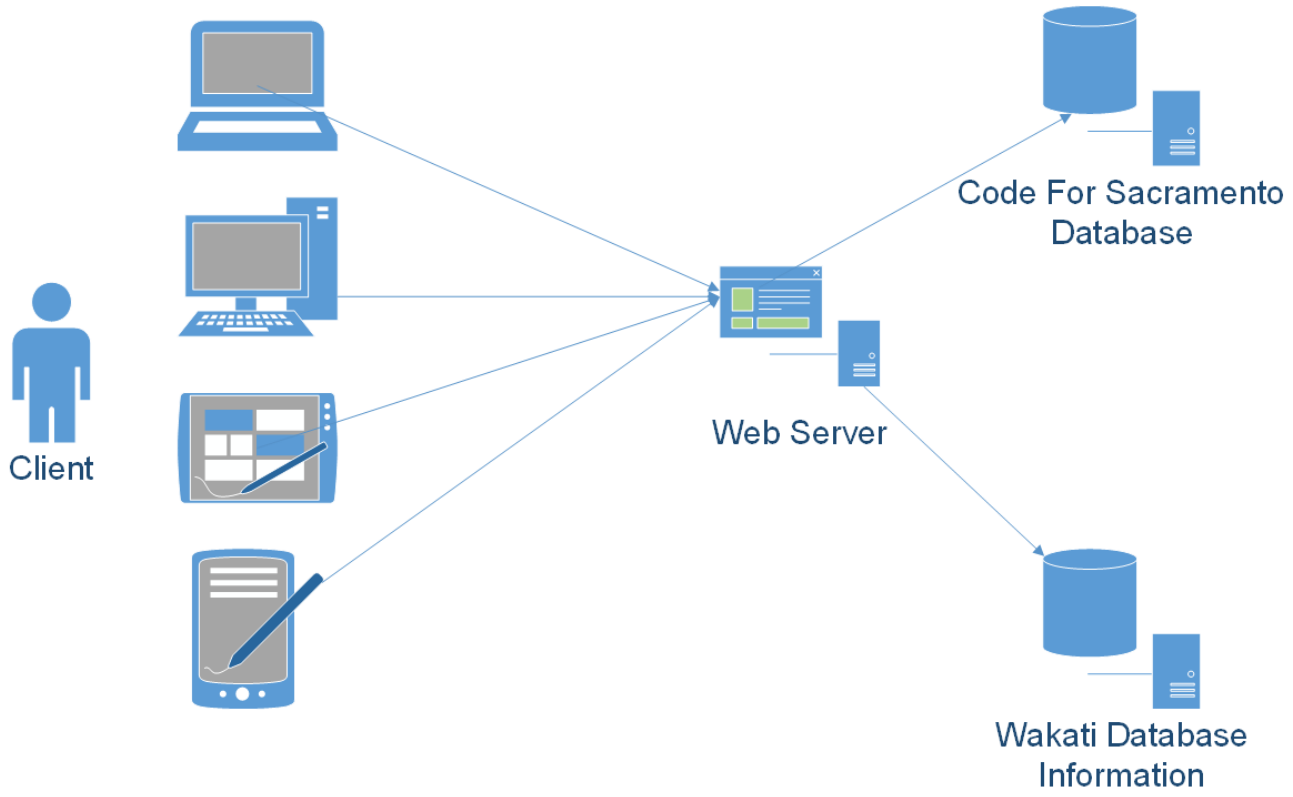

Figure 2.1 Architecture Design

2.2 Software Design Architecture

The software design architecture establishes the high-level software structure of the Benefisher application. This section describes the various web frameworks, design patterns, and components used in the design of the Benefisher application. Also included is a brief discussion on the rationale for choosing the selected elements that comprise the software architecture.

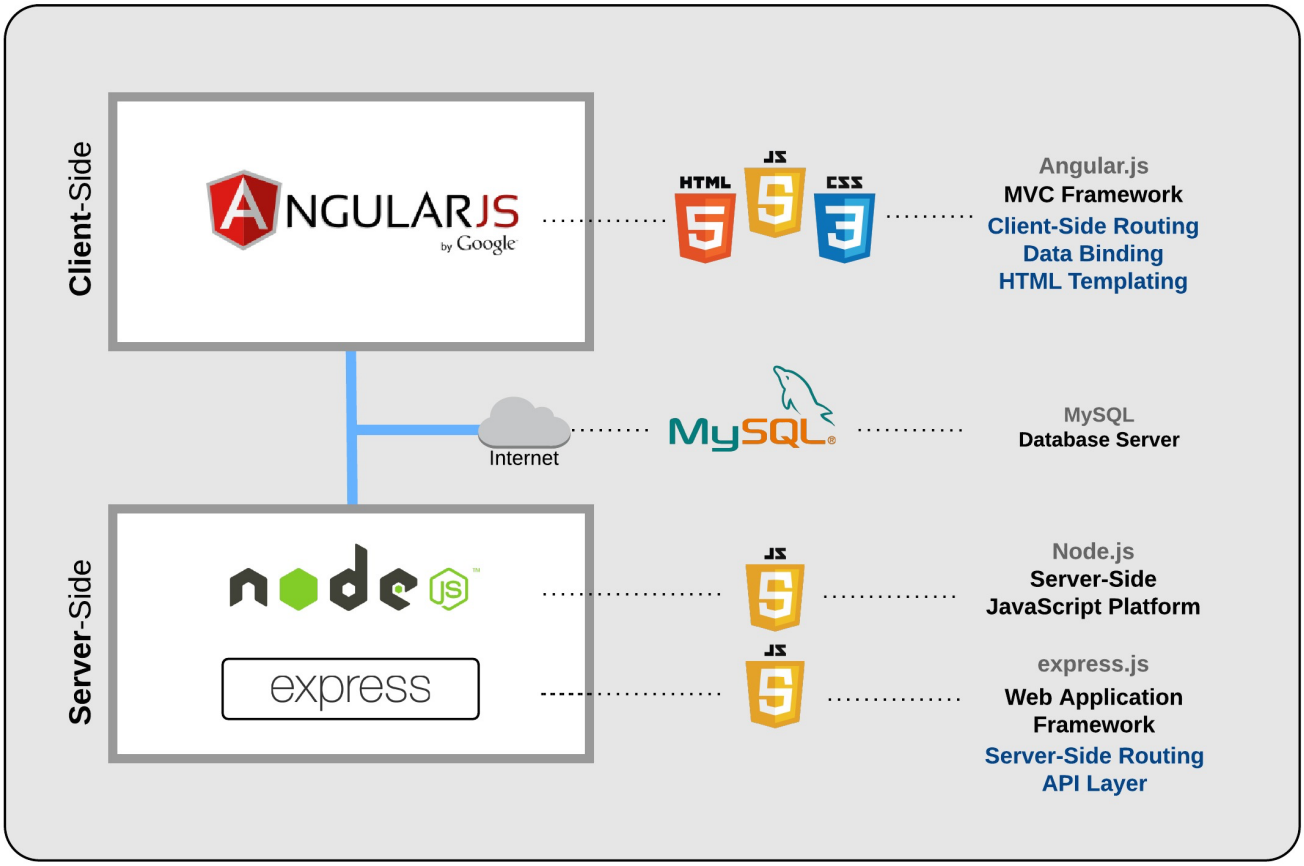

Figure 2.2 Software Development Stack

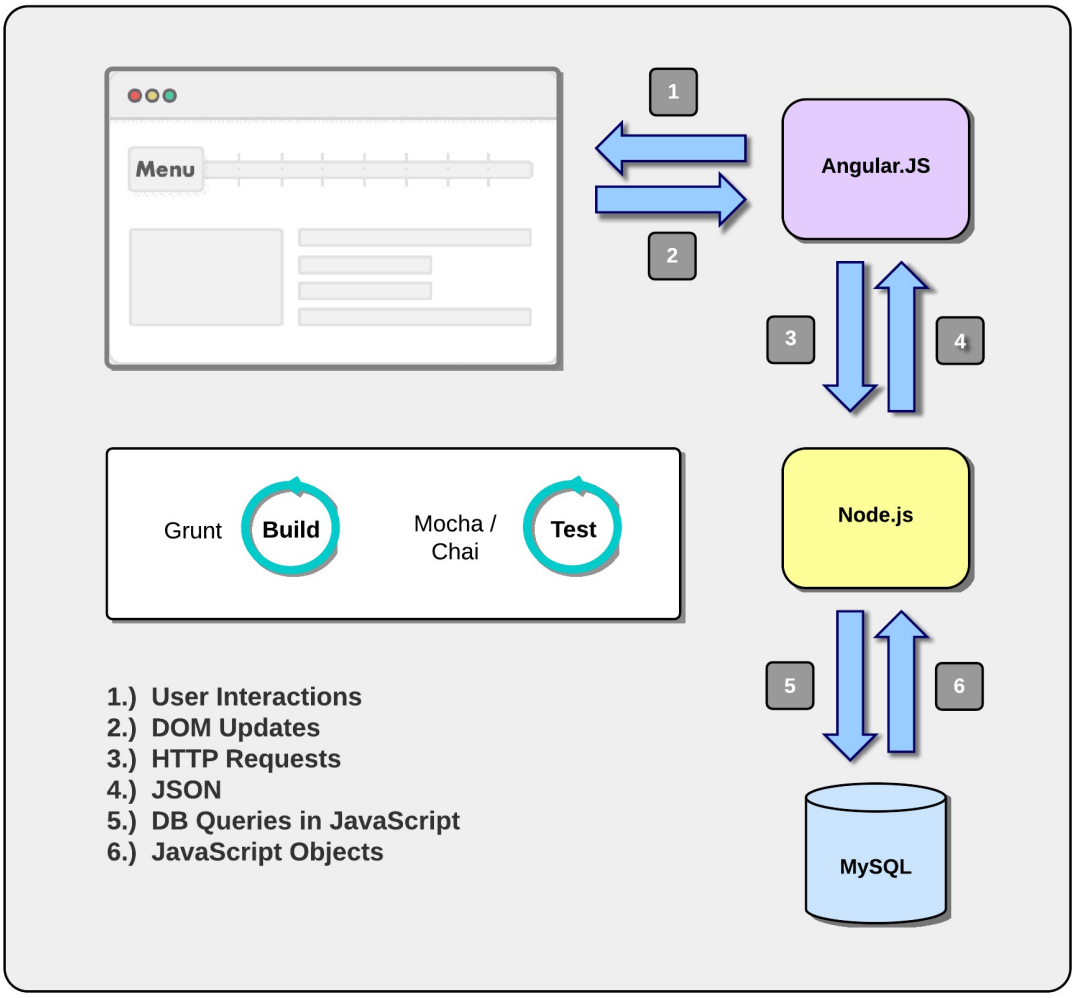

Figure 2.3 Full-Stack Data Flow Diagram

Presentation Layer

In the presentation layer, the client-side of the Benefisher application employs the Model-View-Controller architectural pattern for implementing the user interface. Angular JS was chosen as the framework to accomplish this because of its abilities to be highly testable, utilize data binding, and attach directives to webpages using attributes, tags and expressions. It provides all the functionality needed to handle user input in the browser, manipulate data and control how elements are displayed in the browser view to fulfill the application's client-side requirements.

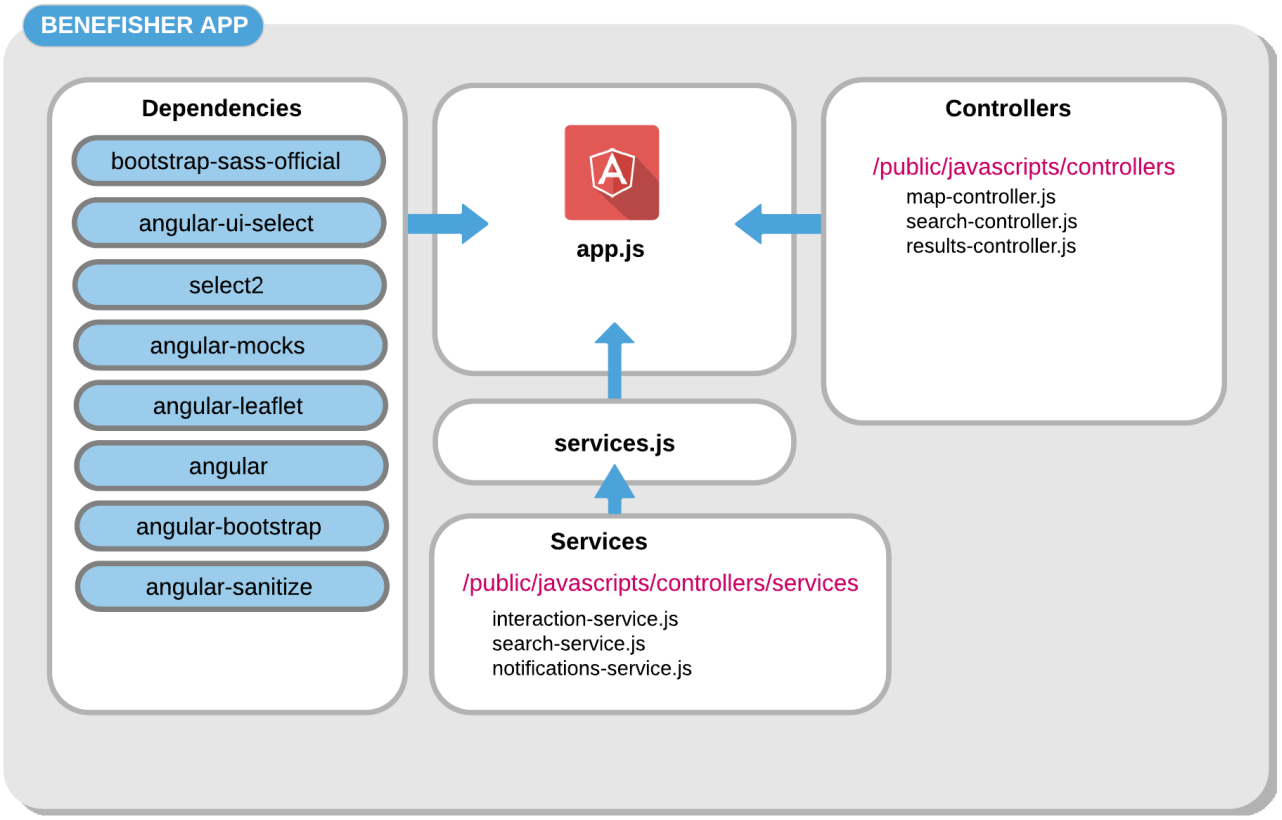

Figure 2.4 Client-Side Architechture

“Business” Logic Layer

The business logic layer components that reside on the server-side of the Benefisher application are developed using the Node.js framework. The Node.js framework was chosen to accomplish this because of its high scalability, event-driven architecture, ease of implementation, and because it allows the development team to code the client- and server-sides of the application in the same language (JavaScript). The server-side components utilize a RESTful approach to provide data resources to client-side components. The advantages of using RESTful APIs are their loose coupling and future extensibility.

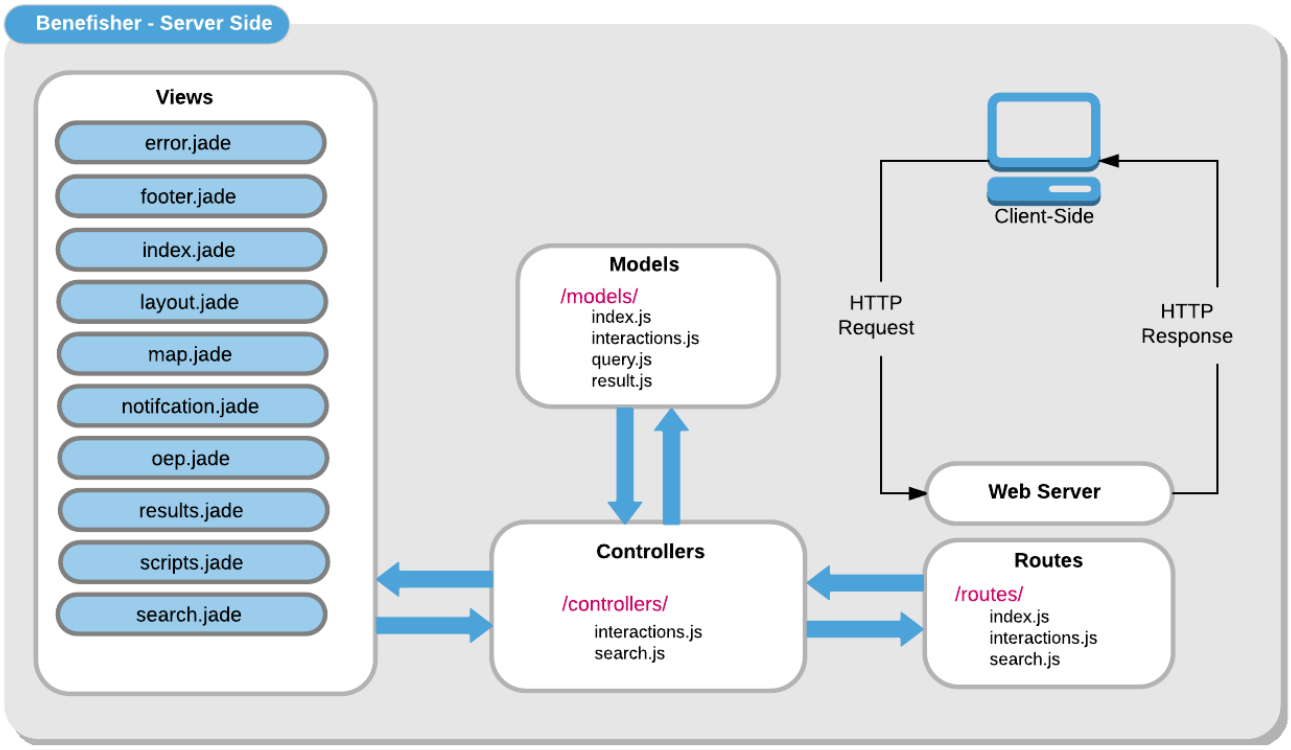

Figure 2.5 Server-Side MVC Architechture

Data Management Layer

The data management layer will utilize a MySQL database to store the data for the Benefisher application. The database will communicate with the server-side components of the business logic layer and will be use to store data of available services and search analytics and statistics.

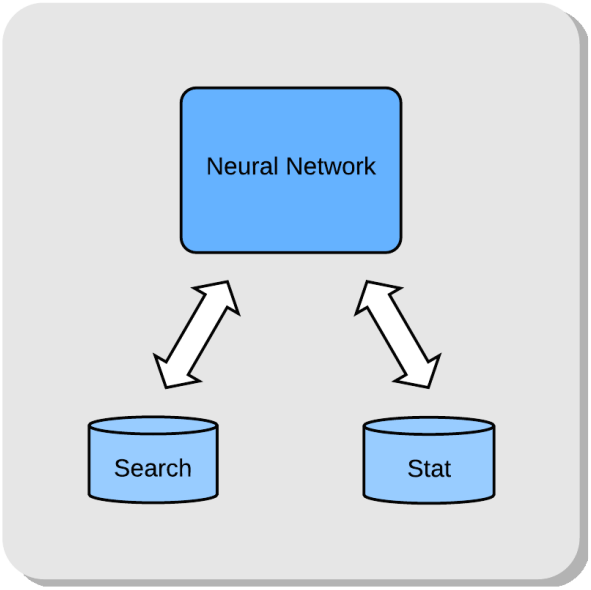

Figure 2.6 Data-Layer Component Architecture

3. INTERFACE DESIGN

Interface design establishes the look, feel, and interactivity of the software from a user's perspective. This includes a mapping of Benefisher's features to the functionality they provide to the user, and images from the application depicting those features.

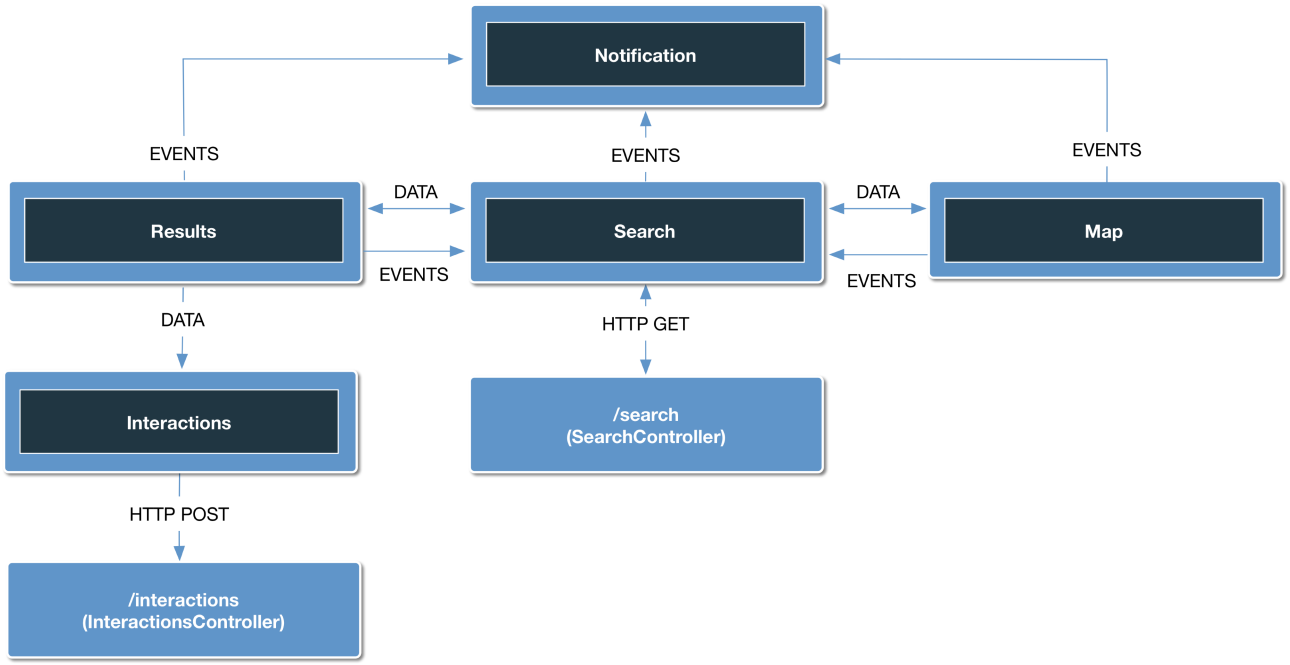

Figure 3.1 User Interface Data-Flow Diagram

Map

Entering text into the search component causes relevant markers to appear on the map. Moving the map causes the markers on the map to be updated with markers relevant to the new bounds of the map viewport. Clicking on markers within the map causes the relevant result in the results component to be highlighted.

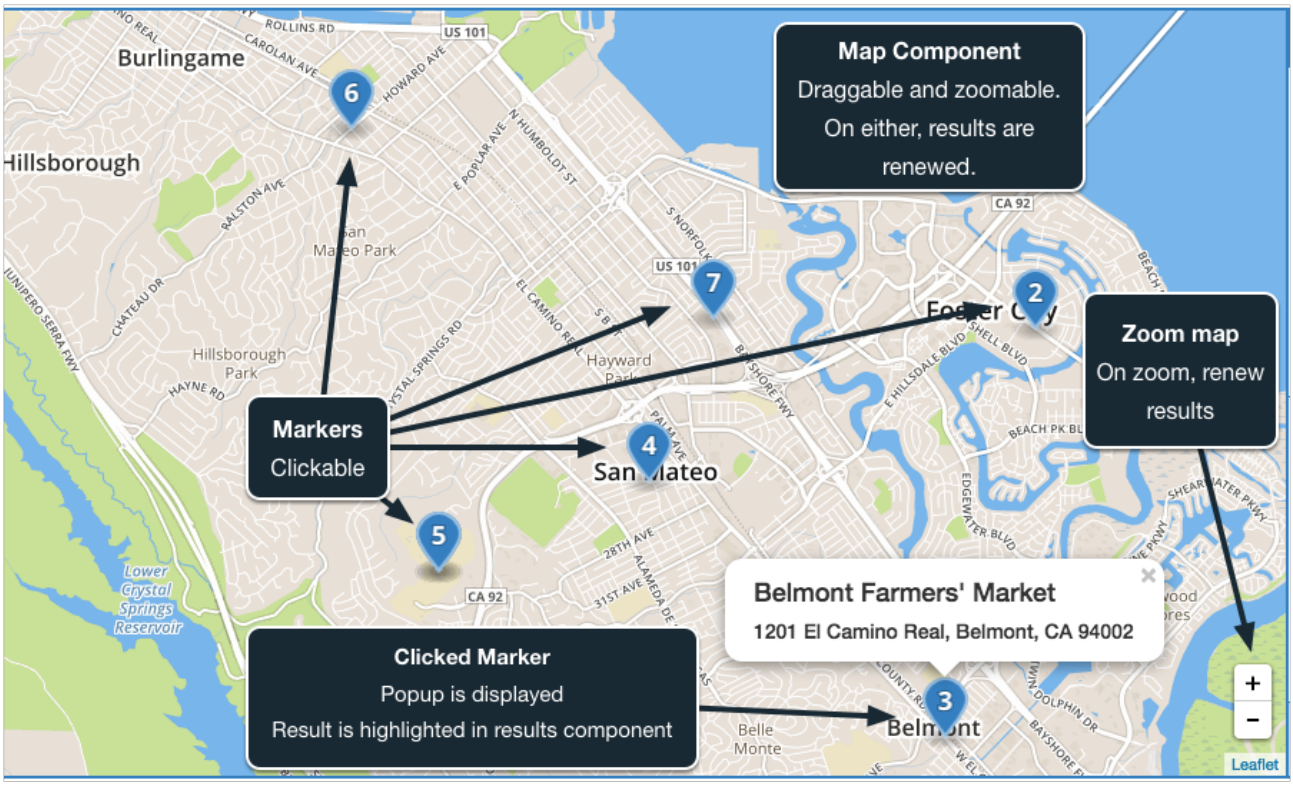

Figure 3.2 Map Component Interactivity

Search

The search component accepts query parameters, saves queries to the statistics database, and retrieves search results from the server. Typing text into the inputs causes a dropdown list of OEP [7] terms to be displayed that the user can select. Selecting a term, or 'panning' the map, causes the component to make an HTTP request to the server to retrieve search results. The search results are sorted for relevance by a neural network that intermediates between the server and the data source. The results are then displayed in the map and results components.

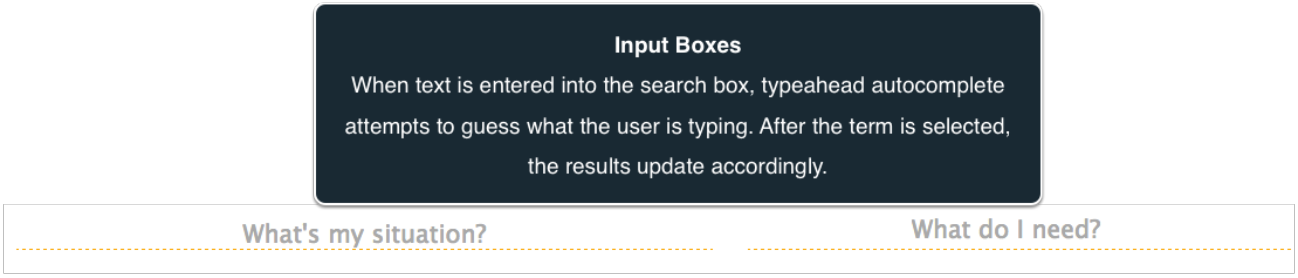

Figure 3.3 Search Component Interactivity

Results

The results component displays more detailed information about the locations displayed on the map. When a user interacts with a result, by emailing the service described by the result, visiting the service's website, hiding a result, etc., that interaction is saved to the statistics database, and the display is updated if necessary.

Figure 3.4 Results Component Interactivity

4. DATA DESIGN

The data design represents how Benefisher will store and obtain the data it needs to work. Included is a high level model of how the data will be stored, as well as the SQL statements needed to create the database.

4.1 Entity Relationship Diagram

The following is a simplified version of the application's Entity Relationship Diagram. See Appendix A for a more complete version.

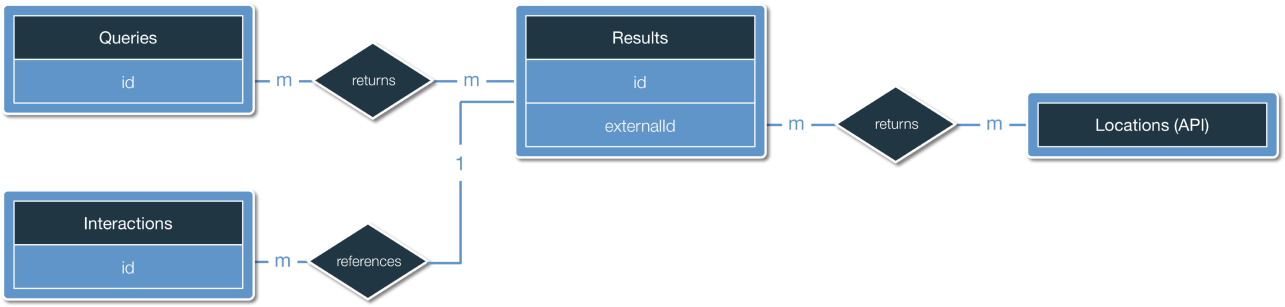

Figure 4.1 Simplified Entity Relationship Diagram

4.2 Creating the Database

This section contains the statements for creating the database.

SQL Table: Search Query:

```
CREATE TABLE IF NOT EXISTS `Queries` (
      `id` INTEGER NOT NULL auto_increment ,
      `bounds` VARCHAR(255),
      `terms` VARCHAR(255),
      `userPostalCode` VARCHAR(255),
      `createdAt` DATETIME NOT NULL,
      `updatedAt` DATETIME NOT NULL,
      PRIMARY KEY (`id`)
) ENGINE=InnoDB;
```

SQL Table: Search Result:

```
CREATE TABLE IF NOT EXISTS `Results` (
      `id` INTEGER NOT NULL auto_increment ,
      `name` VARCHAR(255),
      `externalId` VARCHAR(255),
      `lat` DECIMAL(18,12),
      `lng` DECIMAL(18,12),
      `description` TEXT,
      `address` VARCHAR(255),
      `hours` VARCHAR(255),
      `phone` VARCHAR(255),
      `rawPhone` VARCHAR(255),
      `email` VARCHAR(255),
      `url` VARCHAR(255),
      `createdAt` DATETIME NOT NULL,
      `updatedAt` DATETIME NOT NULL,
      PRIMARY KEY (`id`)
) ENGINE=InnoDB;
```

SQL Table: Search Result Interaction:

```
CREATE TABLE IF NOT EXISTS `Interactions` (
      `id` INTEGER NOT NULL auto_increment ,
      `target` VARCHAR(255), `createdAt` DATETIME NOT NULL,
      `updatedAt` DATETIME NOT NULL,
      `ResultId` INTEGER,
      PRIMARY KEY (`id`),
      FOREIGN KEY (`ResultId`)
            REFERENCES `Results` (`id`) ON DELETE SET NULL ON UPDATE CASCADE
) ENGINE=InnoDB;
```

SQL Table: Query-Result Relationship:

```
CREATE TABLE IF NOT EXISTS `QueriesResults` (
      `createdAt` DATETIME NOT NULL,
      `updatedAt` DATETIME NOT NULL,
      `ResultId` INTEGER , `QueryId` INTEGER ,
      PRIMARY KEY (`ResultId`, `QueryId`),
      FOREIGN KEY (`ResultId`)
            REFERENCES `Results` (`id`) ON DELETE CASCADE ON UPDATE CASCADE,
      FOREIGN KEY (`QueryId`)
            REFERENCES `Queries` (`id`) ON DELETE CASCADE ON UPDATE CASCADE
) ENGINE=InnoDB;
```

5. COMPONENT DESIGN SPECIFICATION

This section details the design of the software and maps the features to their components. In implementing these components, Team Wakati will be adhering to the jQuery Foundation's coding style [9], as described in the Software Requirements Specification (SRS) document [2].

Feature	View	Client	Server	Table/Relations
Map	layout.jade index.jade scripts.jade map.jade notification.jade	app.js services.js map-controller.js search-service.js notification-service.js	app.js index.js search.js neural-network.js	Results Queries
Search	layout.jade index.jade scripts.jade search.jade	app.js services.js search-controller.js search-service.js	app.js index.js search.js neural-network.js	Results Queries
Results	layout.jade index.jade scripts.jade results.jade notifcation.jade	app.js services.js results-controller.js search-service.js interaction-service.js notification-service.js	app.js index.js interactions.js	Interactions
Notification	layout.jade index.jade scripts.jade notifcation.jade	app.js notification-service.js	app.js index.js	

Table 5.1 Feature-Component Matrix

5.1 Sequence Diagrams

These sequence diagrams are used to illustrate the Use Cases outlined in the SRS[1], and clarify the interaction of Benefisher's components. These diagrams depict the ideal, or 'happy', flow.

Search for Services (UC1)

Features: Search, Map, Results

This functionality is used by several other Use Cases (UC2 – UC7), and is not duplicated in their sequence diagrams. The 'subscribers' depicted here are the MapController and ResultController. When a search is executed, both of those components are updated with the results of the search. This Use Case always triggers a save of the search query (UC17).

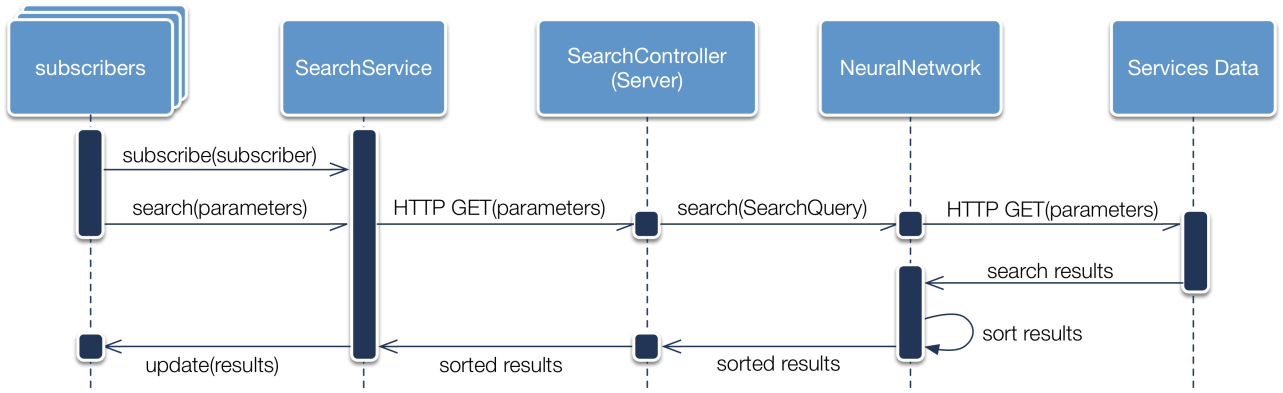

Figure 5.1 Search sequence Diagram

Search by Need (UC2)

Features: Search

This Use Case is triggered by a user selecting at least one predefined need (UC6), and triggers a search for services (UC1).

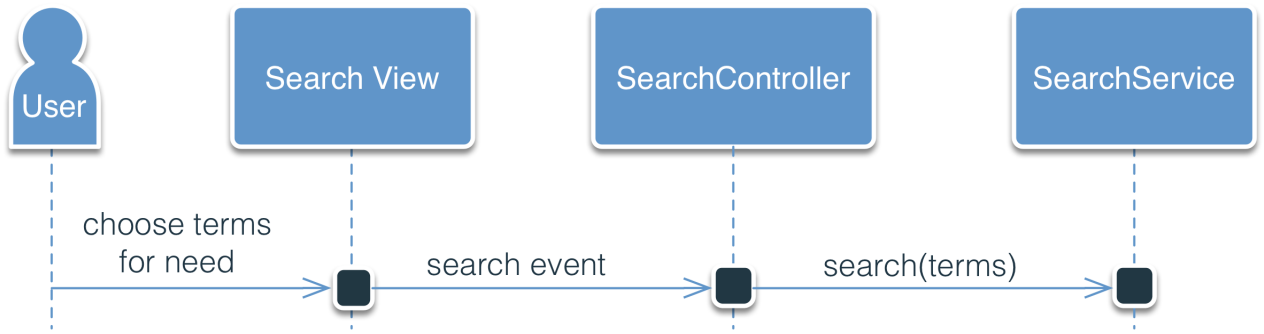

Figure 5.2 Search by Need Sequence Diagram

Search by Situation (UC3)

Features: Search

This Use Case is triggered by a user selecting at least one predefined situation (UC7), and triggers a search for services (UC1).

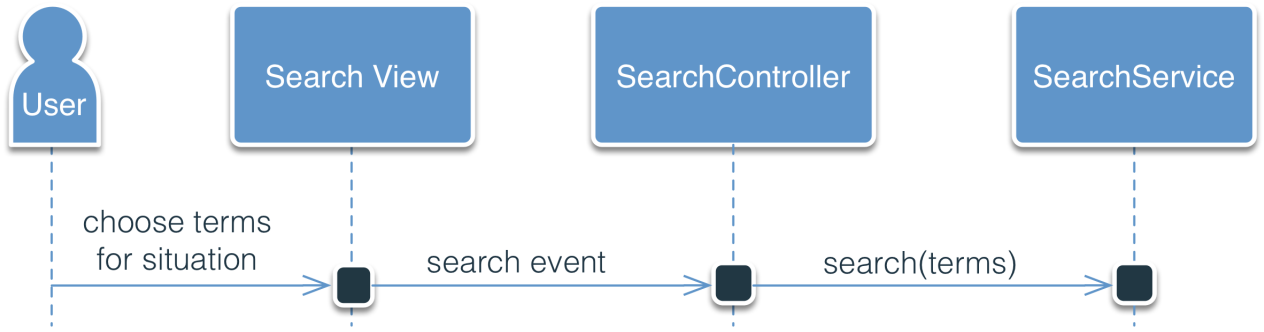

Figure 5.3 Search by Situation Sequence Diagram

Search by Need and Situation (UC4)

Features: Search

This Use Case is triggered when a user selects at least one predefined need and at least one predefined situation (UC6 and UC7). This Use Case triggers a search for services (UC1).

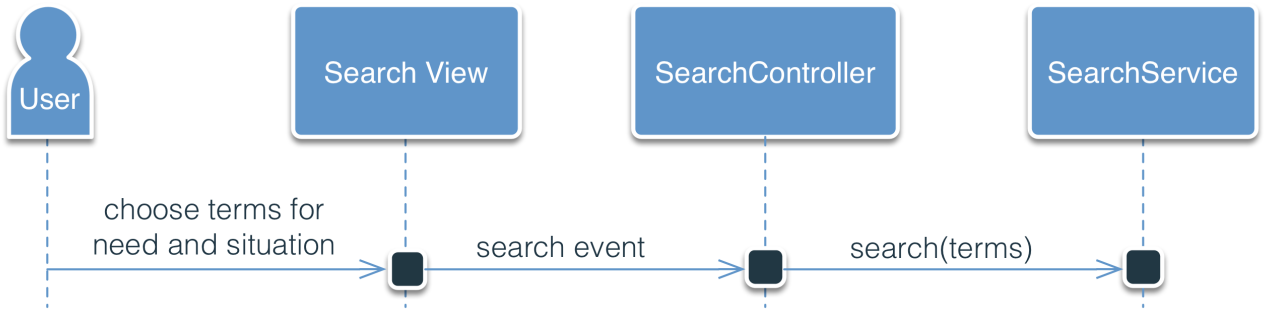

Figure 5.4 Search by Need and Situation Sequence Diagram

Browse Map (UC5)

Features: Map, Search, Results

This Use Case triggers a search for services (UC1).

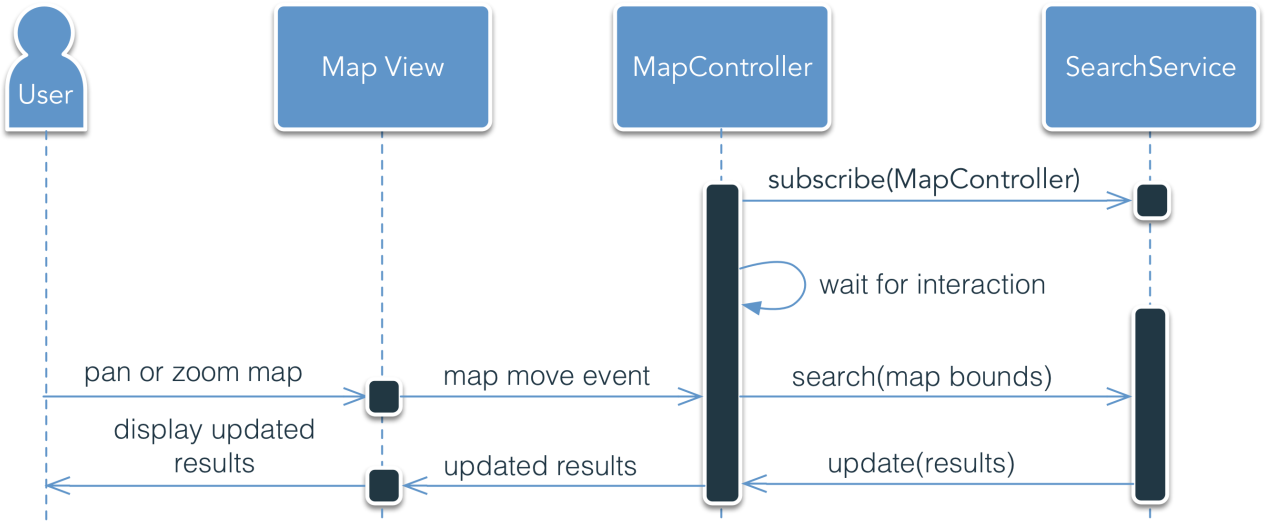

Figure 5.5 Browse Map Sequence Diagram

Select Predefined Need (UC6)

Features: Search

This Use Case triggers a search by need (UC2), or a search by need and situation (UC4) if a predefined situation is also selected (UC7).

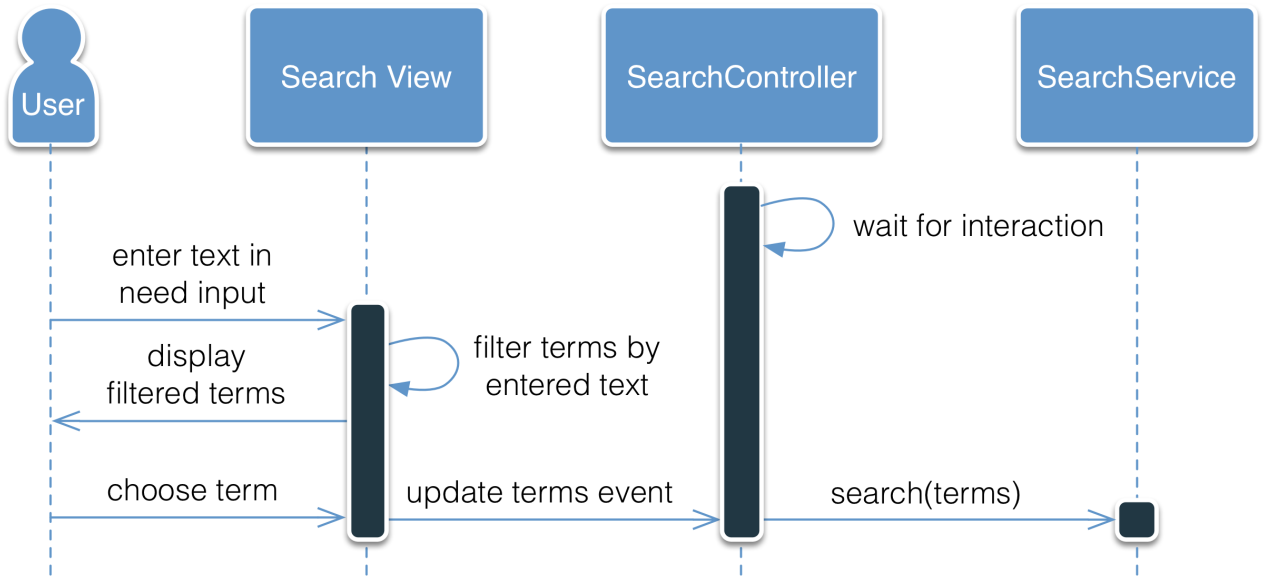

Figure 5.6 Select Predefined Need Sequence Diagram

Select Predefined Situation (UC7)

Features: Search

This Use Case triggers a search by situation (UC3), or a search by need and situation (UC4) if a predefined need is also selected (UC6).

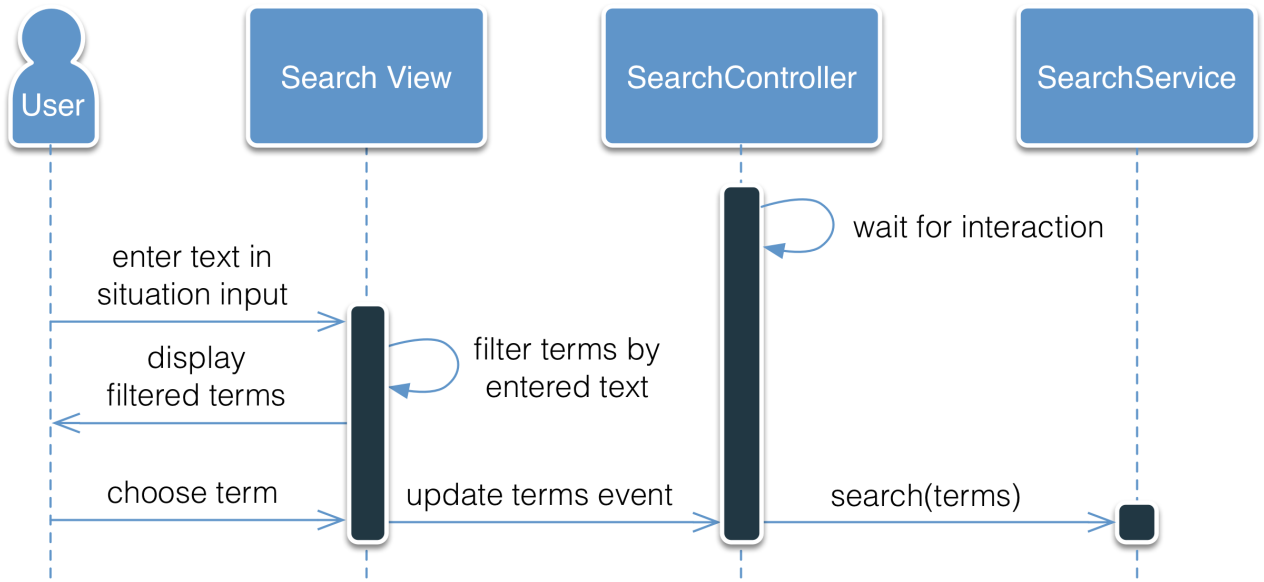

Figure 5.7 Select Predefined Situation Sequence Diagram

Save Interaction Data (UC8)

Features: Results

This Use Case saves data on user interactions with results to be used by the neural network algorithm to present users with the most relevant search results (UC1).

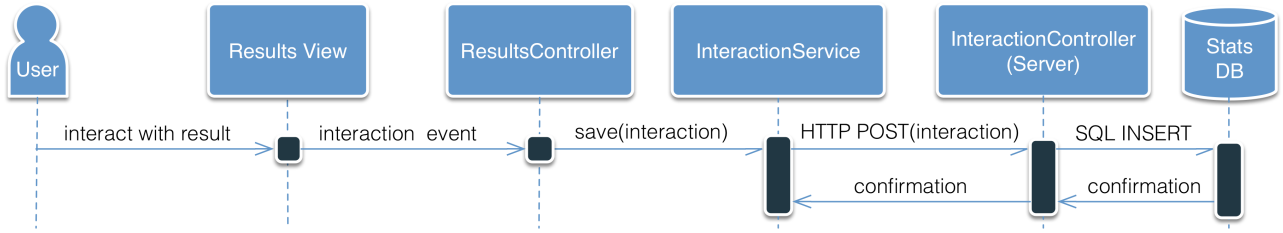

Figure 5.8 Save Interaction Data Sequence Diagram

Interact with Results (UC9)

Features: Results, Map

The application provides a number of ways for users to interact with search results. These interactions are described in more detail in other Use Cases (UC10 - UC15), and all of these interactions trigger a save of the interaction (UC8). No sequence diagram is provided for this Use Case because its events are specific to the individual interactions.

Expand Result (UC10)

Features: Results

Results are provided in a compact state by default, to provide users with an overview of their available options. Results may then be expanded to provide more details about a particular service. This Use Case triggers a save of the interaction (UC8).

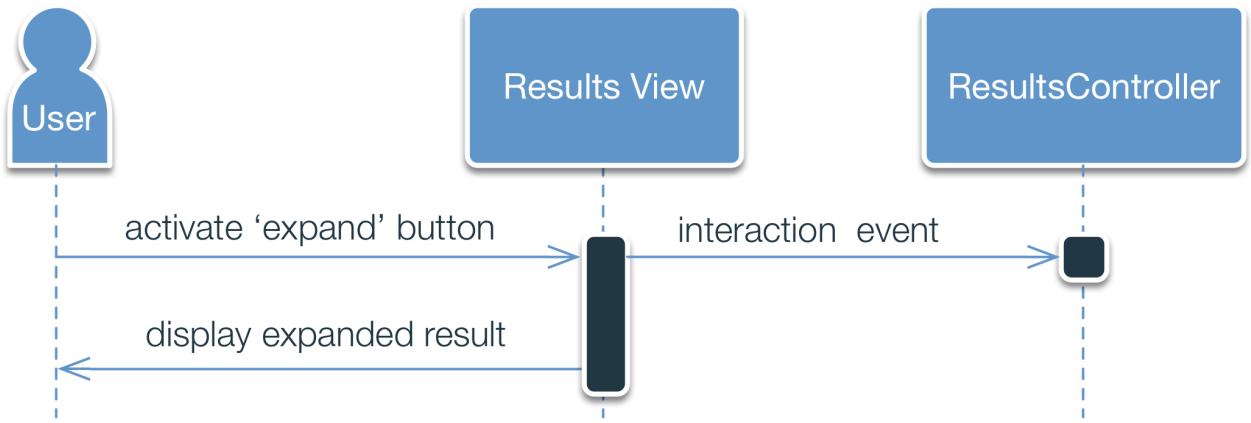

Figure 5.10 Expand Result Sequence Diagram

Get Directions (UC11)

Features: Results

This Use Case redirects users to the Google Maps web application using a URL that informs Google Maps to provide the user with directions to the service from the user's current location. This Use Case triggers a save of the interaction (UC8).

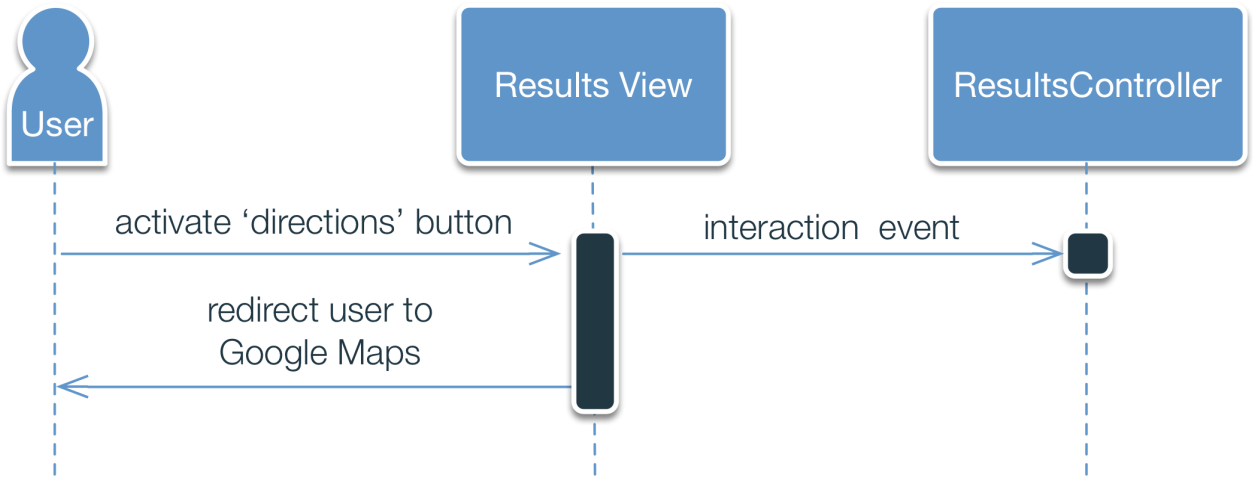

Figure 5.11 Get Directions Sequence Diagram

Call Service (UC12)

Features: Results

This Use Case redirects a mobile user to their device's phone application, and initiates a phione call to the service. This Use Case triggers a save of the interaction (UC8).

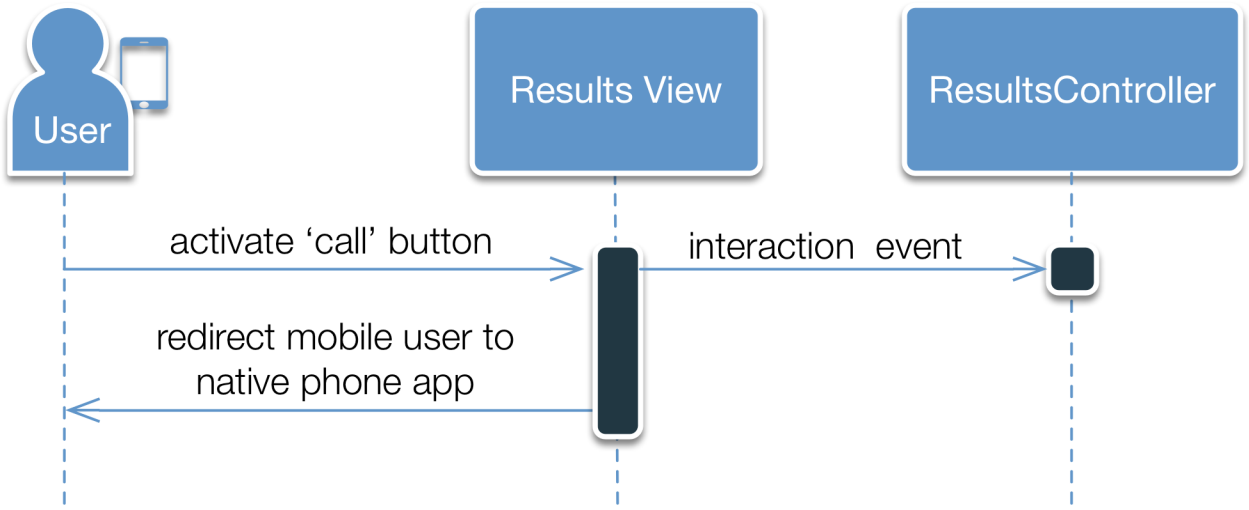

Figure 5.12 Call Service Sequence Diagram

Navigate to Site (UC13)

Features: Results

This Use Case redirects users to the service's web site. This Use Case triggers a save of the interaction (UC8).

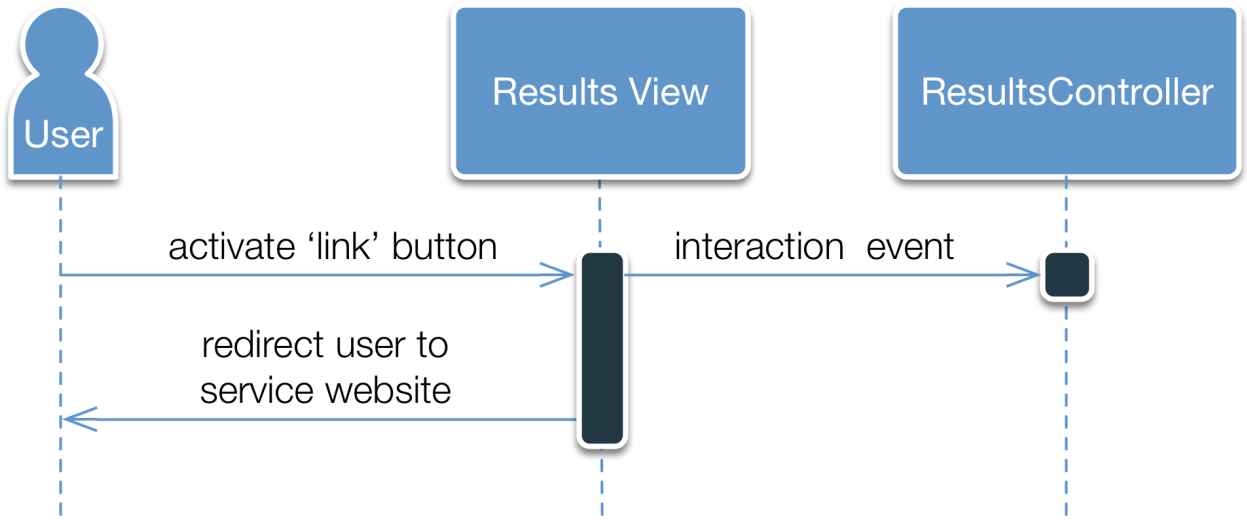

Figure 5.13 Navigate to Service Sequence Diagram

Email Service (UC14)

Features: Results

This Use Case creates a new email using the user's system's default email application, with the service's email address in the 'to' address of the email. This Use Case triggers a save of the interaction (UC8).

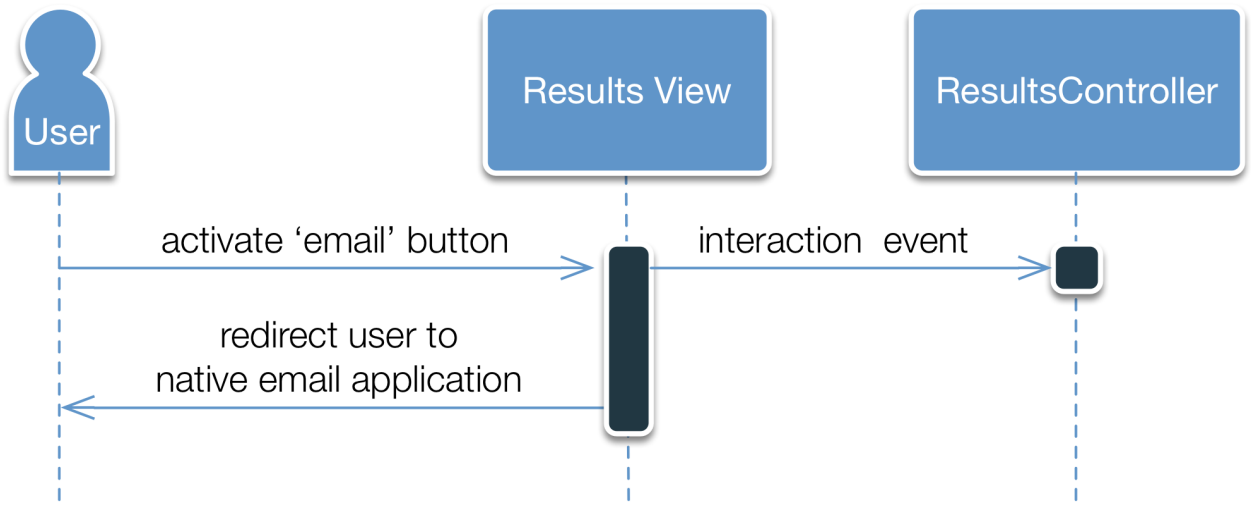

Figure 5.14 Email Service Sequence Diagram

Remove from Screen (UC15)

Features: Results, Map, Search

This Use Case allows a user to ignore an unwanted result. This Use Case triggers a save of the interaction (UC8).

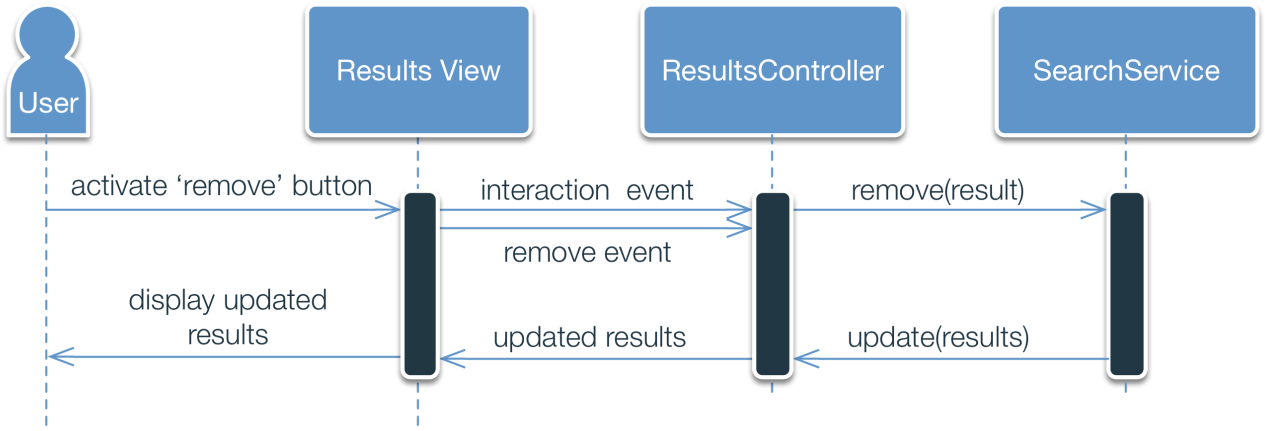

Figure 5.15 Remove from Screen Sequence Diagram

Click Result on Map (UC16)

Features: Results, Map, Search

This Use Case highlights a result when the user clicks its corresponding map marker.

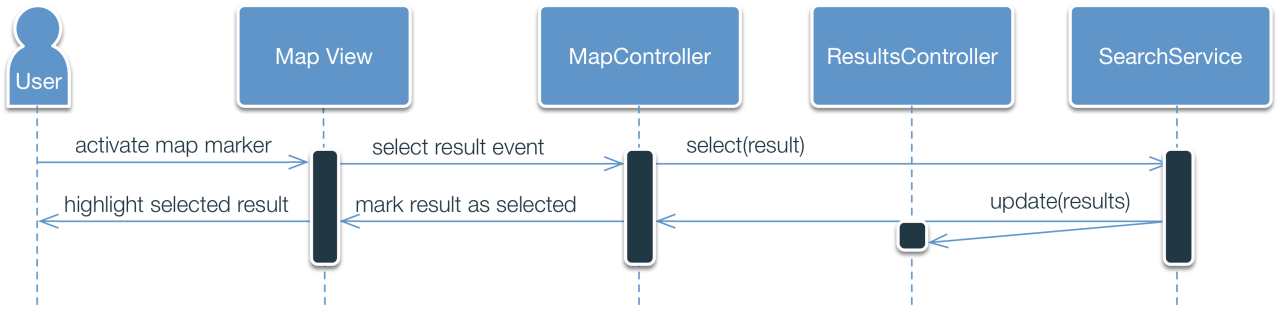

Figure 5.16 Click Result on Map Sequence Diagram

Save Search Query (UC17)

Features: Search

This Use Case saves a search query to be used by the neural network algorithm to present users with the most relevant search results (UC1). This use case is triggered by a search for services (UC1).

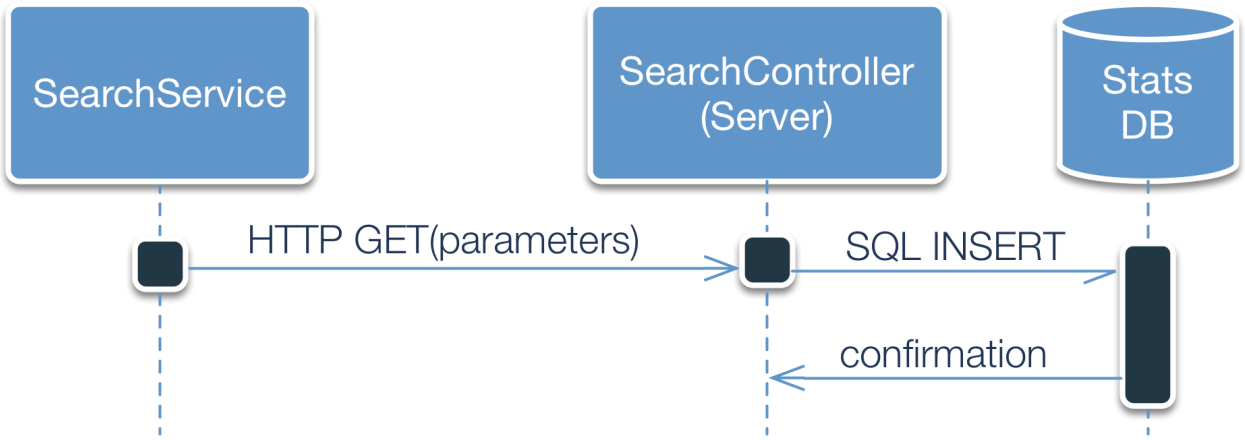

Figure 5.17 Save Search Query Sequence Diagram

5.2 Webpage and Function Design Specifications

This subsection describes the preconditions, interface specifications, processing specifications, database requirements, and postconditions of each component. View components also include a screen print of the component.

1. File: app.js (client)

Preconditions: This component begins the page initialization sequence depicted in Figure 3.1.

Interface specifications: This component does not provide an interface.

Processing specifications: This component loads the Angular modules and dependencies necessary for the remaining client-side components.

Screen print: None.

Database requirements: None.

Postconditions: User leaves the webpage.

2. File: app.js (server)

Preconditions: This component bootstraps the application. It is the first component loaded when the server is started.

Interface specifications: This component does not provide an interface.

Processing specifications: This component loads the critical core modules required to run the Benefisher application and establish the necessary server routes. These include routes for index and search. The core modules include: express.js server, path, favicon, logger, cookie-parser and body-parser. It also starts the web application by creating an Express server instance. This main component also includes functions for handling server errors, such as 404 and 500 errors.

Screen print: None.

Database requirements: None.

Postconditions: User leaves the webpage.

3. File: error.jade

Preconditions: This component is present after an error has been encountered while executing *app.js*.

Interface specifications: When a user attempts to access the Benefisher application and a server error is encountered, this file will be rendered and the user will be presented with an error page (Figure 5.18).

Processing specifications: None.

Screen print:

Figure 5.18 The *error.jade* View

Database Requirements. None.

Postconditions. User leaves the webpage.

4. File: index.jade

Preconditions: This component is a sub-template of *layout.jade*, and is rendered by *index.js* after a user has accessed the application. See *layout.jade* (Figure 5.19) for a depiction of template relationships.

Interface specifications: When a user visits the page, this file loads the visible components of the Benefisher Application, including: *map.jade*, *notification.jade*, *results.jade*, and *search.jade*. This file also loads the scripts required for dynamic application interactions (*scripts.jade*).

Processing specifications: None.

Screen print:

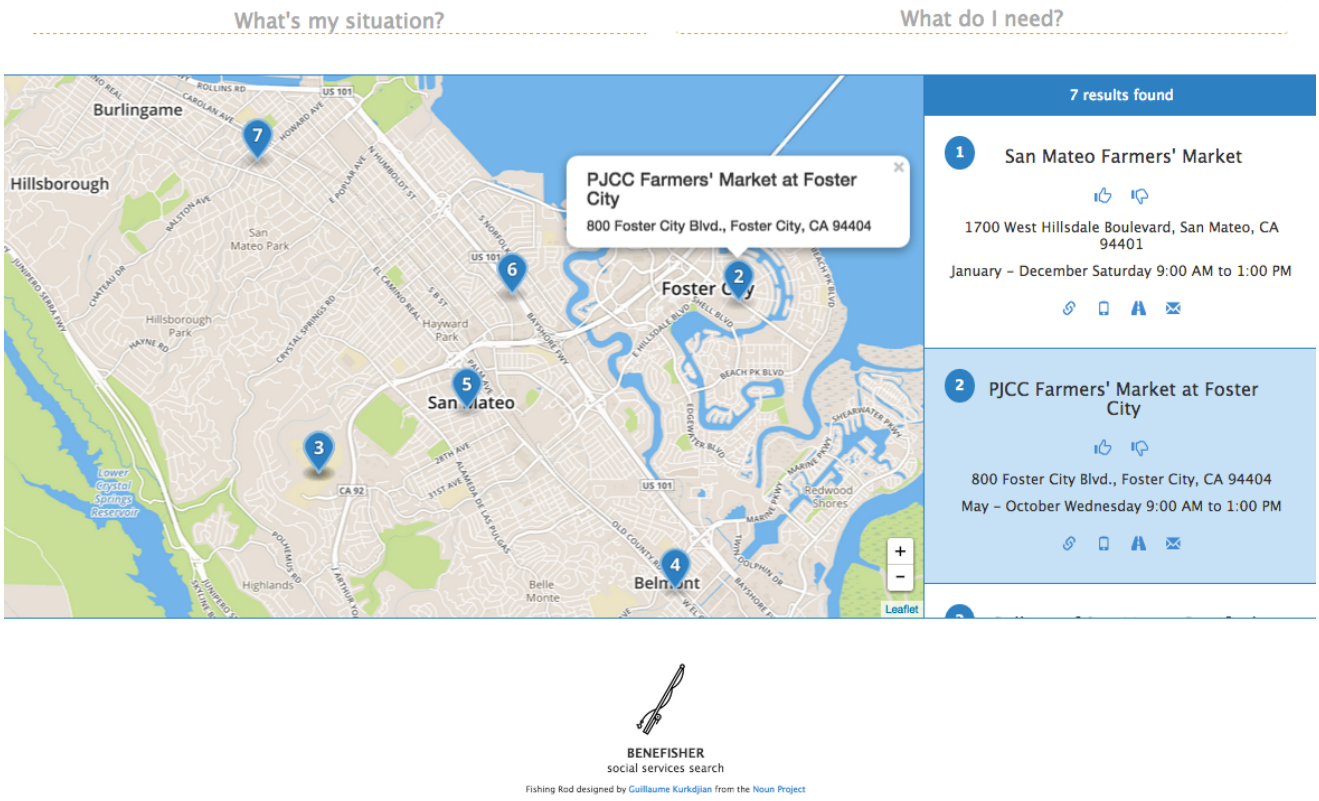

Figure 5.19 The *index.jade* View

Database Requirements: None.

Postconditions: User leaves the webpage.

5. File: index.js

Preconditions: This component is available after the initialization of *app.js (server)*.

Interface specifications: This component is used by app.js to render responses to HTTP GET requests to the application's base URL (e.g. http://www.benefisher.com).

Processing specifications: This component renders layout.jade and returns the resultant HTML via an HTTP response.

Screen print: None.

Database Requirements: None.

Postconditions. This component is only active when processing HTTP requests. Once the HTTP response described above is sent, it is inactive until the next HTTP request.

6. File: interaction-service.js (InteractionService)

Preconditions: This component is available after the page initialization sequence depicted in Figure 3.1 is completed.

Interface specifications: The interaction service provides a single method that allows other components to save statistics about result interactions to the statistics database:

```
StatsService.saveInteraction(interaction)
```

Processing specifications: The interaction service saves interactions to the statistics database via *interactions.js*. The service is used asynchronously and returns a promise that is resolved when the *interactions.js* returns an HTTP response.

Screen print: None.

Database Requirements: The interaction service interacts with the result_interaction database table via HTTP requests to the interactions route (*interactions.js*).

Postconditions: User leaves the webpage.

7. File: interactions.js (InteractionsController)

Preconditions: This component is available after the application initialization sequence depicted in Figure 3.1 is completed.

Interface specifications: This component accepts HTTP POST requests containing an 'interaction' parameter describing a user's interaction with a result, and returns an HTTP response indicating whether the interaction was saved to the database.

Processing specifications: This component validates the content of an HTTP POST request. If the content does not pass validation, an HTTP 400 response is returned, indicating a bad request. If the content passes validation, the interaction data in the *interaction* parameter is persisted to the *result_interaction* table of the stats database, and an HTTP 201 response is returned, indicating that a new resource was created.

Screen print: None.

Database Requirements: Persists interactions to the *result_interaction* table.

Postconditions. This component is only active when processing HTTP requests. Once one of the HTTP responses described above is sent, it is inactive until the next HTTP request.

8. File: layout.jade

Preconditions: This component is rendered by index.js after a user has accessed the application.

Interface specifications: When a user accesses the application, index.js renders this file. This file is responsible for fetching and loading CSS and also establishing the HTML head of the document. See Figure 5.19 for a depiction of template relationships.

Processing specifications: None.

Screen print:

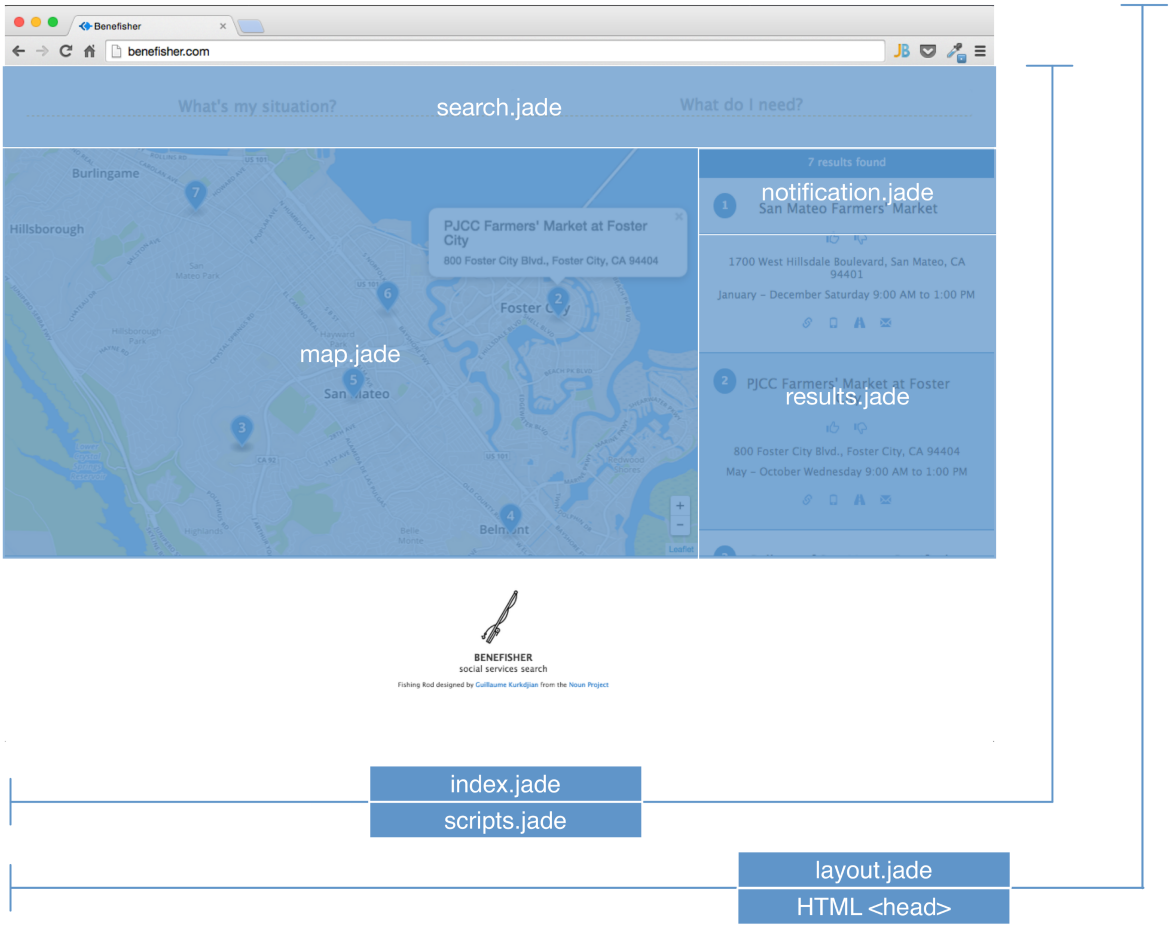

Figure 5.19 Template Relationships in *layout.jade*

Database Requirements: None.

Postconditions: User leaves the webpage.

9. File: map.jade (Map View)

Preconditions: This component is a sub-template of *layout.jade*, and is rendered by *index.js* after a user has accessed the application. See *layout.jade* (Figure 5.19) for a depiction of template relationships.

Interface specifications: A user is able to pan the map, zoom in, zoom out, and center the map on their location, if the user allows the application to obtain their location. A user may select a map marker, which will cause a popup with more information about that result appear above the marker, and the corresponding result to be highlighted in the results component.

Processing specifications: None.

Screen print:

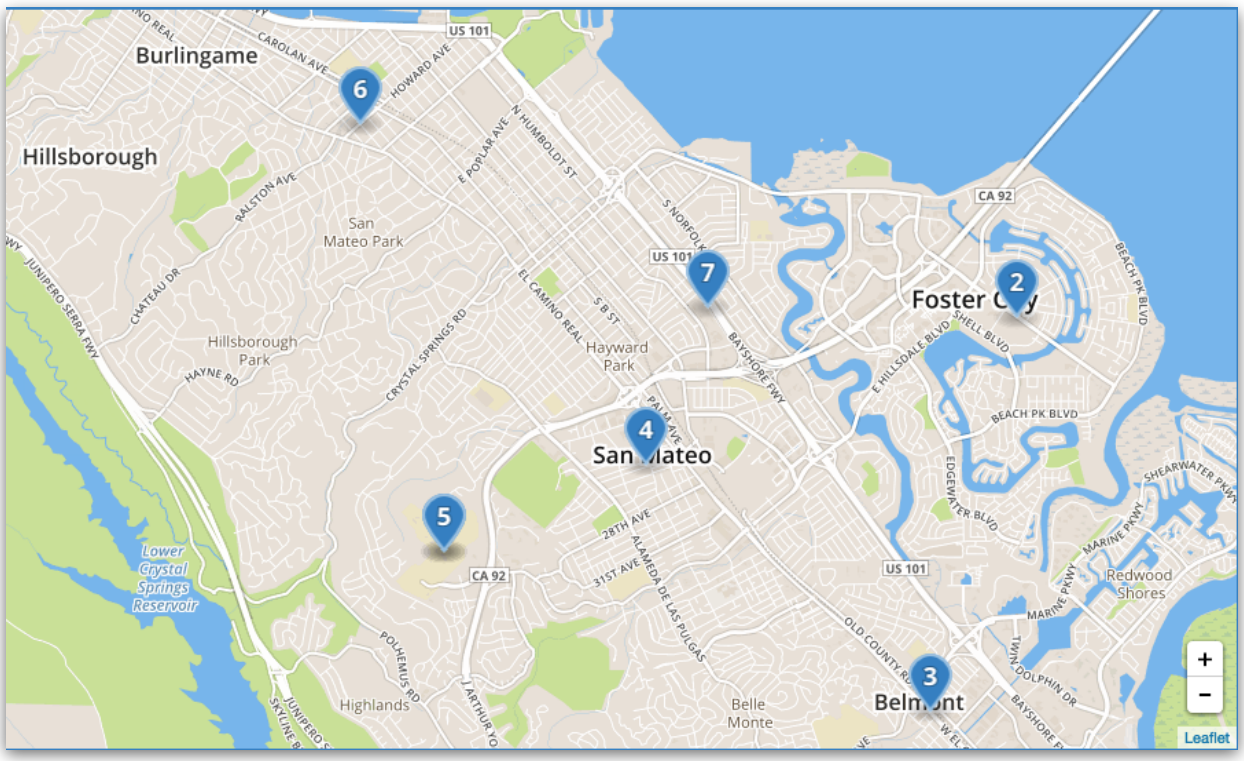

Figure 5.20 The *map.jade* View

Database Requirements: None

Postconditions: The user leaves the webpage.

10. File: map-controller.js (MapController)

Preconditions: This component is available after the page initialization sequence depicted in Figure 3.1 is completed.

Interface specifications: The map controller (MapController) is primarily an event listener for the map element displayed by map.jade, it also implements a subscriber 'update' method that accepts a data object argument that is used to update map markers:

```
MapController.update(data)
```

Processing specifications: When a user changes the bounds of the map via *map.jade*, this component invokes a search for results via the client-side search service (*search-service.js*). This component then waits for the search service to invoke the *update* method described above. When the *update* method is invoked, the updated results are displayed via *map.jade*.

Screen print: None.

Database Requirements: The map controller retrieves results via the client-side search service (*search-service.js*).

Postconditions: User leaves the webpage.

11. File: neural-network.js (NeuralNetwork)

Preconditions: This component is available after the application initialization sequence depicted in Figure 3.1 is completed.

Interface specifications: The neural network will provides a single method for retrieving and ordering search results from the Code for Sacramento data source:

```
NeuralNetwork.getResults(query)
```

Processing specifications: In order to deliver the most relevant results, Benefisher is driven by a multilayer perceptron (MLP) neural network [10, 11]. Inputs to the network come into the input layer (see Figure 5.21) in the form of: user need, situation, and location. These inputs are normalized to either 0 (meaning that the input is not present in the query) or 1 (meaning that the input is present), then passed along to a hidden layer, which is then passed along to the output layer to determine which output is most appropriate for the series of inputs. This system becomes powerful when the system is trained [10], by predicting the correct result and being confirmed as correct or denied as incorrect (meaning that the user positively interacted with the result vs negatively interacting). After being 'trained' as to what a correct result looks like, the connections between the various nodes shift weights through a process known as "backpropagation" [11]. This weight shifting allows the network to become more refined over time, giving the end user more relevant results.

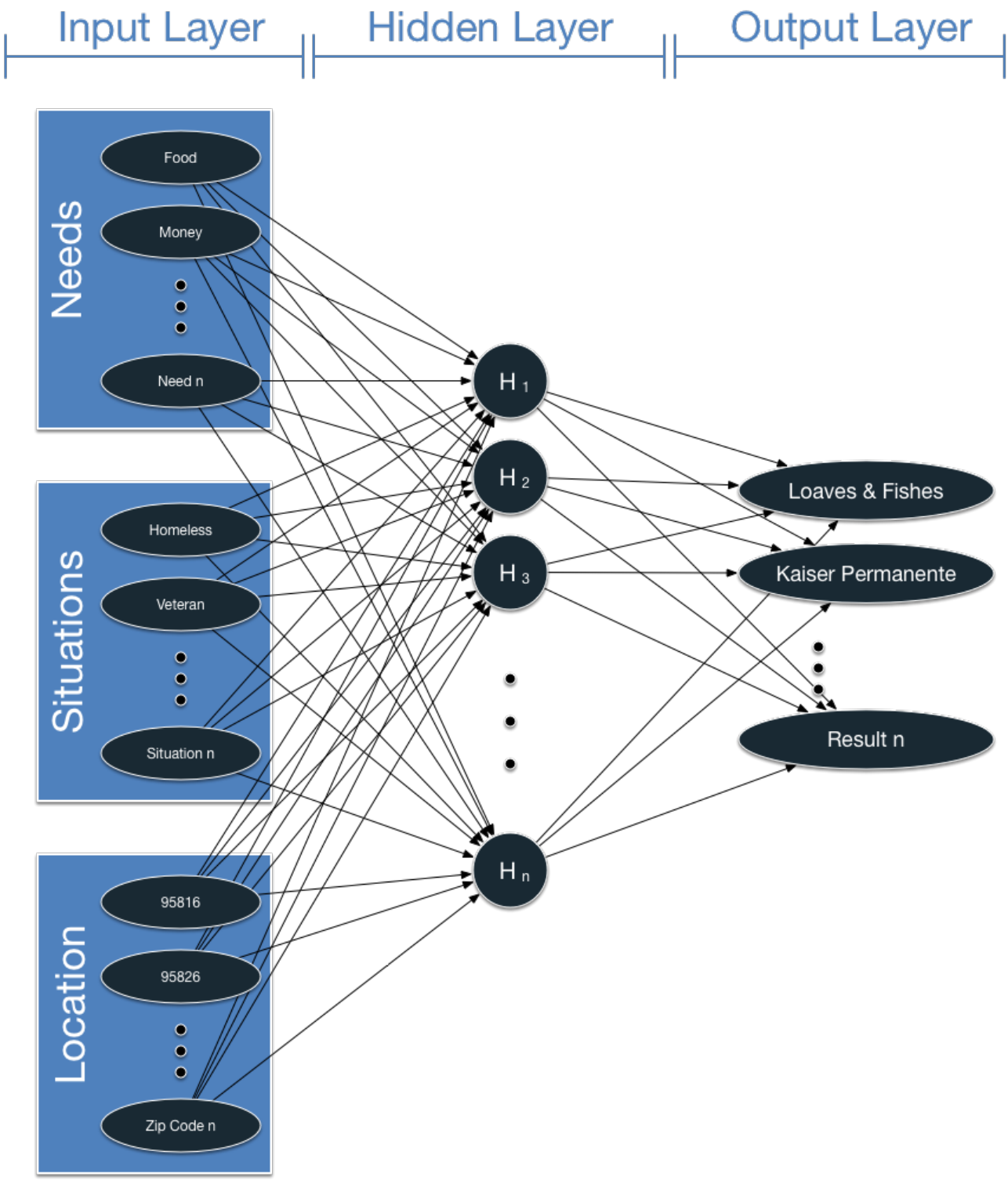

Figure 5.21 MLP Network Design.

Screen Print: None.

Database Requirements: The neural network interfaces directly to the statistics database and the Code for Sacramento data source. It uses data from those sources to compute the most relevant result for the given search query.

Postconditions: The neural network is only active when performing searches. Once the search results have been returned, it is inactive until the next search.

12. File: notification.jade

Preconditions: This component is available after the *notification-service.js* component has completed its initialization.

Interface specifications: A user tries to interact with the Benefisher application, but an API that the application interacts with is unvailable, or a user searches for services and there are no results available for that particular search. A notification is displayed to inform the user that there is a problem, including options to remedy the problem.

Processing specifications: None.

Screen print:

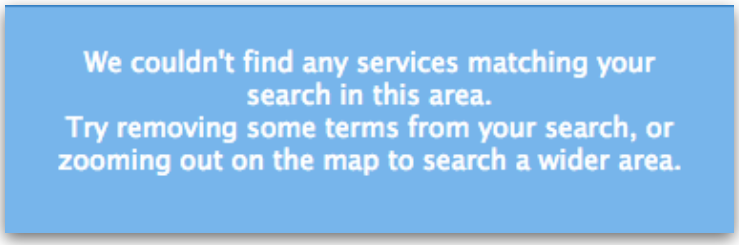

Figure 5.21 The *notification.jade* View

Database Requirements: None

Postconditions: The notification service (*notification-service.js*) removes the notification.

13. File: notification-service.js

Preconditions: This component is available after the page initialization sequence depicted in Figure 3.1 is completed.

Interface specifications: This comonent provides a central notification dispatcher It offers several methods for displaying new notifications with various 'statuses' that affect the duration and appearance of the notification:

```
NotificationService.addNotification(notification, opts)

NotificationService.info(message, opts)

NotificationService.success(message, opts)

NotificationService.warning(message, opts)

NotificationService.error(message, opts)
```

Processing specifications: The notification is added to the display and a timer is set. When the timer expires, the notification is removed from the display.

Screen print: None.

Database Requirements: None.

Postconditions: The user leaves the webpage.

14. File: results.jade (Results View)

Preconditions: This component is available for interaction after the *results-controller.js* component has completed its initialization.

Interface specifications: A user scrolls over the results and is shown the result's corresponding map marker. The user is also able to interact with the result by clicking the icons that are present on the result.

Processing specifications: None.

Screen print:

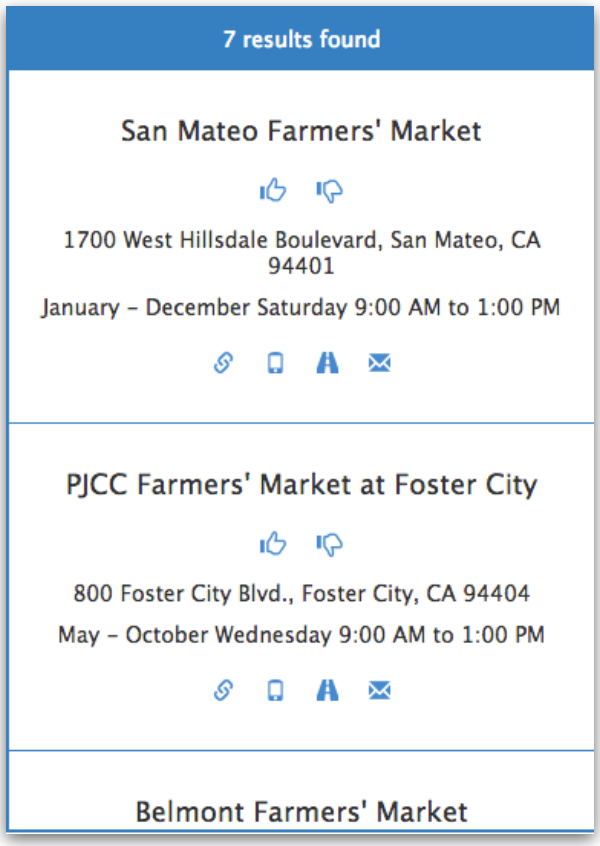

Figure 5.22 The *results.jade* View

Database Requirements: None

Postconditions: User leaves the webpage.

15. File: results-controller.js (ResultsController)

Preconditions: This component is available after the page initialization sequence depicted in Figure 3.1 is completed.

Interface specifications: The results controller is primarily an event listener for the results displayed by *results.jade*. It also implements a subscriber 'update' method that accepts a data object argument that causes the results displayed in *results.jade* to be updated:

```
ResultsController.update(data)
```

Processing specifications: The results controller receives search results through the update method, and displays them via *results.jade*.

Screen print: None.

Database Requirements: The results controller's database interactions are managed by the search service (*search-service.js*).

Postconditions: User leaves the webpage.

16. File: scripts.jade

Preconditions: This component is present after a user has accessed the application. It is required for the webpage, as it is responsible for fetching all of the JavaScript files for the application.

Interface specifications: When a user navigates to the page, all JavaScript files that are required for the application will be fetched and loaded on the client's browser so that they can use the application.

Processing specifications: None.

Screen print: The contents of scripts.jade are not displayed.

Database Requirements: None

Postconditions: User leaves the webpage.

17. File: search.jade (Search View)

Preconditions: This component is available for interaction after the *search-controller.js* component has completed its initialization.

Interface specifications: A user is able to type their situation and need into the specified textboxes. The user is then suggested keywords that are relevant to the entered text.

Processing specifications: None.

Screen print:

Figure 5.23 The *search.jade* View

Database Requirements: None

Postconditions: User leaves the page.

18. File: search.js (SearchController - Server)

Preconditions: This component is available after the application initialization sequence depicted in Figure 3.1 is completed.

Interface specifications: This compone

Processing specifications: When a user chooses search terms via *search.jade*, this component invokes a search for results via the client-side search service (*search-service.js*). This component does not interact with the results retrieved by the search service.

Screen print: None

Database Requirements: This component interacts with the Code for Sacramento data source via the search service (*search-service.js*).

Postconditions:This component is only active when processing HTTP requests. Once the HTTP response described above is sent, it is inactive until the next HTTP request.

19. File: search-controller.js (SearchController)

Preconditions: This component is available after the page initialization sequence depicted in Figure 3.1 is completed.

Interface specifications: This component accepts HTTP GET requests containing search parameters, and returns an HTTP response containing ordered results.

Processing specifications: This component issues an HTTP GET request to the Code for Sacramento data source's REST API. When a successful HTTP response is received, results are upserted to the *search_results* table, and the search query is saved to the *search_query* table. The results are then returned via HTTP response.

Screen print: None

Database Requirements: This component interacts with the Code for Sacramento data source via HTTP requests, and with the application's *search_query* and *search_results* tables directly..

Postconditions: User leaves the page.

20. File: search-service.js

Preconditions: This component is available after the page initialization sequence depicted in Figure 3.1 is completed.

Interface specifications: The search service provides a central service for other components to easily query the Code for Sacramento data source and receive the results they need. This component implements the publisher-subscriber design pattern, and offers a method for other components to subscribe. It also offers a method for querying and receiving results from that query, as well as a method for ignoring a specific result:

```
SearchService.subscribe(subscriber)

SearchService.search(parameters)

SearchService.ignore(result)
```

Processing specifications: One or more components subscribe to the search service. When the 'search' method is invoked, this component issues an HTTP request for results to the server. When a response to that request is received, this component updates its subscribers with results from the response.

Screen print: None

Database Requirements: This component interacts with the database via the server-side search controller (*search.js*) using HTTP requests.

Postconditions: User leaves the page.

6. PERFORMANCE ANALYSIS

This section contains details of implementing the quality attributes detailed in the SRS, as well as any design constraints to Benefisher that have come about as a result of the design and implementation process.

6.1 Required Quality Attributes.

These quality attributes must be addressed in order to deliver a product that meets the requirements, as outlined in the SRS [2].

Reliability

To ensure reliability and relevance, Team Wakati has implemented a neural network machine learning pattern that will analyze a record of search terms with search results. Upon a query for benefits, search terms will be paired with results that users either click on or close. If the user clicks on a result, the query-result connection will be given weight. If the user closes a result, the query-result connection will have its weight lessened. The neural network algorithm will be continually improving and adjusting search results will optimize searches to the current trends of a user base.

Maintainability

Team Wakati is allowing the sponsor Code for Sacramento access to the source code and databases at the end of the project, where the sponsor will continually ensure the code is maintained and the data is correct and valid. At the time of the handoff, Team Wakati will upload the Benefisher source code into the Git repository owned by Code for Sacramento. Code for Sacramento will then use the repository to continue future improvements and otherwise ensure continued maintinence of Benefisher.

Program Quality Attributes

To implement the code, Team Wakati is using the Git repository to coordinate code changes and releases. Each member is using the JetBrains Webstorm 8.0.4 editor for consistency. Source code has been thoroughly commented as to be clear to other team members as well as future maintainers.

Security

Information recorded in the database is central to the application, but not to the user, requiring no information beyond the user's situation, need and location. Zip code searches are saved, generalizing the location that the user is searching and increasing the accuracy of their search without identifying the user personally. Input data is filtered before it is used for queries to protect against SQL injection, including insertion of false data or sabotage by dropping tables.

Transferability and Conversion

The data used by Benefisher is set up to not be dependent on a particular framework, and can be transferred to any other SQL framework

6.2 Optional Quality Attributes

To address the diversity of client hardware, Benefisher uses CSS media queries to detect a user's screen size. Benefisher adjusts its appearance as needed to best accommodate the user's device. Benefisher is designed to be used with index fingers rather than thumbs, to avoid discomfort on larger mobile devices.

Operations

Benefisher is a web-based application and is set up to be continuously available. Continous availiablity means that unplanned outages will be infrequent due to eliminating points of failure. The only two points of failure for Benefisher should be the ISP avalibility of the connection hosting Benefisher or the server hosting Benefisher. Continous availiability is also possible with continous operation where planned outages are avoided. Benefisher in this case should only be taken offline for a few seconds in the event of specific changes such as a site design update. Continuous availability is possible with the implementation, if the sponsor uses a continuous delivery strategy.

Constraints

As a web based application, Benefisher will require an Internet connection for full functionality and cannot be used in an offline capacity.

7. RESOURCE ESTIMATES

This section describes the minimum and recommended system requirements to successfully run Benefisher.

Note: Because server load is difficult to predict, live testing is the best way to determine the hardware resources Benefisher will require in production.

7.1 Minimum Software Requirements

Software Attribute	Minimum
Operating System	Linux - Ubuntu 13.10 Server or later
Database	MySQL - version 4.1.22
Server Framework	Node.js Framework - version 0.10.32 or later

Table 7.1 Minimum Software requirements

7.2 Minimum Hardware Requirements

The values below refer to the minimum available hardware required to run Benefisher and its associated database.

Harware Attribute	Minimum
CPU	2 CPU Cores
Memory	2 GB RAM
Disk Volume	100 GB
Internet Connectivity	Ethernet

Table 7.2 Minimum Hardware requirements

7.3 Recommended Hardware Requirements

The values below refer to the recommended hardware required to run Benefisher and its associated database. A dedicated web server is recommended when deploying Benefisher in production

Hardware Attribute	Recommended
CPU	4 CPU Cores or more
Memory	16 GB RAM or more
Disk Volume	200 GB or more
Internet Connectivity	Ethernet Connectivity

Table 7.3 Recommended Hardware requirements

8. SOFTWARE REQUIREMENTS TRACEABILITY

The matrix relates the design components to their associated requirements by listing the subsection numbers in this document and the associated subsection numbers in the Software Requirements Specification Document.

Component	SDS Section	SRS Section
Map Component	3, 5.1, 5.2	2.1, 2.2, 2.3, 3.1
Search Component	3, 5.1, 5.2	2.1, 2.2, 2.3, 3.1
Results Component	3, 5.1, 5.2	2.1, 2.2, 2.3, 3.1
Notification Component	3, 5.1, 5.2	2.1, 2.2, 2.4, 3.5
Statistics Component	3, 4.1, 4.2, 5.1, 5.2	2.2, 3.1, 3.2, 3.5
Neural Network Component	3, 4.1, 4.2, 5.1, 5.2	3.1, 3.5

Table 8.1 Software Requirements Traceability Matrix

9. APPROVALS

The signatures here indicate that the relevant stakeholders understand the contents of this document, and approve of the outlined software design.

Name	Signature	Date
Adrian Chambers		
Anthony Cristiano		
Daniel Green		
James Doan		
Jesse Rosato		
Sponsor Representative		
Meiliu Lu (Advisor)		

Table 9.1 Project Team, Advisor, and Sponsor Signatures

APPENDIX A

Entity Relationship Diagram.

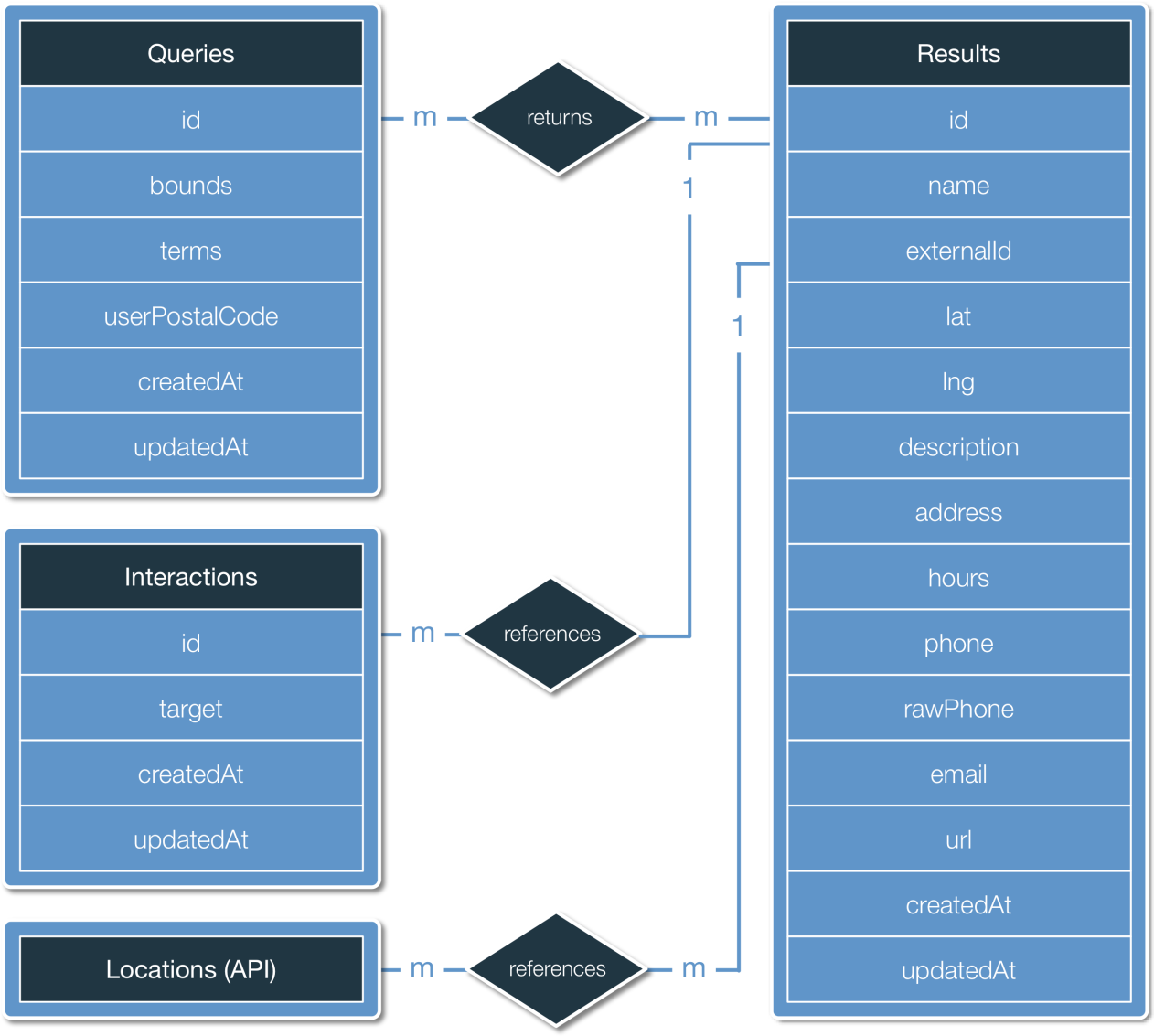

Figure A.1 System ERD

APPENDIX B

Listing of each database table and its attributes.

***Queries* Table**			
Column Name	**Type**	**Description**	**Use**
id	INT	Unique id for a user search query.	Primary key when retrieving search queries.
bounds	VARCHAR(255)	A string describing the map bounds of the query	Used to track search queries.
terms	VARCHAR(255)	A comma separated series of search terms.	Used to track search queries.
user_postal_code	VARCHAR(10)	The postal code associated with a user search query.	Used to track queries by geographic location.
createdAt	DATETIME	The date and time a user search query was executed.	Used to track search queries over time.
updatedAt	DATETIME	The date and time a user search query was executed.	Used for record-keeping.

Table B.1 *Queries* Table Description

Results Table			
Column Name	**Type**	**Description**	**Use**
id	INT	Unique id for a user search result.	Primary key when retrieving search results.
externalId	VARCHAR(255)	Unique identifier for a service related to a search result.	Foreign key when retrieving a service from a search result.
name	VARCHAR(255)	The name of the service.	For display to user.
lat	FLOAT(18,12)	The latitude of the service's location.	Used to plot service locations on map.
lng	FLOAT(18,12)	The longitude of the service's location.	Used to plot service locations on map.
description	TEXT	A description of the service.	For display to user.
address	VARCHAR(255)	The human-readable address of the service's location.	For display to user.
hours	VARCHAR(255)	A text description of the hours the service is available.	For display to user.
phone	VARCHAR(20)	A formatted phone number.	For display to user.
rawPhone	VARCHAR(20)	An unformatted phone number.	Used to generate a phone URL for mobile users.
email	VARCHAR(255)	An email address.	For display to user.
url	VARCHAR(255)	A URL.	For display to user.
createdAt	DATETIME	The date and time a result was saved to the database.	Used for record-keeping.
updatedAt	DATETIME	The date and time a result was updated.	Used for record-keeping.

Table B.2 *Results* Table Description

QueriesResults Table			
Column Name	**Type**	**Description**	**Use**
id	Integer	Unique ID for a query-result relationship.	Unique ID for a query-result relationship.
ResultId	Integer	Unique ID for a result.	Foreign key when retrieiving query-result relationships.
QueryId	Integer	Unique ID for a query	Foreign key when retrieiving query-result relationships.

Figure B.3 *QueriesResults* Table Description

Interactions Table			
Column Name	**Type**	**Description**	**Use**
id	INT	Unique identifier for a user interaction with a search result.	Primary key when retrieving search result interaction.
target	VARCHAR(255)	A description of the target (e.g. “call”, or “get directions”) of a user’s interaction with a search result.	Used to track types of interactions with search results.
ResultId	VARCHAR(255)	Unique identifier for a search result related to a search result interaction.	Foreign key when retrieving a search result from a search result search interaction.
createdAt	DATETIME	The date and time a user interacted with a search result.	Used to track interactions over time.
updatedAt	DATETIME	The date and time an interaction record was updated	Used for record-keeping.

Table B.4 *Interactions* Table Description