System Design Document

# <Website-name-here>

Client

Morgan Madeira

Team 3

Connor Cook

Matt Law

Brent Mattox

Austin Murdock

Michael Neary

March 2nd, 2014

[Put product name here]

System Design Document

#### Table of Contents

Page

## 1. Introduction

* 1. Purpose of This Document
  2. References

1. System Architecture

2.1 Architectural Design

2.2 Decomposition Description

3. Persistent Data Design

5.1 Database Descriptions

5.2 File Descriptions

1. Requirements Matrix

Appendix A – Agreement Between Customer and Contractor

Appendix B – Peer Review Sign-off

Appendix C – Document Contributions

1. **Introduction**

1.1 Purpose of This Document

This document describes the software architecture by identifying the sub-systems of the website. This document will also outline the communication between these sub-systems and how the sub-systems are controlled.

This document will serve to guide the project developers and verify the customer of the sponsor.

1.2 References

Provide a list of all applicable and referenced documents and other media. Minimally, references to the SRS and UI Design Document go here. If you used any other documents or references to arrive at this design (e.g., the Somerville text, UML references, documents provided by the customer, websites), list them here. See the Writing Resources on Blackboard for the appropriate formats for references.

Burbeck, Steve (1992). Application Programming in Smalltalk-80: How to use Model-View-Controller (MVC). Retrieved from <http://st-www.cs.illinois.edu/users/smarch/st-docs/mvc.html>

Ehsan, Max. "Architecture of Laravel Applications." *Architecture of Laravel Applications*. N.p., n.d. Web. 03 Mar. 2014. <http://laravelbook.com/laravel-architecture/>.

1. **System Architecture**
   1. Architectural Design

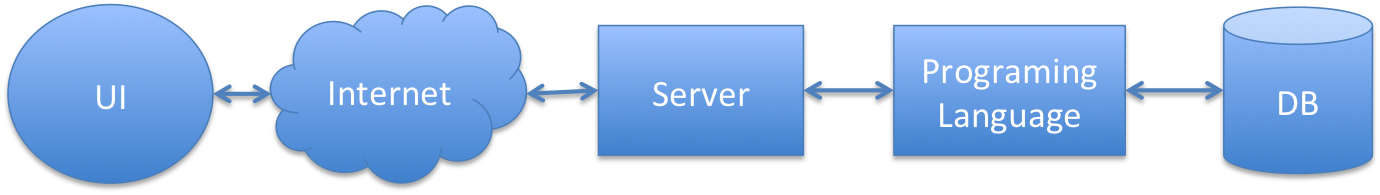


Figure 2.1 Horizontal software solution stack.

The system runs on a software solution stack consisting of an operating system, web server, database, and scripting language.

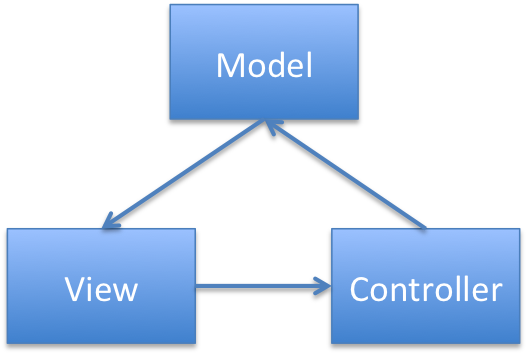
Currently the system is built on the LAMP software bundle as follows:

* Linux as the operating system
* Apache as the web server
* MySQL as the database management system
* PHP as the scripting language

The development and interaction of these components is abstracted using the Laravel framework. Users will interact with the system using a web browser user interface, which will be facilitated by the aforementioned Model-View-Controller software pattern.

2.2 Decomposition Description

Figure 2.2: A typical Model-View-Controller relationship.



Our project will utilize a slightly altered Model-View-Controller software pattern, extended by Laravel’s capabilities as a full-stack framework. **Model-View-Controller**, commonly known as **MVC**, is a commonly used pattern for implementing web-based user-interfaces.

The MVC design pattern is made up of three components:

* The **model** is the domain the software encompasses- they are based on items and concepts such as a person, a post, or a product. Models both store data and enforce rules regarding its storage, ensuring constraints are properly observed. For our product, the models would include *recipes*, *reviews*, *users*, and *comments*.
* The **view** is the visual representation of the model that can contain additional markup to be rendered to the user’s browser. The view generates the user interface and formats the data contained in the models. While it may present varying methods of input to the user, it doesn’t handle the input itself- its work is done once the data is displayed.
* The **controller** facilitates the link between the view and the models. It processes input, performs actions on the model, decides what actions to take- such as whether to present a login-page or a redirect page. For example, it is the controller that examines a *recipe* and pulls up its *comments* to pass to the view, which then presents it.

In Laravel and for our project, this MVC relationship is altered slightly. To best understand these changes, we will examine a complete request in a Laravel framework (see Figure 2.3).

First, the browser sends a request to the application via the web server hosting it (1). Laravel’s routing engine processes the request and hands it off the appropriate controller class (2). The controller then interacts with the data models sitting on top of the database, pulling needed information as necessary (3). Once the controller invokes the model, it invokes a view containing the requested information (4) which in turn returns the complete web page to the user’s browser.

Figure 2.3: Lifecycle of a request in a Laravel application.

1. **Persistent Data Design**
   1. Database Descriptions

The system uses a MySQL database facilitated in part by multiple aspects of the Laravel framework, additionally supplemented by possible future additions such as mongodb. For now, this database utilizes Eloquent, a method for simplifying access to databases, enforcing relationships between models, and manipulating data. It is configured via the database.php config file found in /app/config/.

As such, our access to the database will be roundabout- for example, defining a database relationship for recipes could be expressed as:

**class** **Recipe** **extends** **Eloquent**

{

**public** **function** **author**()

{

**return** $this->belongsTo('Owner');

}

**public** **function** **comments**()

{

**return** $this->hasMany('Comments');

}

}

* 1. File Descriptions

Our project utilizes a standard Laravel file directory system. The top level folder is named after our application, working title being “digidiet”. Inside are three major folders. Chief among them is **app**, containing the majority of our project, including models, views, controllers, and assets. It also holds the “routes.php” file which informs the application how to handle requests.

The **public** is the only exposed folder in the application, most importantly containing the bootstrap “index.php” file which initializes the application itself.

Lastly, the **vendor** folder contains the core of Laravel, its source code, dependencies, plug-ins, and additional prepackaged functionality. For the most part, we will not be developing in this folder, just utilizing its contents.

1. **Requirements Matrix**

Use a tabular format to show which system components (e.g., functions and/or methods) satisfy each of the functional requirements from the SRS. Refer to the functional requirements by use case number and name.

**Appendix A – Agreement Between Customer and Contractor**

Place on a separate page. Describe what the customer and your team are agreeing to when all sign off on this document. [One paragraph] Include a statement that explains the procedure to be used in case there are future changes to the document. [One paragraph] Provide lines for typed names, signatures, and dates for each team member and the customer. Provide space for customer comments.

**Appendix B – Team Review Sign-off**

Place on a separate page. Provide a brief paragraph stating that all members of the team have reviewed the document and agree on its content and format. Provide lines for typed names, signatures, dates, and comments for each team member. The comment areas are to be used to state any minor points regarding the document that members may not agree with. Note that there cannot be any major points of contention.

**Appendix C – Document Contributions**

Identify how each member contributed to the creation of this document. Include what sections each member worked on and an estimate of the percentage of work they contributed. Remember that each team member must contribute to the writing (includes diagrams) for each document produced.

**Unified Modeling Language (UML)**

**Class Diagrams**

Reference: **UML Distilled**, 2nd edition, Martin Fowler and Kendall Scott, 2000

Note: The information below has been modified slightly to meet the purposes of CMSC 345

Symbols:

Class – Represented by a box as follows:

*Class Name*

*Operations*

*Attributes*

*Class Name*

Navigability – Represented by a solid line with an arrowhead at one or both ends (unidirectional or bidirectional association, respectively). Indicates the direction(s) of an association between classes. (Below, A can navigate to B, but not the reverse.)

A

B

Generalization – Represented by a solid line with a hollow arrowhead at one end. Indicates that one class is a generalization of another. (Below, B is a generalization of A.)

A

B

Composition – Represented by a solid line with a solid diamond at one end. Indicates that an instance of one class is “owned” by a single instance of another. (Below, an instance of B is owned by a single instance of class A. Note: if an instance of A is deallocated, the associated instance(s) of B are also deallocated.)

B

A

Dependency – Represented by a dashed line with an arrowhead at one end. Indicates that one class depends on the interface to another class. (Below, an instance of A is dependent on the interface to an instance of B.)

A

B

Multiplicity – Indicates how many instances of one class type are associated with another class. (Below, 1 to 5 instances of class A are associated with class B.)

B

A

1**. .** 5

Note: We will not be using any other symbols in our class diagrams.

Format for class attributes:

*visibility name* : *type* = *defaultValue*

where *visibility* = + for public, # for protected, - for private

*name* = attribute name

*type* = data type

*defaultValue*  = default value

Format for class operations:

v*isibility name* (*parameter-list*) : *return-type*

where *visibility* = + for public, # for protected, - for private

*name* = operation name

*parameter-list*  = comma-separated parameters with the syntax

*direction name* : *type* = *defaultValue*

where *direction*  = in for input

out for output

inout for input/output

*name* = parameter name

*type* = parameter type

*defaultValue* = default value

*return-type* = return type