

California State University, San Bernardino

**CSUSB ScholarWorks**

---

Theses Digitization Project

John M. Pfau Library

---

2013

## Ticketing and event management web service

Nikolay Figurin

Follow this and additional works at: <https://scholarworks.lib.csusb.edu/etd-project>



Part of the [Databases and Information Systems Commons](#)

---

### Recommended Citation

Figurin, Nikolay, "Ticketing and event management web service" (2013). *Theses Digitization Project*. 4231.  
<https://scholarworks.lib.csusb.edu/etd-project/4231>

This Project is brought to you for free and open access by the John M. Pfau Library at CSUSB ScholarWorks. It has been accepted for inclusion in Theses Digitization Project by an authorized administrator of CSUSB ScholarWorks. For more information, please contact [scholarworks@csusb.edu](mailto:scholarworks@csusb.edu).

TICKETING AND EVENT MANAGEMENT WEB SERVICE

---

A Project  
Presented to the  
Faculty of  
California State University,  
San Bernardino

---

In Partial Fulfillment  
of the Requirements for the Degree  
Master of Science  
in  
Computer Science

---

by  
Nikolay Figurin  
September 2013

TICKETING AND EVENT MANAGEMENT WEB SERVICE



---

A Project  
Presented to the  
Faculty of  
California State University,  
San Bernardino



---

by  
Nikolay Figurin  
September 2013

Approved by:

  
  
Ernesto Gomez, Advisor, School of  
Computer Science and Engineering

  
  
Keith Schubert

  
  
Kerstin Voigt

28 AUGUST 2013  
Date

© 2013 Nikolay Figurin

## ABSTRACT

Event management and ticket sales platforms are moving to the web. This service, called TicketStand, is utilized as an event creation and ticketing system. The project aims at a version of the software that includes: user account creation, event creation per user, tiered ticket sales per event, search capabilities, a focus on scalability and data redundancy, a focus on verification of event creator identities and administration features, a focus on account expandability, an ability to easily expand functionality, ease of migration and stability, low and inexpensive maintenance, as well as an optimal framework relying on PaaS CDN with the option of database abstraction.

## ACKNOWLEDGEMENTS

I wish to express my deepest gratitude to the Computer Science department for giving me the resources and opportunity to further not only my education but also my understanding of the field by allowing me to pursue the M.S. degree. I wish to specifically thank Dr.Gomez, Dr.Schubert, and Dr.Voigt for being on my committee and offering an incredible amount of help throughout the entirety of the program.

## TABLE OF CONTENTS

<i>Abstract</i> . . . . .	iii
<i>Acknowledgements</i> . . . . .	iv
<i>List of Figures</i> . . . . .	viii
<i>1. Introduction</i> . . . . .	1
1.1 Purpose of the Project . . . . .	1
1.2 Event Management and Ticketing System Participants . . . . .	2
1.2.1 Sellers . . . . .	2
1.2.2 Event Manager/Creator/Organizer . . . . .	3
1.2.3 Event Attendance Coordinator . . . . .	3
1.2.4 Administrator . . . . .	4
1.2.5 Customer . . . . .	5
1.3 Two Types of Event Management Workflow . . . . .	5
1.3.1 Individual Seller Model . . . . .	5
1.3.2 Team-based Model . . . . .	5
1.4 Project Products . . . . .	6
<i>2. Event Management and Ticketing System Structure</i> . . . . .	8
<i>3. Database Design</i> . . . . .	22

4. <i>System Implementation</i> . . . . .	26
4.1 <i>General Structure of the System</i> . . . . .	26
4.2 <i>Scenario-Specific Workflows</i> . . . . .	30
5. <i>System Validation and Security</i> . . . . .	34
6. <i>Installation and Maintenance</i> . . . . .	37
6.1 <i>Installation</i> . . . . .	37
6.2 <i>Maintenance and Scaling</i> . . . . .	39
7. <i>Conclusion and Future Directions</i> . . . . .	40
7.1 <i>Conclusion</i> . . . . .	40
7.2 <i>Future Directions</i> . . . . .	41
<i>Appendix A: Routing Configuration File</i> . . . . .	42
<i>Appendix B: Account Controller Class</i> . . . . .	51
<i>Appendix C: Administration Controller Class</i> . . . . .	61
<i>Appendix D: Event Controller Class</i> . . . . .	65
<i>Appendix E: MyEvent Controller Class</i> . . . . .	80
<i>Appendix F: MyPurchase Controller Class</i> . . . . .	91
<i>Appendix G: Public Controller Class</i> . . . . .	94
<i>Appendix H: Purchase Controller Class</i> . . . . .	101
<i>Appendix I: Model of Ticket Class</i> . . . . .	106
<i>Appendix J: Model of User Class</i> . . . . .	109



<i>Appendix K: Model of Tier Class</i> . . . . .	113
<i>Appendix L: Front Page View File</i> . . . . .	116
<i>Appendix M: View Portion of the editmyeventdetails File</i> . . . . .	123
<i>References</i> . . . . .	131

## LIST OF FIGURES

2.1	High Level TicketStand Workflow for Creating an Event. . . . .	9
2.2	TicketStand Frontpage. . . . .	10
2.3	An Event Page in TicketStand. . . . .	12
2.4	TicketStand Login Page. . . . .	13
2.5	TicketStand Registration Page. . . . .	14
2.6	TicketStand Account Dashboard. . . . .	15
2.7	TicketStand's My Events Page. . . . .	17
2.8	An Edit Event Details View in TicketStand. . . . .	18
2.9	Summary of Requested Tickets in a Purchase. . . . .	20
2.10	A List of All Purchased Tickets By a Logged In User. . . . .	21
3.1	Ticketstack Database Schema from TicketStand. . . . .	23
4.1	TicketStand System Structure Detailing a Request and Response Scenario. . . . .	27

## 1. INTRODUCTION

Event management and ticket sales platforms are moving to the web. This service, called TicketStand, is utilized as an event creation and ticketing system.

### *1.1 Purpose of the Project*

The project purpose revolves around creating web based application that includes: user account creation, event creation per user, tiered ticket sales per event, search capabilities, a focus on scalability and data redundancy, a focus on verification of event creator identities and administration features, a focus on account expandability, an ability to easily expand functionality, ease of migration and stability, low and inexpensive maintenance, as well as an optimal framework relying on a platform as a service content delivery network with the option of database abstraction. The current options for creating an event and managing ticket sales, attendance, and other information digitally leaves users with little alternatives. Most free and low-cost solutions deliver services with agreements that disallow the use of such systems for profit, and are usually supported by advertisements. This is hurtful to event organizers as the options are scarce, the end user agreements are constrictive, and the services offered are centralized- thus hurting sales, if any are allowed to begin with. More expensive, enterprise options result in high upfront costs with a large percentage

being taken off in royalties on each purchase. Such a system is not attainable for event organizers of a smaller scale and may not be optimal due to the centralized nature of the services causing hard dependencies, thus making a switch from one system to another unfavorable[8]. This project aims to solve the aforementioned issues by creating a web application that would enable its users to manage events as well as ticket sales with no tie to a centralized system, unless they wish. The web application is portable enough to work under most modern platform as a service hosts and is expandable enough to easily add new functionality via the model view controller pattern utilized by the Laravel 4 framework.

## *1.2 Event Management and Ticketing System Participants*

There are five main types of participants that may be found in the process of organizing, creating, and managing an event and its purchases in such a system - sellers, event organizers, event attendance coordinators, administrators, and customers[9].

### *1.2.1 Sellers*

Sellers are individuals that wish to create an event and sell the admission tickets for it on their own. These individuals are expected to create an event and manage it through the service without another party. The primary concerns of a seller are to disperse all information about the event to the public through a searchable and publically accessible form, and to receive payment for the tickets as well as a list of attendees for easy management. It is possible to create an event with tickets that have no price- therefore creating a free event while keeping a list of attendees. An

example of this type of seller would be a wedding host, or a small seminar gathering organizer.

### *1.2.2 Event Manager/Creator/Organizer*

Event managers and organizers have the role of creating the event with all required data. These users are usually part of a team where all event-related tasks are distributed between the members. The primary goal of an event manager is to populate the event form with all necessary information about the event. Everything from the event address to the contact information may be included on the event form. The event form also allows for multiple tiers of tickets, such as general admission tier, a VIP tier, and a founder tier, as an example. These users also provide support for the customers they reach via an external service of their choosing (email, phone, and similar means). The event organizers may use a single account with their teammates for account creation and management to minimize the chances of any fragmentation in the workflow and in the public-facing portion of the application. The consolidation of different roles into one account also allows for quick changes by all roles in the case of an error or other emergency. This scheme of account management may also enable venues to create their own accounts to discuss the event details with other event organizers while keeping the events centralized to that venue in particular.

### *1.2.3 Event Attendance Coordinator*

The event attendance coordinator is a user that focuses on the attendance aspect of each event. Such a user may perform a function similar to that of a bouncer

or guardian at the door to ensure that the participants admitted to the event are legitimate clients that have purchased tickets to the event. Generally events will hold multiple event attendance coordinators at the doors, forcing the utilization of a synchronizing ticket tracker to be present in the application. The event attendance coordinators may require access to details such as the unique code or ID of a ticket, the event information per-ticket, tier information, customers first and last names, and the email of the customer. Predominantly the ticket code or ID is used due to privacy concerns exhibited with smaller vendors.

#### *1.2.4 Administrator*

The administrator plays a vital role in account management of all registered users as well as the ability to edit any event details that do not withhold a required degree of integrity. Generally administrators are users which focus on the setup and maintenance of a non-centralized application as well as aiding sellers, event organizers, and event attendance coordinators with any required permissions and external tasks. The administrator role may be extended to other participants in the system on a per-user basis, thus allowing an external party of maintenance or support staff to quickly make changes to events and users such as deletion of events, users, and elevation of roles. Administrators also require the ability to freeze accounts and events to disallow ticket sales while keeping the fraudulent sellers information up in the public domain.

### *1.2.5 Customer*

The customers are the users that purchase event tickets through the system directly from the event organizer teams or individual sellers. Generally a customer requires the ability to search all events of interest and their relevant information prior to committing to a purchase for an admission ticket. Customers also have the need to go back through their past purchases, and view the events they have purchased tickets for. The customer will also have a need to create a full account detailing their contact information (such as email) and be able to refer back to all of their consolidated information in one place.

## *1.3 Two Types of Event Management Workflow*

### *1.3.1 Individual Seller Model*

This scenario includes an individual creating and managing the event entirely by themselves with no team workflow. Such a use may be exhibited in very small events such as parties and general get-togethers. This workflow revolves solely around the user as all required duties (outside of payment handling) are completed by the one seller. The event creation process may be summed up with account creation, event information aggregation into one searchable entity or form, and attendance tracking during the event if required.

### *1.3.2 Team-based Model*

This scenario features a team where multiple tasks may be broken up per person but consolidated under one account. This may include an event organizer, event

attendance coordinator, and administrator if the system is not centralized. The tasks may also be split into groups and each person may hold responsibility over any atomic tasks they require[10]. Such a workflow may separate a venue from event organizers, or aid with distributing the workload between a team of event organizers and their staff.

#### *1.4 Project Products*

To better suit the needs of event organizers, event attendance coordinators, individual event sellers, and clients, we need to build a system that will allow a non-centralized, web-based event management and ticketing platform which will allow all users to fulfill their required tasks in an orderly and consolidated manner. This project builds such a system with focus on extensibility, portability, and security. The aforementioned participants may fulfill all tasks through the web-based application with little to no maintenance and a small amount of technical background. The service allows for creation of accounts, creation of events, search of all events, aggregation of event information through a single form, multiple layers of security, an option to implement any required payment APIs, fast installation through multi-tiered configurations and migrations, as well as allowing editing of events, tracking attendance and ticket redemption, deleting events, and freezing of fraudulent accounts and event pages.

The project limits its scope to just these services and the aforementioned participants. Payment options are abstracted due to the nature of the market and disability to select a universal payment gateway, while features such as inter-platform communication are simply a non-issue due to the spread and standardization of other



readily available communication protocols[11]. Each user registered in the service may both purchase and sell tickets. The list of aforementioned features applies to all registered users with the exception of administrative role properties, which must be enabled per-user by the initial administrator that will be created on the initial setup of the application. This project has successfully delivered an extendable system which provides the functionality of all mentioned requirements with a complete user interface that is compatible with the WebKit and Gecko web engines, multiple safeguards for security, dependable structure for easy expansion and implementation of external APIs, complete migration list for portability, and consolidated configuration lists and dependency managers.

## 2. EVENT MANAGEMENT AND TICKETING SYSTEM STRUCTURE

Although the project may be molded into many different services, the project was originally built around very specific workflows. These workflows focus on purchasing, selling, and managing event tickets as well as managing user accounts. Views come into play as they are the part of the framework that allow generation of HTML elements, and each step of the way relies on the combination of the logic done in the controller, the redirecting and passing of variables done by the routing functions, and the database links created in the database and models of the application. On a high level, the system may use this structure to redirect the user through a specific sequence such as publishing an event as exhibited in Figure 2.1. The workflow of the product is straightforward and simple to understand. A non-registered user may go to the web address of the site and browse through the front page, the search pages, event pages, and the registration and sign-in pages on the service. The front page (as seen in Figure 2.2) features general information about the service and what it does. In the case of a venue using such a system, the venue may populate these static fields with background information and history of the location, as well as a dynamic call-to-action button urging visitors to register. Located below the general information is a search bar. The search bar allows search of all events through the use of event name, event type, event description, user name, city, state, and zip code of the event.

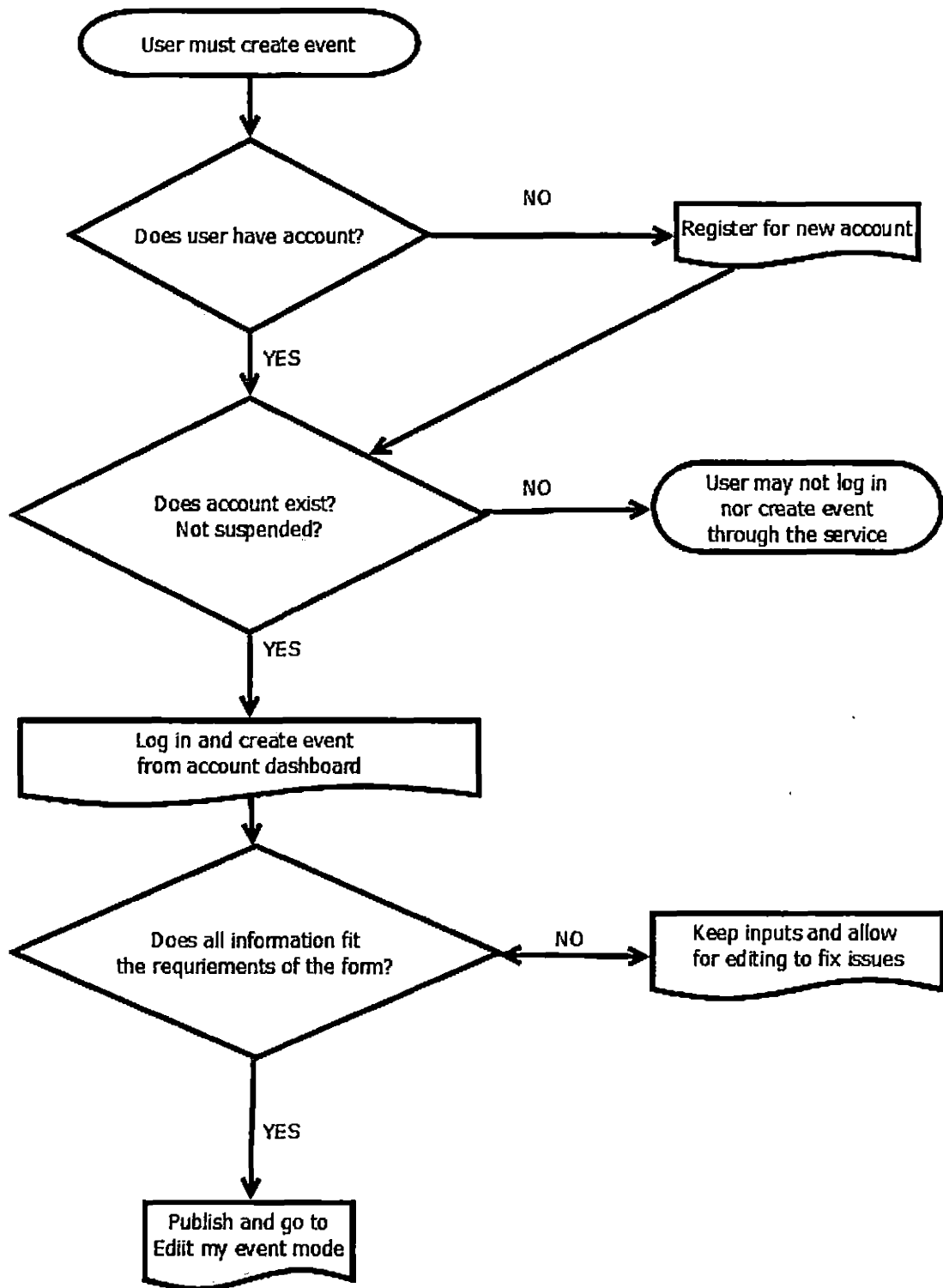
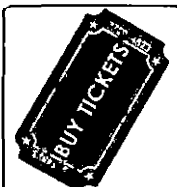


Fig. 2.1: High Level TicketStand Workflow for Creating an Event.


5

**TicketStand**


Welcome to Nobby Figur's U.S. project. This service, currently called ticketstand, is a ticket purchasing and selling platform. This site can be hosted not only on a PaaS, as initially expected, but also on most other servers with very little change. The service is easily expandable and well-organized with an MVC pattern. Some security features are absent that would come standard with proper hosting.  
The site runs on **Laravel 4** and **MySQL** (and a ton of other tools).



Use TicketStand to purchase any event tickets on-line quickly with our easy checkout. Currently, Payment is made as C/P. The Card is processed for immediate billing.



Use TicketStand to sell any event tickets you own. Attention: currently, you must first create and use the service to hold your event to be able to sell.



This service allows you to manage any of your tickets. It allows you to manage your tickets by event, by event name, by event date, and by event location.

Disclaimer: This is a development server, do not use this outside of testing.

[Register Today](#)

---

**Search Events**

Search Events

Search Names  
  Search Types  
  Search Description  
  Search Users  
  Search Dates  
  Search States  
  Search Zipcodes

---

**Upcoming Events**

Event One from DB 0 Tickets Left

**Left** **Price** **Future** 2013-12-17 21:53:59 Redlands, CA

Fig. 2.2: TicketStand Frontpage.

The visitor may search a term to be redirected to a page of all events that match the query. Under the search bar is a list of 10 upcoming events sorted by closest date first. Each event in this list is in stub form. The search view paginates results at five events per page, with each event listed being in a stub form. The stub form is a frame of a 200 pixel height that contains the events most immediate information. This information includes the event name, date, time, city, state, amount of tickets left, three categories (or types) to classify the event, an event summary, and a call-to-action button to view the event. The three categories (or types) are visible as small pill buttons which when clicked will redirect the user to the search page displaying all other events with the selected type. Selecting the See event info call-to-action will result in a new page detailing the event view.

The event page (seen in Figure 2.3) lists more information about the event than the stub view. The information listed includes the event name, the three type pill buttons, the date of the event, time of the event, city, state, zip code, street address, summary of the event, number of tickets available, a full-text event description, up to three tier fields, three photos pertaining to the event, an interactive Google maps frame showing the location of the event and allowing the possibility to go to a new window with directions via Google maps, a field detailing the event creators information including their username, first and last name, and email, and lastly the dynamic tri-state purchase button. The purchase button holds the values of Buy Tickets with a redirect to a purchase page (or the login page if the user isnt signed in), Sold Out specifying the tickets are all sold, and Event Over denoting that the event in question has passed.

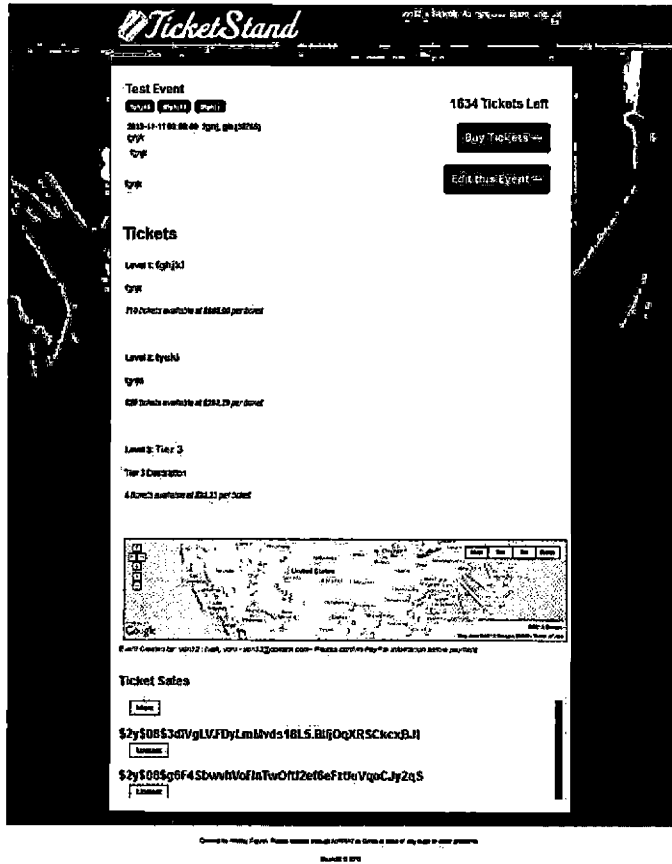


Fig. 2.3: An Event Page in TicketStand.

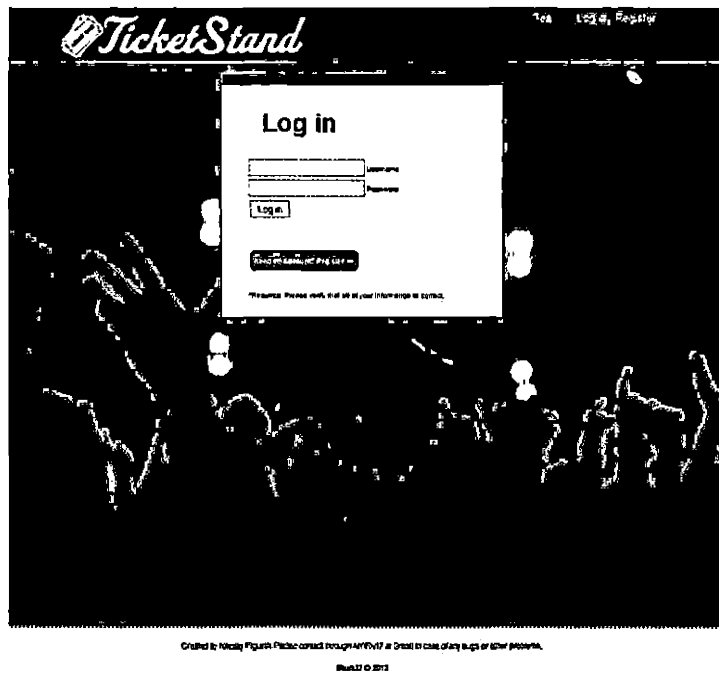


Fig. 2.4: TicketStand Login Page.

The header of the service holds four links. The links in the header are in the logo itself, which is a static asset and redirects to the front page of the service, and three dynamic buttons. If the user has an account and is logged in, they will see their name, a link to search, a link to the account dashboard, and a link to log out of the service. If the user is logged out, they will see the search link, a link to log in, and a link to register (or create a new account). The login page (seen in Figure 2.4) holds four elements specific to it. The form has a username field and the password field as well as the login button for that form. On submission of the form, the input is validated to check against an array of rules such as length requirements and if the user does indeed exist with that password. A green Need an Account? Register button may be found below the form. This button will redirect the visitor to the registration form. Once on



Fig. 2.5: TicketStand Registration Page.

the registration form (seen in Figure 2.5) , the user will see all required fields to create a new account. This includes the fields for username, password, repeat password, first name, last name, email, website (optional), and zip code. All fields have their own validation on submission. The validation rules include checks for what type of input is received (numeric or not), the values of the numeric input (specifically if the zip code is valid), if the email is of a proper structure (contains at least one character before the @ symbol and is followed by a domain name, and if the password fields match. Under this form we have another button not unlike the one seen in the login screen to give the visitor the option to go directly to the login form. On creation of a new account, the user is redirected to the login form where they may see a message reading Account Created Successfully! and sign in. After signing in, the user will see



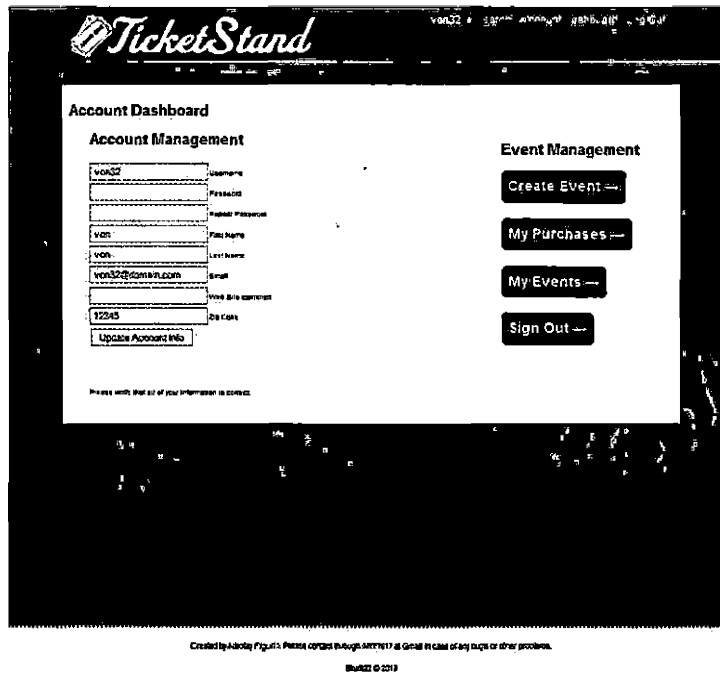


Fig. 2.6: TicketStand Account Dashboard.

the account dashboard.

The account dashboard (seen in Figure 2.6) includes two main portions- the user account management on the left and the event management on the right. The account management portion of the dashboard includes all fields that are stored in the account and were required upon registration. These fields are prepopulated from the server and allow the user to update their account information with one click. The form has its own validation and allows the user to change their password, username, and any other fields they wish. The event management portion of the account dashboard holds a collection of links. These links include the option to create an event, a link to the users purchases (this will not be seen if the user has no purchases on their account), a link to the users created events, and a sign out option. The user may

wish to select the link which would allow them to create an event. This redirect will take them to the event creation form. Here the user may find blocks of fields labeled as basics, location, descriptions, photo upload, tier 1 tickets, tier 2 tickets, and tier 3 tickets. The basics block contains the fields for the event name, the three types that the event should be listed under, and the date and time of the event. The location of the event includes the events location name, street address, city, state abbreviation, and the five digit zip code. The descriptions block allows the population of the event summary and the event description. The photo upload block allows the uploading of three different images to be displayed on the event page by browsing to the file and selecting it via a popup browser. The tier 1, tier 2 and tier 3 ticket blocks allow the user to specify the tier name, tier description, number of tickets, and price of each ticket per tier. This view holds its own validation rules checking for types of input, and looking over file size, as well as recovering all entered data in case the event creation did not go through properly. This allows the user to continue fixing any fields without having to repopulate the form from the beginning.

After the event is created by the user, the user may see it under the My Events page (seen in Figure 2.7) that is accessible via the account dashboard. This view holds a list of all events created by the user in a manner not unlike how the search function displays them. The primary difference in this view is that each stub form has a Remove Event button. Clicking this button will take the user to a confirmation page asking them if they truly wish to remove the event with the option of removing and keeping. The remove option deletes the event and all of its relevant information, while the keep option redirects back. The event view is comprised of a few dynamic

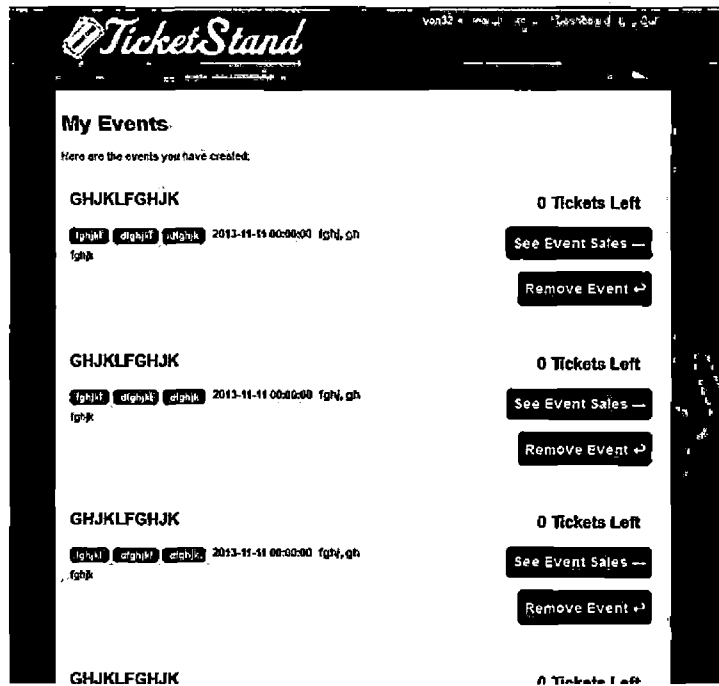


Fig. 2.7: TicketStand's My Events Page.

elements that are hidden to an unregistered user. If the event creator is on the event view, they may scroll down to see a Remove Event button and a Ticket Sales field. The ticket sales field will either read No Tickets Sold if there are no sales for the event, or will display a scrollable field with unique ticket codes that denote a users ticket. Each code has its own button (or flag) which reads Mark or Unmark where Mark suggests the ticket is not yet marked, and is therefore unused, whereas Unmark denotes that the ticket has already been redeemed. This view will refresh with each mark action, thus allowing multiple event attendance coordinators to use the same component in synchronization and see near real-time changes between the statuses of the tickets.

Another dynamic element on this view is the addition of an Edit this Event button

**TicketStand**

**Edit Event Details** [View Event Page](#) [Go to My Events](#)

**Basics**

Event Name / Page Title

Event Type - Category, Role, Role, or anything Custom / For Searchable

Date & Time

**Location**

Location Name

Street Address

City

State / Province

Country

Postal Code

**Descriptions**

Event Summary - This will be seen at the page for attending users (100 Characters)

Event Description - This will be used for the event page

Fig. 2.8: An Edit Event Details View in TicketStand.

which may be seen right under the Buy Tickets button. Selecting this option will take the user to the Edit Event Details view. This view (seen in Figure 2.8) is similar to the event creation form with the difference of saving the changes instead of creating a new event, as well as pre-populating all of the data in the event. The validation holds up not unlike the event creation page validation, and all input data is stored and recovered in the case of an error. This view also adds three new actions that the user may perform. The user may go to view the event page, go to the list of their events, and may delete the event. This concludes most of the functionality that a seller or event organizer would require. Administrators receive the same functionality with additional privileges such as being able to edit and delete any events in the system

and seeing an Administration Panel in the event views. The administration panel allows administrators to set other users as elevated to the administrator role or to demote them from the administrator status. The panel also allows an administrator to delete the users account, or to lock their account. The usefulness of locking an account may be exhibited in a scenario where a user may create a fraudulent event, and the administrator wishes to disable purchases (by changing event date or tickets available) and freezing the users account to leave all of their information on the page. Please recall that the event creators information is always displayed in the bottom of the event page. The frozen user will not be able to log in, and will therefore be unable to edit their information to conceal it, nor delete the event or account altogether, allowing storage for records, or simply displaying the information for the sake of the affected customers. Administrators cannot demote themselves from the administrator status. All users have the ability to not only create events, but also purchase tickets.

The purchasing mechanism may be seen by selecting the Buy Tickets button on any event page that has tickets and the event date has not passed. The redirect will take the user to a purchase selection form. At this point, a full implementation of this project for commercial use may swap this element out with a payment gateway as the existing code uses database calls to simulate payment transactions as it is unsafe to have a sandboxed PayPal API frame in the development environment that is accessible by the public. The purchase selection view is composed of the ticket tiers that are available in the event, the event name, tier summaries, a dropdown selection to choose the amount of tickets that the user wishes to purchase, along with the price

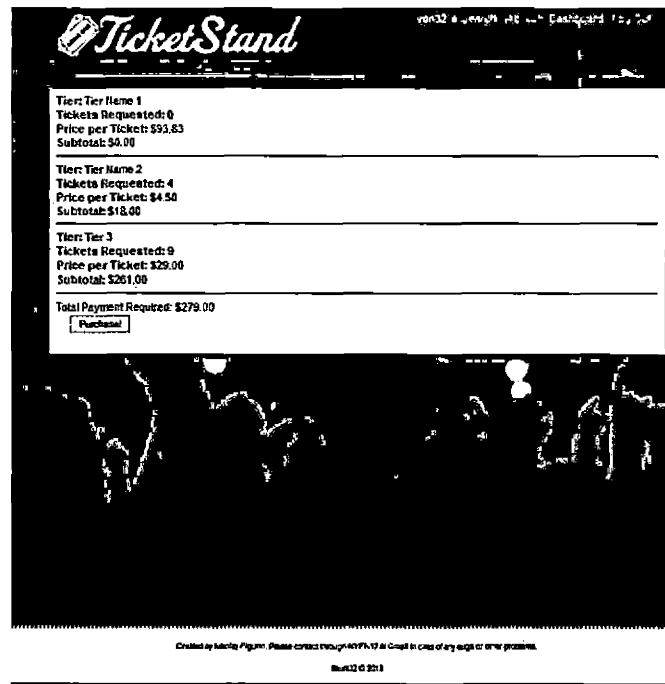


Fig. 2.9: Summary of Requested Tickets in a Purchase.

of each ticket. Once the number of tickets is selected and the form is submitted, the user may go through a summary of their purchase before committing to the payment. The summary lists the tier name, tickets requested, price per ticket, and a subtotal as seen in Figure 2.9) . Once payment is complete the purchaser will be redirected to the My Purchases view (seen in Figure 2.10) , where they will see stub forms comprised of the event name, a specific ticket code, a status for the code (used or not used), city and state of event, and the event summary. The stub forms also include a button to see the event page for further review. The user may then log out of the system and see the same information again on their next login.

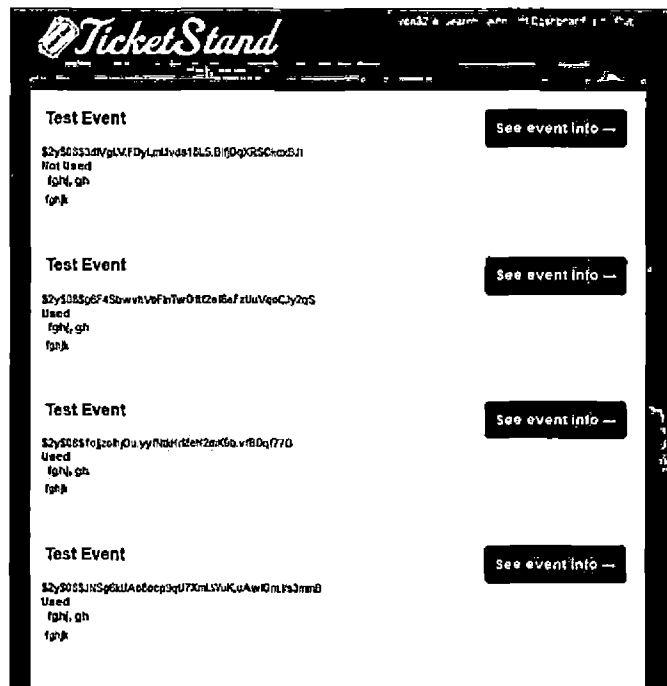


Fig. 2.10: A List of All Purchased Tickets By a Logged In User.

### 3. DATABASE DESIGN

The web application relies on a MySQL database with a schema that is optimized for the Eloquent Object Relational Mapper supplied by the Laravel 4 framework. Object-relational mappers allow for different sources and destinations of data to communicate with each other through a virtual interface that both may understand and stay compatible with. The Eloquent ORM is based around the ActiveRecord API[?] and provides a simple way of interfacing the logic components of the application with the database as well as supplying a schema builder class that allows the developer to specify what tables and columns must be created in a PHP-like syntax without the need for raw SQL commands (although they are possible). The database design revolves around the utilization of the Eloquent ORM as it enables easy chaining of tables with the minimal use of pivot tables and provides a very flexible interface that continues to function properly even after changes to the database are made.

The schema for the application is comprised of eight tables with at most a one to many relationship as seen in Figure 3.1. The schema is made of the users table, purchases table, transactions table, tickets table, tiers table, events table, images table, and the locations table. The schema itself is circular in design due to the fact that the users table must connect in both directions to purchases and events as a single user may purchase and create an event. The users table holds all required



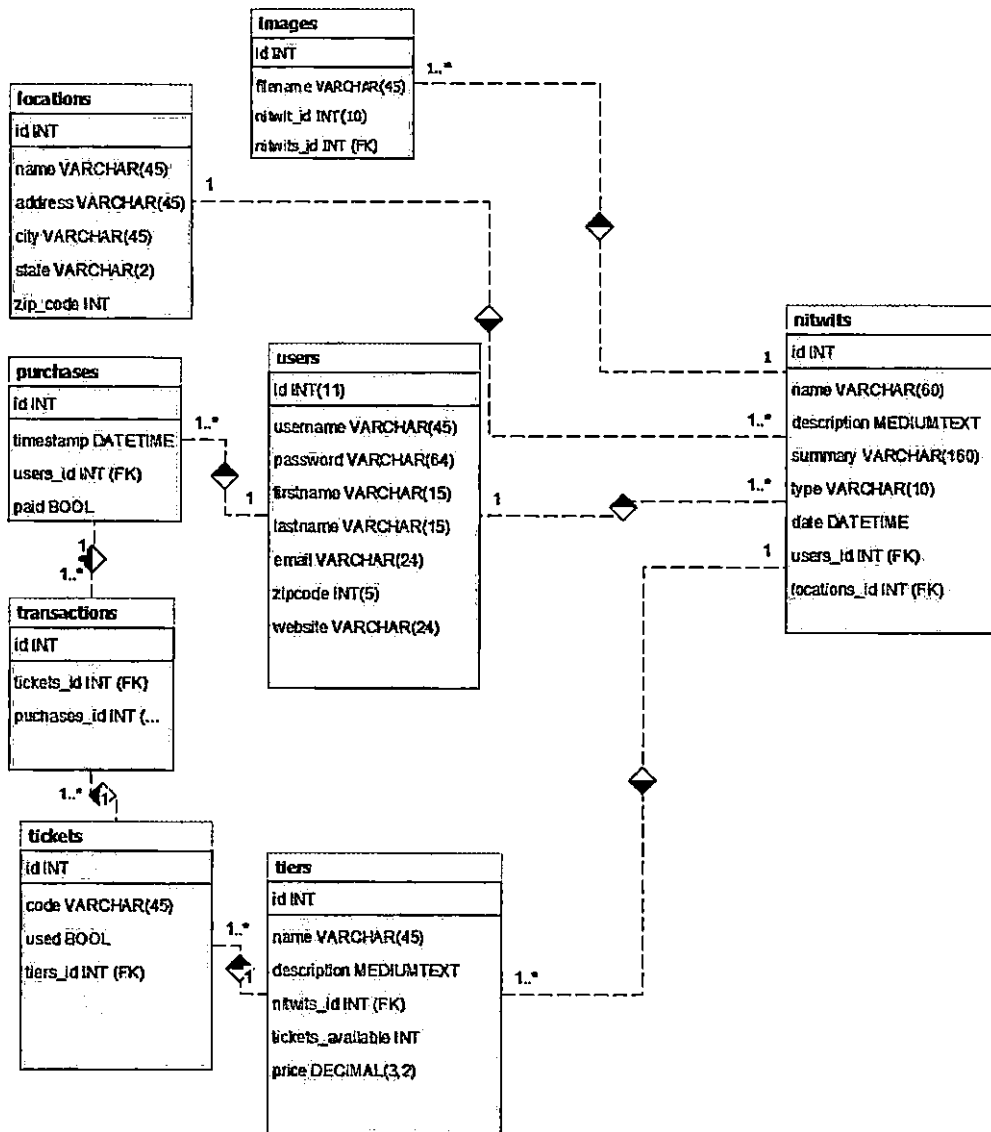


Fig. 3.1: Ticketstack Database Schema from TicketStand.

data for the user accounts including flags for account suspension and administrative privileges, as it holds a one to many relation to the purchases table. The purchases table represents the state of a purchase before the payment has gone through. It holds the timestamp for the purchase, the users id, and a Boolean value to denote if the purchase was paid for or not. This allows for the possibility of running a cleanup script on each purchase to delete any purchases that are not paid for within a set amount of time. This script would run in the database itself, but may be enabled in the application as a PHP function. Once the purchase is paid for, the tickets for that specific event are created and tied to the user through a transactions table.

The transactions table serves as an intermediary table between the two to disallow the performance pitfalls of a many to many relation to occur by storing the tickets id and the purchases id in a single row. The tickets table lists a specific code which is created by a PHP function that takes the current time down to a millisecond and hashes it, a Boolean value denoting if the ticket has been used or not, and a tiers id which acts as a foreign key to the tiers table. The tiers table includes the tier name, description, event id, the tickets available, and the price of the tier. This table belongs to one event through a many to one relation. The event table holds the event name, description, summary, type, date, and three foreign keys. The foreign keys point to the users table, the locations table, and the images table. The images table currently only holds the values for the event that the images are tied to, the user that uploaded the image, and the location of the image.

While it may seem unnecessary to abstract images into its own table, it is important to note that the platform as a service host, Pagodabox, that the project is initially

made to run on allows persistent, non-volatile file storage, thus only requiring a name and location of the file. However, a majority of other cloud hosting solutions do not offer non-volatile storage options, and therefore the abstraction of an images table aids in uploading images on such a platform as the only change would be an addition of two columns to the table to allow the image files to be stored directly in the MySQL database. The locations table has also been abstracted from the event table because of future plans to allow users to select previous locations they have held events at, thus potentially minimizing overhead in the database. Generally the Eloquent ORM allows for quick traversing of all tables with a minimal use of pivot tables, and therefore a circular design with predominantly single relations between tables was optimal[12].

## 4. SYSTEM IMPLEMENTATION

This chapter goes into the details of the systems implementation. It covers the portions of the project that allow the functionality that may be exhibited by the interface of the application, as well as cover some basic patterns seen in the Laravel 4 framework and any other tools that are included in it. An important bit of information about the framework in question is its use of the MVC (model-view-controller) pattern. This pattern allows for easy separation and management of the logic, data, and visual components of a system[1]. It is also important to be familiar with the LAMP (commonly Linux, Apache, MySQL, and PHP) software bundle which follows ANSI SQL standards in conjunction with support for popular languages like PHP (in our case)[2].

### *4.1 General Structure of the System*

The general structure of the system relies on three predominant components as the Laravel 4 framework is based on the Model View Controller pattern, and runs on most popular hosting environments as it expects a Linux Apache MySQL and PHP stack. In general, all traffic first goes through the the routes portion of the code. The routes file is a collection of get, post, and put method-based redirects that set a flag to complete a specific check or filter before routing the requests over to a specific

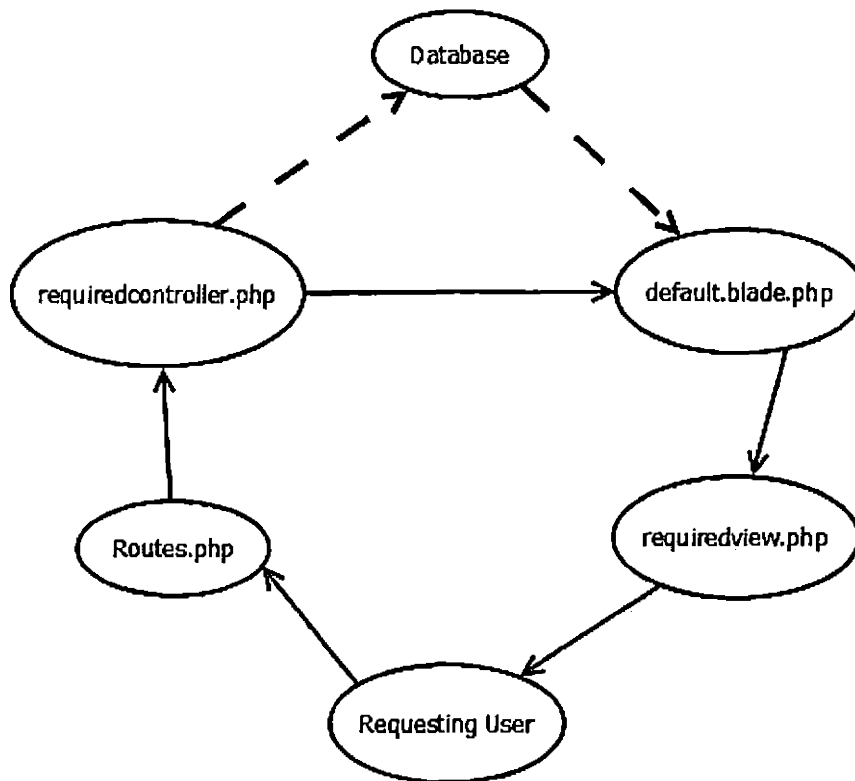


Fig. 4.1: TicketStand System Structure Detailing a Request and Response Scenario.

view and execute a specific controllers methods. An example of a general request and response cycle in the application may be seen in Figure 4.1.

The routing mechanism in Laravel 4 is extensible enough to allow filters, before methods, and route grouping, thus giving the developer plenty of functionality in what should be done before the request is considered as complete. In this projects case, we may see that the routes will differ depending on the get and post method, not strictly the URL name supplied. This allows for passing of variables through the session files, which are stored in the form of a flat file (as opposed to a cookie).

The storage of a session as a flat file also aids in the checks against cross-site request forgery[13], something that is discussed in another chapter. Routes however

may pass data through query strings as well, which would generally create security issues by design. However, the concerns are mended with the use of Auth class filters. Generally, the routes are set up to check if the URL requested should be a member-only piece of the system, or if its publically visible. We also check for the type of request and make sure that no POST requests can be forced into the system through the use of before filters. Some of the main filters use the Auth class to determine if the user is logged in or if they are the owner of an event, and a similar check to determine if the user requesting the page is an administrator. Once the requests go through and are cleared by the routes component, the controllers come into play. Each controller has its own family of methods that relate to each other in functionality. This is done primarily for organization as most of the methods are not tied to each other directly, and do not extend each other through imports or inheritance.

The main controllers are AccountController, AdminController, BaseController, EventController, KeywordController, MyEventController, MyPurchaseController, PublicController, and PurchaseController. Each controller has a self-explanatory title to best communicate what it does. The My prefix refers to the controllers that are specific to a tied user, whereas the controllers lacking the prefix are those that are used by all users, and do not pass data per-session, per-user. This is also the step where all database related processing occurs. Through the use of the Eloquent ORM, the system may retrieve all necessary information to feed into the view by passing the information through a series of arrays, or using the static save and update methods to update any tables that require changes. It is also possible to force SQL statements into the database directly through the controller, but this is not seen in this project.

Once the data has been processed and written back to the database through the use of the controllers, most controllers will either redirect the user to a GET function that will render a view, or they will render a view outright. The views available in the application are what displays the basic HTML structure of the page, as well as doing some minor processing work. Primarily, we will see the processing portions of the view strictly perform more authenticity checks and in some cases queries through the Eloquent ORM.

The views in the application use the Blade templating engine. This system allows for shortcodes to echo out specific variables, form creation, and a multitude of loops with as little as two characters of code[14]. This is commonly seen throughout the project for form creation using the Form class. The Blade templating engine also allows the developer to separate the structure of the pages into multiple sections and re-use it in other views. This is how the header, footer, and metadata is passed on each page throughout the project. The specific setup relies on a blade file called `default.blade.php` which holds the header, footer, and several variables that may be passed to the head and script tags of the file. In the middle of the `default.blade.php` file, we have a `yield` for content. This `yield` is a marker for where the rest of the code will be imported in each page, depending on the view being rendered. This allows for quick changes in the system to be reflected system-wide, as well as increasing readability of code as the structure is generally object-oriented[15]. Lastly, the basic structure relies on the `public` folder which hosts what is actually seen as the root of the server. This is one of Laravels best points, and is something that increases the integrity of the applications security by hiding more core files than most PHP-based

frameworks normally do. The public directory is also home to any uploaded files by all users (as it may be seen publically anyway during upload), and the single CSS file that drives all styling of every view.

## 4.2 Scenario-Specific Workflows

In the scenario of fetching a public page, the controller simply creates a basic data array containing the user id of the visitor (if they have one) and the title of a page and passes it to the view component that renders the page. In the case of the event creation component of the system, the route may fail the authentication of the user as being the owner, and will therefore redirect the user to the account dashboard with an error message. However, if the authentication does not fail, the route will use the `showeventcreation` method from `EventController` to generate the view `eventcreation` with an array of basic data such as user id and title, as previously explained. If the same page is loaded with a legitimate use of a POST method instead of a generic GET method, we will see the utilization of a different method under the same controller. This scenario will have the routes call up the `posteventcreation` method under the same controller. This method once again does the Auth method check to see who the user pushing the POST is, and if they can create such an event. The method grabs all input from the referrer screen (specifically the event creation form) and checks it against the rules array. The check here not only ensures that no extra data is being forced through the method, but also does the validation tasks. A few examples in the rules array may be the requirement of the event summary to be required, alphanumeric, and a maximum of 160 characters, and the requirement of the images



attribute to be a set of specific extensions and filetypes, as well as specifying a hard limit of 300KB in filesize prior to being uploaded. The method also ensures that all data is temporarily stored and will get re-populated in the event of an error, so that the user will not have to start the event creation process all over again.

Once the validator completes and passes, we will fill a series of arrays (specifically `locationdata`, `eventdata`, `tierdata`, and `images`) which will later be validated once more and pushed through to the Eloquent ORM. This portion of the controller will specify any new methods and update methods that will be passed to Laravels Schema Builder component for creation and updating of the controllers respective tables. While reviewing the code, one may notice that the event variables are interchangeable with the `nitwit` variable due to the fact that name spacing support was not completely implemented in Laravel 3[16], and was therefore split into two different variables as this project began while version 4 of the Laravel framework was not released. The Eloquent ORM not only does any specified checks in its owner controllers, but will also go through the migrations provided in the project to check against any constraints or inconsistencies in the data push. This requires the ORM to go through the models supplied in the workflow to check what the structure and relations of the database tables are. This allows for the use of a constraint-free MySQL database setup, where the application strictly relies on the Eloquent ORM. However, this project does not get rid of the checks performed within the MySQL server itself, and therefore checks over both constraints in the database itself, as well as the eloquent ORM through the use of models.

The migration option provided by the Laravel 4 framework allows for automatic

creation of a database during a push, as well as seeding in tester events and users. In the case of the project, this strategy is used to create an administrator with the username of Administrator and the password of Password upon the installation of the application. The current setup also populates the event tables with four fake, and incomplete events just for the sake of running constraint checks through the MySQL database upon install. The migrations portion of the project are a large part of what makes this application so portable. The fact that the application may be pushed from a git repo onto nearly any LAMP stack is owed to the use of migrations as they will create the database in a matter of seconds. Each migration creates what could be considered an extremely detailed model of a specific table and its relations. This is the equivalent of using the PHP language for database creation over straight SQL statements. Another large part of why the system is so portable is the use of the configuration files which are all consolidated in the config folder of the application[17]. These files allow the user to specify what database name and type they wish to use, if they want to include any imported packages (by default, a profiler is enabled to gauge performance of SQL queries and page loadtimes), and other options such as SSL encryption. In the scenario of purchasing a ticket through the system, the workflow goes through routes to check for the users authentication token and id, and pushes to a GET method of the PurchaseController controller.

The authentication portion reads to a flatfile and takes note of the IP that the user is requesting from. This flatfile has a lifetime of a half hour and expires if the session is terminated, unless another newer token is requested. The controller renders a view, which in turn renders the forms required to select the event tickets, all styled by the

CSS file tickets.css which is located in the public directory of the application. The controller redirects once a POST method is called on the form, and pushes the pulls the information via an Input class to create a new view that holds a small amount of processing in the view to display the subtotals per-tier and the total cost of all tickets in the cart. Once the purchase method is pushed POST, we will see the Eloquent ORM write back to the database by marking the paid Boolean value as true, therefore checking the rest of the controller into creating the tickets required and tying the two tables via the transactions intermediary table. As the tickets are created, the date method grabs the current time down to the millisecond and hashes it through into a long, but unique code that may be later used as the ticket reference code. Such a task is also possible through the migrations in the project, as a similar principle is used in creating the administrator account. The password Password is hashed during the migration, thus providing a secure password without changing it each installation. Due to the fact that all login helpers created in the project do a hash compare as opposed to a plaintext compare, this is the optimal choice for both password and ticket. Again, the hash key is set in the config directory of the application.

## 5. SYSTEM VALIDATION AND SECURITY

Computer system validation (or CSB) is the process of assuring that a system does what is required of it without fail as part of its development cycle[3]. In the case of this application, the installation steps include validation checks right in the system upon deployment. Upon completion, it is safe to assume that all components work if the front page may load (the front page accesses multiple parts of the system just to ensure this check by design). This is possible due to the use of migrations, which are a type of version control aspect and auto-population tool of a database[4] and the Eloquent ORM. The migrations created in this application are written with their own specific syntax as part of the Laravel specification. Security is also a main focus of the application. Primarily, we rely on security tokens for session validation, as well as rule-based logical checks, database constraints, and route filters for data passing and writing. Security tokens allow for a check of someone's identity via the use of data specific to the client communicated with[5] while the other checks ensure that no extra data (or incorrect data) is passed through the system.

One of the primary focuses of this system was the aspect of its security. The Laravel 4 framework allows for many different ways of doing validation and authentication checks, but for the sake of the project, most of the methods and actions go through three or more checks before completing. An example of such a scenario would be the

event creation process, where we have the route be the first layer of protection by using a filter and the Auth class to confirm if the user is registered, where the user is coming from, and if they are an administrator or own the event (in the case they wish to edit it). The second layer comes in the controller. Before any read or write logic is processed, the controller often checks for the same authenticity by fetching the current user with a token. The token in question is per IP, per timeslot, and per request. This token disallows cross-site reference forgery attacks such as man in the middle backtracks by forcing a check between the client machine directly. In other words, the token is strictly valid for the machine of that IP, timeslot, and request, and will not work for any other machines.

After the controller has completed all checks, the absence of raw SQL query interpretation in the code adds another layer of security by making SQL injections impossible due to format inconsistencies. This bleeds over into another check in the models section of the application where the ORM will complete all constraint checks of the information that was passed in to flag any unrequested data, or incorrect values. The controller also often runs through a specified set of validation rules, as well, to ensure that the data is clear before the call to the Eloquent ORM is even made. On top of all this, we still have the benefit of all constraint checks being performed on the MySQL server directly, disallowing any inconsistencies or broken rules in the data being passed once more[18]. Finally, we exhibit the views themselves as often having both validator functions and authentication components to ensure that the user is indeed who they say they are, and that the data being passed fits all requirements. These multiple layers of security bring the application closer to the level of

security required for public use. It may be apparent that other means of cracking, social engineering, and bruteforcing may be exhibited, as well as the host servers being taken over (which is outside of the application's control), but most, if not all of these concerns are easily addressable with further expansion of the application. It is always important to remember that any application may be compromised even if it may be deemed as "unhackable". This is also a major reason that there are options to abstract the database and payment system entirely. Moreover, it is possible to test all of these features with the use of migrations and seeding.

Upon installation, the migrations will automatically create all tables in the database. The migrations specify the types of variables that are pushed through, along with their relationships, and down to the details such as what engine the table should use. The seeding portion of the migrations runs after all tables were created and linked together through the models and foreign key constraints to ensure that the proper data may be pushed through. If the seeding portion of the application throws an error, then there is something incorrectly configured on the server environment itself. Whereas if the seeding goes through, then most basic security checks should function as expected.

## 6. INSTALLATION AND MAINTENANCE

The installation of the system is simple enough to only require basic knowledge of creating user accounts in online services and a general understanding of navigating an MVC-based application. The primary concern is the basic understanding of cloud computing and platforms as a service. While the cloud may be described as a large and distributed server farm chain[6], platforms as a service (PaaS) may be seen as access to the cloud through a middleman that manages the software bundle and cloud configuration for you[7]. However, little background is required to use such a service as they are generally extremely simple due to the utilization of a user interface with inline documentation.

### 6.1 *Installation*

The basic installation of this product revolves around using a platform as a service host such as Pagodabox or AppFog to install a one-click configuration of Laravel 4 onto the server, and set up a git repository with that host. Alternatively, private hosting may be used for running this application, but will require the installation of Laravel 4. The installation process for Laravel 4 simply consists of downloading the Laravel 4 framework from the official repository and extracting the Laravel 4 codebase into the host directory of the server and changing the secret key located

in the config file. In the case of using a platform as a service such as Pagodabox or AppFog, the user may create a new account, create a new instance from the service's account dashboard, and deploy a new instance of Laravel 4. Every platform as a service is different, but generally the user will face the requirement of linking a git repository to their platform of choice by using the platform's linking tool to generate a new RSA key and pulling in the repository to their local machine. Once again, it is important to change the secret key found in the config file of Laravel 4. This file is what makes the hashing function work, as well as what provides a modifier for the random function. Once the repository is set up, the administrator may copy all of the files of the project into the Laravel installation directory. After extraction, the administrator must install the composer, a minimal dependency manager[19], on their server, and run the command `php composer update` to fetch any required dependencies.

After the basics for the PHP server are set up, be sure to configure the MySQL server to allow file uploads via the `php.ini` configuration file. This file will also contain the possibility of using SSL features on the server, if desired. Next, it is important to navigate to the directory of the application and run the `php artisan migrate` command. This migrate command will take all of the migrations in the `/database/migrations/` directory and populate the MySQL database with the required tables and seeds[20]. At this point, the installation is generally done. The MySQL name may be changed from `ticketstack` to whatever is desired via the `config/database.php` file, as well as the type of database you wish to use- although only MySQL is tested as of this time.

This concludes the basic installation of the TicketStand web-application. All up-



dates to the system may be pushed through a git push and onto a platform as a service which will automatically start a new instance of the server with the base Laravel 4 framework on it, fetch all dependencies via the Composer dependency manager, clone the database over, reroute traffic to the new instance, and finally decommission the previous infrastructure.

## 6.2 *Maintenance and Scaling*

This project requires minimal maintenance due to the nature of the Eloquent ORM and how it is used. There is virtually no overhead created in the system that is not utilized. If the administrator wishes to, they may create two scripts to periodically flush out the images table and the purchases table. However, it is of utmost important to note that only the entries that have the paid Boolean marked as a null value should be marked for script deletion, not those with a value of 1. This is because paid purchase entries are used to link the existing tickets and transactions over to the users table. The automation should only remove unnecessary purchases that were never paid, and are therefore unlinked to any valuable information. Due to the fact that the project was designed with platforms as a service in mind, the only changes required for proper scaling may be done through the control panel of the server that the application is hosted on. In the case of using Pagodabox, scaling should require no changes outside of throwing more hardware at the application, and allowing the infrastructure to take care of all rebalancing[21]. It is heavily urged to increase the database capacity resources allocated to the database first.

## 7. CONCLUSION AND FUTURE DIRECTIONS

### 7.1 *Conclusion*

The project discussed in this work solves multiple issues that event organizers regularly face when utilizing the services that are readily available. Outside of being a non-centralized platform, TicketStand also delivers on new features such as event creation from each user, and an administration panel with elevated role management. The project has been built with security, expandability, and portability as priorities, and has delivered in all fields. More importantly, the application has a real-world impact as it may be used by organizers from the very first day of its public release. The application also affects a large spectrum of users as it is suitable for extremely small events such as parties with under a dozen people all the way up to massive festivals with thousands of attendees. Personally, the project has provided a deep understanding of PHP, CSS, basic HTML, JavaScript, the jQuery library, Laravel, Composer, Eloquent, Fluent, MySQL, back-end PaaS management, the Git version control system, and many other tools that are a necessity in web application development. The project has also forced me to become comfortable with the unknown through the discovery of multiple bugs found in the Laravel framework during the time of development. This knowledge will stay with me forever and will prove invaluable in my future endeavors.

## 7.2 *Future Directions*

As previously mentioned, the project was built with expandability in mind. The commitment to expandability was of the utmost importance due to the plans of adapting the system for other needs such as table reservations, appointment scheduling, and more robust functionality as an event ticketing platform. Initially the project was to be rewritten due to the multiple rewrites taking their toll on the integrity of the structure, but it has become apparent that the current state of the application allows for continued work on this iteration as opposed to requiring a complete rewrite. Some immediate features that will be implemented include emailing lists, email notifications, barcode and QR code implementation, per-seat ticket sales, and enhanced search engine optimization. There are also plans to enhance the mobile experience as well as allow more customizability through the frontend of the application, as opposed to having to make code changes in the project itself upon setup.

APPENDIX A  
ROUTING CONFIGURATION FILE

```
<?php
```

```
Route::get('/', 'PublicController@showfrontpage');
```

```
Route::get('/search', 'PublicController@showsearch');
```

```
Route::get('/search/results',  
array('uses' => 'PublicController@postshowsearch', 'as' => 'post.  
show.search'));
```

```
Route::get('events/{id}', 'EventController@showEvent');
```

```
Route::get('/registration', array('before' => 'guest', 'uses'  
=>  
'AccountController@showregistration'));
```

```
Route::get('/login', array('before' => 'guest', 'uses' =>  
'AccountController@showlogin'));
```

```
Route::post('/registration', array('before' => 'guest', 'uses'  
,  
=> 'AccountController@postregistration'));
```

```
Route::post('/login', array('before' => 'guest', 'uses' =>
'AccountController@postlogin'));
```

```
Route::filter('guest', function($route, $request)
{
if (Auth::check())
{
return Redirect::to('/accountdashboard');
}
});
```

```
Route::filter('csrf', function()
{
if(Request::forged()) return Response::error('500');
});
```

```
Route::get('/accountdashboard', array('before' => 'auth|user'
,
'uses' => 'AccountController@showaccountdashboard'));
```

```

Route::get('/myevents', array('before' => 'auth|user', 'uses'
    =>
'MyEventController@showmyevents'));
Route::get('/myevents/{eventid}', array('before' => 'auth|
    user',
'uses' => 'MyEventController@showeventdetails'));
Route::get('/myevents/{eventid}/edit', array('before' =>
'auth|user', 'uses' => 'MyEventController@editmyeventdetails'
));
Route::post('markTicketUsed/{ticketid}', array('before'=>'auth
    |user', 'uses'=>'EventController@markTicketUsed'));
Route::post('markTicketNotUsed/{ticketid}', array('before'=>'
    auth|user', 'uses'=>'EventController@markTicketNotUsed'));

Route::post('updateStatus', array('before'=>'auth|user', 'uses'
    =>'MyEventController@updateStatus'));
Route::get('/event/creation', array('before' => 'auth|user',
'uses' => 'EventController@showeventcreation'));
Route::get('/purchaseselection', array('before' => 'auth|user
    ',
'uses' => 'PurchaseController@showpurchaseselection'));

```

```
Route::post('/purchaseprocessed/{purchaseid}', array('before' =>
    'auth|user', 'uses' =>
    'PurchaseController@postpurchaseprocessed'));
```

```
Route::get('/mypurchases', array('before' => 'auth|user', '
    uses'
=> 'MyPurchaseController@showmypurchases'));
```

```
Route::get('/mypurchase/{ticketid}', array('before' =>
    'auth|user', 'uses' => 'MyPurchaseController@showmypurchase')
);
```

```
Route::get('/myevents/{eventid}/delete', array('before' =>
    'auth|user', 'uses' => 'MyEventController@eventdeletion'));
```

```
Route::get('/myevents/{eventid}/deleted', array('before' =>
    'auth|user', 'uses' => 'MyEventController@posteventdeletion')
);
```

```
Route::post('events/{id}/purchase', array('before' =>
    'auth|user',
    'uses'=>'EventController@postpurchaseEventTickets', 'as'=>'
    post.purchase.event.tickets'));
```

```
Route::get('events/{id}/purchase', array('before' =>
    'auth|user',
```



```

'uses'=>'EventController@purchaseEventTickets', 'as'=>'
    purchase.event.tickets'));

Route::post('/myevents/{eventid}/edit', array('before' =>
'auth|user', 'as'=>'postedevent', 'uses' =>
'MyEventController@posteditmyeventdetails'));

Route::delete('myevents/image/{imageid}/delete', array('before
=>'auth|user', 'as'=>'delete.event.image', 'uses'=>'
MyEventController@deleteImage'));

Route::put('/accountdashboard', array('before' => 'auth|user'
,
'uses' => 'AccountController@putaccountdashboard'));

Route::post('/myevents', array('before' => 'auth|user', 'uses
,
=> 'MyEventController@postmyevents'));

Route::post('/myevents/{eventid}', array('before' =>
'auth|user', 'uses' => 'MyEventController@postmyeventdetails'
));

Route::post('/event/creation', array('before' => 'auth|user',
'uses' => 'EventController@posteventcreation'));

Route::post('/purchaseselection', array('before' => 'auth|
user',

```

```

'uses' => 'PurchaseController@postpurchaseselection'));
Route::get('/purchaseprocessed', array('before' => 'auth|user',
    'uses' => 'PurchaseController@postpurchaseprocessed'));
Route::post('/purchasepayment', array('before' => 'auth|user',
    'uses' => 'PurchaseController@postpurchasepayment'));

Route::post('/mypurchases', array('before' => 'auth|user',
    'uses' => 'MyPurchaseController@postmypurchases'));
Route::post('/mypurchase/{ticketid}', array('before' =>
    'auth|user', 'uses' => 'MyPurchaseController@postmypurchase')
    );

Route::filter('user', function($route, $request)
{
    if (Auth::guest())
    {
        return Redirect::to('/login');
    }
});

```

```

Route::get('/adminuser/{creator}/makeadmin', array('before'
    =>
    'auth|admin', 'uses'=>'AdminController@makeadmin'));
Route::get('/adminuser/{creator}/demoteadmin', array('before'
    =>
    'auth|admin', 'uses'=>'AdminController@demoteadmin'));
Route::get('/adminuser/{creator}/unlock', array('before' =>
    'auth|admin', 'uses'=>'AdminController@unlockuser'));
Route::get('/adminuser/{creator}/lock', array('before' =>
    'auth|admin', 'uses'=>'AdminController@lockuser'));
Route::get('/adminuser/{creator}/delete', array('before' =>
    'auth|admin', 'uses'=>'AdminController@deleteuser'));

Route::filter('admin', function($route, $request)
{
    if (Auth::user()->admin != 1)
    {
        return Redirect::to('/');
    }

});

```

```
Route::get('logout', function() { Auth::logout(); return  
Redirect::to('login');});
```

APPENDIX B

ACCOUNT CONTROLLER CLASS

```
<?php
```

```
class AccountController extends BaseController {
```

```
public $restful = true;
```

```
public function showlogin()
```

```
{
```

```
$view = View::make('login');
```

```
$view->title = "Log-in";
```

```
return $view;
```

```
}
```

```
public function postlogin()
```

```
{
```

```
$username = Input::get('username');
```

```
$user = User::where('username',$username)->first();
```

```
$userdata = Input::only('username', 'password');
```

```
$rules = array(
```

```

'username' => 'required|between:2,45',
'password' => 'required|between:5,64',
);

$messages = array(
'required' => 'The :attribute field is required.',
'between' => 'The :attribute must be between :min and :max
characters.',
'unique' => 'The :attribute must be unique.',
);

$validator = Validator::make($userdata, $rules, $messages);

if($validator->fails())
{
return
Redirect::to('/login')->withErrors($validator)->withInput();
}

if ($user){
if($user->locked == 1){

```

```

return Redirect::to('/login')->with('message', 'Your Account
    Was
    Locked. ');
    }
}

```

```

if ( Auth::attempt($userdata) )
{

return Redirect::to('/accountdashboard');
}
else
{
return Redirect::to('/login')->with('login_errors', true);
}

}

```

```

public function showregistration()
{
$view = View::make('registration');
$view -> title = "Register";
return $view;
}

```



```
}
```

```
public function postregistration()
```

```
{
```

```
$inputs = Input::all();
```

```
$rules = array(
```

```
'username' => 'required|between:2,45|unique:users,username',
```

```
'password' => 'required|between:5,64',
```

```
'repeatpassword' => 'same:password',
```

```
'firstname' => 'required|between:2,15',
```

```
'lastname' => 'required|between:2,15',
```

```
'email' => 'required|email|between:6,24|unique:users,email',
```

```
'website' => 'max:24',
```

```
'zipcode' => 'required|between:1000,99950|integer',
```

```
);
```

```
$messages = array(
```

```
'required' => 'The :attribute field is required.',
```

```
'between' => 'The :attribute must be between :min and :max.',
```

```
'max' => 'The :attribute must be a max of :max  
characters.',
```

```
'min' => 'The :attribute must be a min of :min  
characters.',
```

```
'unique' => 'The :attribute already exists.',
```

```

'integer' => 'The :attribute must be a number.',
);

$validator = Validator::make($inputs, $rules, $messages);
if ($validator->fails()){
return
Redirect::to('/registration')->withErrors($validator)->
    withInput();
}
else
{

User::create(array(
'username'=>Input::get('username'),
'password'=>Hash::make(Input::get('password')),
'firstname'=>Input::get('firstname'),
'lastname'=>Input::get('lastname'),
'email'=>Input::get('email'),
'website'=>Input::get('website'),
'zipcode'=>Input::get('zipcode'),
));
return Redirect::to('/login')->with('message', 'Account_Made!
Please_log_in.');
```

```
}
```

```
}
```

```
public function showaccountdashboard()  
{  
  $id= Auth::user()->id;  
  $purchases = Purchase::where('user_id', $id)->get();  
  return View::make('accountdashboard')  
  ->with('purchases', $purchases)  
  ->with('title', 'Dashboard')  
  ->with('user', User::find($id));  
}
```

```
public function putaccountdashboard()  
{  
  $id= Auth::user()->id;  
  $inputs = Input::all();  
  $rules = array(  
    'username' =>  
    'required|between:2,45|unique:users,username,',$id,
```

```

'password' => 'between:5,64',
'repeatpassword' => 'same:password',
'firstname' => 'required|between:2,15',
'lastname' => 'required|between:2,15',
'email' =>
'required|email|between:6,24|unique:users,email,',$id,
'website' => 'max:24',
'zipcode' => 'required|between:1000,99950|integer',
);
$password = Input::get('password');

$messages = array(
'required' => 'The :attribute field is required.',
'between' => 'The :attribute must be between :min and :max.',
'max' => 'The :attribute field must be a max of :max
characters.',
'min' => 'The :attribute field must be a min of :min
characters.',
'unique' => 'The :attribute already exists.',
);

$validator = Validator::make($inputs, $rules, $messages);
if ($validator->fails()){

```

```

return
Redirect::to('/accountdashboard')->withErrors($validator)->
    withInput();
}
else
{

    if ($password){
DB::table('users')->where('id', $id)
->update(array(
    'username' =>Input::get('username'),
    'password'=>Hash::make(Input::get('password')),
    'firstname'=>Input::get('firstname'),
    'lastname'=>Input::get('lastname'),
    'email'=>Input::get('email'),
    'website'=>Input::get('website'),
    'zipcode'=>Input::get('zipcode'),
));}
    else{

DB::table('users')->where('id', $id)
->update(array(

```

```
'username' =>Input::get('username'),
'firstname'=>Input::get('firstname'),
'lastname'=>Input::get('lastname'),
'email'=>Input::get('email'),
'website'=>Input::get('website'),
'zipcode'=>Input::get('zipcode'),
));}
}
return Redirect::to('/accountdashboard')->with('message', '
Account_Info_Updated!');

}

}
```

APPENDIX C  
ADMINISTRATION CONTROLLER CLASS

```
<?php
```

```
class AdminController extends BaseController {
```

```
public function makeadmin($id)
{
```

```
DB::table('users')->where('id', $id)->update(array('admin'=>'
1' ));
```

```
return Redirect::back()->with('message', 'Promoted to Admin!
'); }
```

```
public function demoteadmin($id)
{
```

```
DB::table('users')->where('id', $id)->update(array('admin'=>'
0' ));
```

```
return Redirect::back()->with('message', 'Demoted from Admin!
'); }
```

```
public function unlockuser($id)
```



```

{

DB::table('users')->where('id', $id)->update(array('locked'=>
    '0'
    ));
return Redirect::back()->with('message', 'Account_Unlocked!')
    ; }

```

```

public function lockuser($id)

```

```

{

DB::table('users')->where('id', $id)->update(array('locked'=>
    '1'
    ));
return Redirect::back()->with('message', 'Account_Locked!');
}

```

```

public function deleteuser($id)

```

```

{
$user = User::find($id);
$user->delete();

return Redirect::to('accountdashboard')->with('message', "
    User's
    Account_Deleted!");
}

```

APPENDIX D  
EVENT CONTROLLER CLASS

```
<?php
```

```
class EventController extends BaseController {

public function showEvent($id)
{

$nitwit = Nitwit::with('images')->findOrFail($id);
if (Auth::user()) {
if ($nitwit->isOwner() || Auth::user()->admin == 1) {
$nitwit=Nitwit::with('images', 'tiers.tickets')->where('id',
    $id)->first();

$tickets = array();

foreach($nitwit->tiers as $tier)
{
foreach($tier->tickets as $ticket)
{
$tickets[$ticket->id]['code'] = $ticket->code;
$tickets[$ticket->id]['used'] = $ticket->used;
$tickets[$ticket->id]['id'] = $ticket->id;

```

```
}
```

```
}
```

```
return View::make('eventpage', array(
```

```
'title'=>'View_Event',
```

```
'event'=>$nitwit,
```

```
'tickets'=>$tickets));
```

```
}
```

```
}
```

```
return View::make('eventpage', array(
```

```
'title'=>'View_Event',
```

```
'event'=>$nitwit
```

```
));
```

```
}
```

```
public function showeventcreation()
```

```
{
```

```
$view = View::make('eventcreation');
```

```
$view->title = "Event_Creation";
```

```
return $view;
```

```
}
```

```

public function posteventcreation()
{
$userid= Auth::user()->id;
$inputs = Input::all();
$rules = array(
'eventname' => 'required|between:6,60 ',
'eventtype1' =>'required|max:10 ',
'eventtype2' =>'required|max:10 ',
'eventtype3' =>'required|max:10 ',
'datetime' => 'required|date ',
'locationname' => 'required|max:45 ',
'streetaddress' => 'required|between:4,45 ',
'city' => 'required|between:2,45 ',
'state' => 'required|size:2|alpha ',
'zipcode' => 'required|size:5 ',
'eventsummary' => 'required|max:160 ',
'eventdescription' => 'required ',
'images []' => 'image|max:300 ',
'tiername1' => 'required|max:45 ',
'tierdescription1' => 'required ',
'numberoftickets1' => 'required|max:1000|numeric ',

```

```

'priceofticket1' => 'required|max:999.99|numeric',
'tiername2' => 'max:45',
'numberoftickets2' => 'max:1000|numeric',
'priceofticket2' => 'max:999.99|numeric',
'tiername3' => 'max:45',
'numberoftickets3' => 'max:1000|numeric',
'priceofticket3' => 'max:999.99|numeric',
);

$messages = array(
'required' => 'The :attribute field is required.',
'between' => 'The :attribute must be between :min and :max
characters.',
'max' => 'The :attribute field must be a max of :max
characters.',
'min' => 'The :attribute field must be a min of :min
characters.',
'unique' => 'The :attribute must be unique.',
'integer' => 'The :attribute must be a number.',
'date' => 'The date is not in a valid format',
'image' => 'The file uploaded must be an image',
);

```

```

$validator = Validator::make($inputs, $rules, $messages);
if($validator->fails()){
return
Redirect::to('/event/creation')->withErrors($validator)->
    withInput();
}
else
{

$locationdata = array(
'name' =>Input::get('locationname'),
'address'=>Input::get('streetaddress'),
'city'=>Input::get('city'),
'state'=>Input::get('state'),
'zip_code'=>Input::get('zipcode'),
);

$eventdata = array(
'name' =>Input::get('eventname'),
'type1'=>Input::get('eventtype1'),
'type2'=>Input::get('eventtype2'),
'type3'=>Input::get('eventtype3'),
'date'=>Input::get('datetime'),

```

```
'summary'=>Input::get('eventsummary'),  
'description'=>Input::get('eventdescription')  
);
```

```
$tierdata1=(array(  
'name' =>Input::get('tiername1'),  
'description'=>Input::get('tierdescription1'),  
'price'=>Input::get('priceofticket1'),  
'tickets_available'=>Input::get('numberoftickets1'),  
));
```

```
$tierdata2=(array(  
'name' =>Input::get('tiername2'),  
'description'=>Input::get('tierdescription2'),  
'price'=>Input::get('priceofticket2'),  
'tickets_available'=>Input::get('numberoftickets2'),  
));
```

```
$tierdata3=(array(  
'name' =>Input::get('tiername3'),  
'description'=>Input::get('tierdescription3'),  
'price'=>Input::get('priceofticket3'),
```



```
'tickets_available'=>Input::get('numberoftickets3'),
));
```

```
$destinationPath = "uploads/".$id;
```

```
$locations = Location::create($locationdata);
```

```
$nitwit = new Nitwit($eventdata);
```

```
$nitwit -> location_id = $locations->id;
```

```
$nitwit -> user_id = Auth::user()->id;
```

```
$nitwit->save();
```

```
$bool = false;
```

```
$images = Input::file('images');
```

```
foreach($images as $image) {
```

```
if ($image) { $bool = true; }
```

```
}
```

```
if ($bool) {
```

```
foreach($images as $image) {
```

```
$name = date('YndHis');
```

```
$name.= $image->getClientOriginalName();
```

```

$uploadSuccess= $image->move($destinationPath,$name);
if($uploadSuccess) {
$imageData = array(
'filename'=>$name,
'nitwit_id'=>$nitwit->id
);
$image = Image::create($imageData);
}
else {
echo "died_uploading_due_to_dev_environment.";
}

}
}

```

```

$tier1= new Tier($tierdata1);
$tier1 -> nitwit_id = $nitwit->id;

```

```

$tier2= new Tier($tierdata2);
$tier2 -> nitwit_id = $nitwit->id;

```

```

$tier3= new Tier($tierdata3);
$tier3 -> nitwit_id = $nitwit->id;

$tier1->save();
$tier2->save();
$tier3->save();

$eventid = $nitwit->id;
return Redirect::to("/myevents/$eventid/edit")->with('message
    ',
    'Event_Created_Successfully!');
}
}

public function purchaseEventTickets($id)
{
    $today = date("Y-m-d_H:i:s");
    if (Nitwit::find($id)->date < $today)
    {
        return Redirect::to('/accountdashboard')->with('message', '
            Event
            is_over!');
    }
}

```

```

}
else {
$title = 'Purchase_Tickets';
$event= Nitwit::find($id);
if($event) {
$tiers = Tier::where('nitwit_id', $id)->get();

return View::make('purchaseselection', compact('title', '
    tiers',
    'event')));}
else{
return Redirect::to('/accountdashboard')->with('message', '
    Event
    is_over!');
}}
}

```

```

public function postpurchaseEventTickets($id)

```

```

{
$event = Nitwit::with('tiers')->find($id);
$inputs = Input::all();
$userid= Auth::user()->id;

```

```

$title = 'Confirm Purchase & Payment';

$rules = array(
'tickets []' => 'max:1000|numeric',
);

$messages = array(
'max' => 'The :attribute field must be a max of :max
characters.',
'numeric' => 'The :attribute must be a number.',
);

$validator = Validator::make($inputs, $rules, $messages);
if($validator->fails()){
return
Redirect::to('/event/creation')->withErrors($validator)->
withInput();
}
else
{
if ($event->date <= (date("Y-m-d\H:i:s"))){
return Redirect::to('accountdashboard')->with('message', 'Old
Event');
}
}

```

```

$tickets = Input::get('tickets');

foreach($tickets as $tierId=>$ticketCount) {
    foreach($event->tiers as $tier) {
        if ($tierId == $tier->id) {
            if ($ticketCount <= $tier->tickets_available) {
                $info[$tier->id] = array(
                    'tiername'=>$tier->name,
                    'tickets_wanted'=>$ticketCount,
                    'price'=>$tier->price,
                    'description'=>$tier->description
                );
            }
        }
        else {
            return Redirect::to('accountdashboard')->with('message', 'Sold
            out_of'. $tier->name . '_Tickets!');
        }
    }
}
}
}
}

```

```

$data = array(
  'users_id' => Auth::user()->id,
  'paid' => '0',
  'timestamp' => date("Y-m-d_H:i")
);

$unpaidpurchase = new Purchase($data);
$unpaidpurchase -> user_id = Auth::user()->id;
$unpaidpurchase ->save();

Session::flash('tierdatas', $info);

return
View::make('purchasepayment')->with('tierdatas', $info)->with(
  'title',
  'Ticket_Payment')->with('purchaseId', $unpaidpurchase->id);
}
}

```

```
public function markTicketUsed($id) {  
    $ticket = Ticket::findOrFail($id);  
    $ticket->used = 1;  
    $ticket->save();  
  
    return Redirect::back();  
}  
  
public function markTicketNotUsed($id) {  
    $ticket = Ticket::findOrFail($id);  
    $ticket->used = 0;  
    $ticket->save();  
  
    return Redirect::back();  
}  
  
}
```



## APPENDIX E

### MYEVENT CONTROLLER CLASS

```
<?php
```

```
class MyEventController extends BaseController {

    public function showmyevents()
    {
        $id = Auth::user()->id;
        $events = Nitwit::where('user_id', $id) ->get();
        return View::make('myevents', array(
            'title' => 'My_Events',
            'events' => $events,
        ));
    }

    public function showeventdetails($eventid)
    {
        $nitwit= Nitwit::findOrFail($eventid);

        return View::make('eventpage')
            ->with('title', 'Edit_Event')
            ->with('locations', Location::find($nitwit->location_id))
    }
}
```

```
->with ('event', $nitwit);
```

```
}
```

```
public function editmyeventdetails($eventid)
```

```
{
```

```
$event =
```

```
Nitwit::with('location')->where('id',$eventid)->first();
```

```
if ($event) {
```

```
if ($event->user_id == Auth::user()->id | Auth::user()->admin
```

```
==
```

```
1) {
```

```
return
```

```
View::make('editmyeventdetails',compact('event'))->with('
```

```
title','Edit
```

```
Event');
```

```
}
```

```
else {
```

```
return Redirect::to('/accountdashboard/')->with('message',
```

```
"That's not your event!");
```

```
}
```

```
}
```

```

else {
return Redirect::to('/accountdashboard/')->with('message', "
    No
Such_Event!");
}
}

```

```

public function posteditmyeventdetails($eventid)
{

```

```

$nitwit = Nitwit::find($eventid);
if ($nitwit) {

$inputs = Input::all();
$rules = array(
'eventname' => 'required|between:6,60',
'eventtype1' => 'required|max:10',
'eventtype2' => 'required|max:10',
'eventtype3' => 'required|max:10',
'datetime' => 'required|date',
'locationname' => 'required|max:45',
'streetaddress' => 'required|between:4,45',

```

```

'city' => 'required|between:2,45',
'state' => 'required|size:2',
'zipcode' => 'required|size:5',
'eventsummary' => 'required|max:160',
'eventdescription' => 'required',
'images[]' => 'image|max:300',
'numberoftickets' => 'max:1000|integer',
'priceofticket' => 'max:999.99|integer',
);

$messages = array(
'image' => 'The file uploaded must be an image',
'required' => 'The :attribute field is required.',
'between' => 'The :attribute must be between :min and :max
characters.',
'max' => 'The :attribute field must be a max of :max
characters.',
'min' => 'The :attribute field must be a min of :min
characters.',
'unique' => 'The :attribute must be unique.',
'integer' => 'The :attribute must be a number.',
'date' => 'The date is not in a valid format',
);

```

```

$validator = Validator::make($inputs, $rules, $messages);
if($validator->fails()){
return Redirect::back()->withErrors($validator)->withInput()
    ;}
else
{
$locationdata = array(
'name' =>Input::get('locationname'),
'address'=>Input::get('streetaddress'),
'city'=>Input::get('city'),
'state'=>Input::get('state'),
'zip_code'=>Input::get('zipcode'),
);

$eventdata = array(
'name' =>Input::get('eventname'),
'type1'=>Input::get('eventtype1'),
'type2'=>Input::get('eventtype2'),
'type3'=>Input::get('eventtype3'),
'date'=>Input::get('datetime'),
'summary'=>Input::get('eventsummary'),
'description'=>Input::get('eventdescription'),

```

```
);
```

```
foreach($nitwit->tiers as $index=>$tier) {  
    $index += 1;  
    $tierdata = array(  
        "name"=>Input::get("tiername{$index}"),  
        "description"=>Input::get("tierdescription{$index}"),  
        "price"=>Input::get("priceofticket{$index}"),  
        "tickets_available"=>Input::get("numberoftickets{$index}")  
    );  
    $tier->update($tierdata);  
}
```

```
$destinationPath.= "uploads/" . $nitwit->user_id;  
$location_id = Nitwit::find($eventid)->location_id;  
$locations = Location::find($location_id);  
$nitwit->update($eventdata);  
$locations->update($locationdata);
```

```
$bool = false;  
$images = Input::file('images');  
foreach($images as $image) {
```

```

if ($image) { $bool = true; }
}

if ($bool) {

foreach($images as $key=>$image) {
if ($image) {
$name = date('YndHis');
$name.=$image->getClientOriginalName();
$uploadSuccess= $image->move($destinationPath,$name);
if($uploadSuccess) {
$imageData = array(
'filename'=>$name,
'nitwit_id'=>$nitwit->id
);
$image = Image::create($imageData);
}
else {
echo "died";
}

}
}
}

```



```

}

return
Redirect::to("/myevents/$eventid/edit")->with('message', '
    Event
Updated_Sucessfully!');
}}
else{
return $eventid;
}

}

public function eventdeletion($id)
{
$view = View::make('eventdeletion');
$view -> title = "Event_Deletion";
$view -> eventid = $id;
return $view;
}

public function posteventdeletion($id)

```

```

{
$event = Nitwit::find($id);
$event->tiers()->delete();
$event->delete();
return Redirect::to("/myevents/");
}

```

```

public function deleteImage($id) {
$image = Image::findOrFail($id);

$image->delete();
return Redirect::back()->with('message', 'Image Deleted
Successfully!');
}

```

```

public function updateStatus() {
$codes = Input::get('used_tickets');

foreach($codes as $code) {
$ticket = Ticket::where('code', $code)->first();
$ticket->used = 1;
$ticket->save();
}
}

```

```
return Redirect::back()->with('message', 'Tickets_Marked!');  
}  
  
}
```

APPENDIX F  
MYPURCHASE CONTROLLER CLASS

```
<?php
```

```
class MyPurchaseController extends BaseController {

    public function showmypurchases()
    {
        $a =
        Purchase::with('tickets.tier.nitwit')->where('user_id',Auth::
            user()->id)->get();
        if ($a){
            foreach($a as $purchase) {
                foreach($purchase->tickets as $ticket) {
                    $tickets[] = $ticket;
                }
            }

            $view = View::make('mypurchases',compact('tickets'));
            $view -> title ="My Purchases";
            return $view;}
        else {
```

```
return Redirect::to('accountdashboard')->with('message', 'You
Have.No.Tickets.');
```

```
}
```

```
}
```

```
}
```

APPENDIX G  
PUBLIC CONTROLLER CLASS

```
<?php
```

```
class PublicController extends BaseController {
```

```
public function showfrontpage()
```

```
{
```

```
$today = date("Y-m-d_H:i:s");
```

```
$events = Nitwit::where('date', '>=', $today)->take(10)->get();
```

```
return View::make('frontpage')
```

```
->with ('title', 'Welcome_to_Ticketstand')
```

```
->with ('events', $events);
```

```
}
```

```
public function showsearch()
```

```
{
```

```
$view = View::make('search');
```

```
$view -> title = "Search_Events";
```



```

return $view;
}

public function postshowsearch()
{

$searchTerms = Input::get('q');
$searchtype = Input::get('searchtype');
$perPage = 5;

if ($searchtype == 'byname'){
$results['Name'] =
Nitwit::where('name', 'like', "%{$searchTerms}%")->paginate(
    $perPage);
}

if($searchtype == 'bytype'){
$results['Type'] =
Nitwit::where('type1', 'like', "%{$searchTerms}%")->orWhere('
    type2',
'like',

```

```

"%{$searchTerms}%" )->orWhere( 'type3 ', 'like ', "%{$searchTerms}%"
    )->paginate( $perPage);
}

if($searchtype == 'bydescription'){
$results[ 'Description' ] =
Nitwit::where( 'description ', 'like ', "%{$searchTerms}%" )->
    paginate( $perPage);
}

if($searchtype == 'byuser'){
$users = User::where( 'username ', 'like ',
"%{$searchTerms}%" )->get();
$user_id = array();
foreach($users as $user) {
$user_id [] = $user->id;
}
if (count($user_id)) {
$results[ 'User' ] =
Nitwit::whereIn( 'user_id ', $user_id)->paginate( $perPage);
}
}
}

```

```

if($searchtype == 'bycity'){
$locations = Location::where('city', 'like',
"%{$searchTerms}%" )->get();
$locationIds = array();
foreach($locations as $location) {
$locationIds [] = $location->id;
}
if (count($locationIds)) {
$results ['City'] =
Nitwit::whereIn('location_id', $locationIds)->paginate(
    $perPage);
}
}

```

```

if($searchtype == 'bystate'){
$locationIds = array();
$locations = Location::where('state', 'like',
"%{$searchTerms}%" )->get();
foreach($locations as $location) {
$locationIds [] = $location->id;
}
}

```

```

if (count($locationIds)) {
    $results['State'] =
    Nitwit::whereIn('location_id', $locationIds)->paginate(
        $perPage);
}
}

```

```

if ($searchtype == 'byzipcode'){
    $locations = Location::where('zip_code', 'like',
"%{$searchTerms}%")->get();
    $locationIds = array();
    foreach($locations as $location) {
        $locationIds[] = $location->id;
    }
    if (count($locationIds)) {
        $results['Zipcode'] =
        Nitwit::whereIn('location_id', $locationIds)->paginate(
            $perPage);
    }
}
}

```

```

$title = "Search_Events";

```

```
$view = View::make('search',compact('results','title'));  
return $view;  
}  
  
}
```

APPENDIX H  
PURCHASE CONTROLLER CLASS

```
<?php
```

```
class PurchaseController extends BaseController {

public function showpurchaseselection($id)
{
$today = date("Y-m-d_H:i:s");
dd(Nitwit::find($id)->date);
if (Nitwit::find($id)->date < $today)
{
return Redirect::to('/accountdashboard')->with('message', '
    Event
is over!');
} else {
$view = View::make('purchaseselection');
$view -> title = "Purchase_Selection";
return $view;}
}

public function showpurchasepayment()
```

```

{
$today = date("Y-m-d_H:i:s");
dd(Nitwit::find($id)->date);
if (Nitwit::find($id)->date < $today)
{
return Redirect::to('/accountdashboard')->with('message', '
    Event
is_Lover!');
} else {
$view = View::make('purchasepayment');
$view -> title = "Purchase_Payment";
return $view;
}}

```

```

public function postpurchaseprocessed($purchaseId)
{
if (!Session::has('tierdatas')) {
return
Redirect::to('accountdashboard')->with('message', 'Purchase
failed!');
}
}

```



```

$purchase = Purchase::findOrFail($purchaseId);

$purchase->paid = 1;

$purchase->save();

foreach(Session::get('tierdatas') as $tierId=>$tier) {

for ($a = 1; $a <= $tier['tickets_wanted']; $a++) {

$ticket = array(
'code'=>Hash::make(date('YmdHisu')),
'tier_id'=>$tierId
);

$ticket = Ticket::create($ticket);

$transaction = array(
'ticket_id'=>$ticket->id,
'purchase_id'=>$purchase->id
);

$transaction = Transaction::create($transaction);
}
}

```

```

$tierObject = Tier::find($tierId);
echo "Before:_" . $tierObject->tickets_available . "<BR>";
echo "Subtracting:_" . $tier['tickets_wanted'] . "<BR>";
$tierObject->tickets_available -= $tier['tickets_wanted'];
echo "After:_" . $tierObject->tickets_available . "<BR>";
$tierObject -> save();
}

return Redirect::to('mypurchases')->with('message', 'Tickets
Purchased!');
}
}

```

APPENDIX I  
MODEL OF TICKET CLASS

```
<?php
```

```
// Model: 'Ticket' - Database Table: 'tickets'
```

```
Class Ticket extends Eloquent
```

```
{
```

```
public $timestamps = false;
```

```
protected $table='tickets';
```

```
protected $guarded = array();
```

```
public function transactions()
```

```
{
```

```
return $this->belongsToMany('Transaction');
```

```
}
```

```
public function tier()
```

```
{
```

```
return $this->belongsTo('Tier');
```

```
}
```

```
public function purchases() {
```

```
return $this->belongsToMany('Purchase', 'transactions',
```

```
    'purchase_id', 'ticket_id');
```

```
}
```

APPENDIX J  
MODEL OF USER CLASS

```
<?php
```

```
// Model: 'User' - Database Table: 'users'
```

```
use Illuminate\Auth\UserInterface;
```

```
use Illuminate\Auth\Reminders\RemindableInterface;
```

```
Class User extends Eloquent implements UserInterface ,
```

```
    RemindableInterface {
```

```
protected $fillable = array('username', 'password', '
```

```
    firstname', 'lastname', 'email', 'website', 'admin', '
```

```
    locked', 'zipcode');
```

```
public $timestamps = false;
```

```
protected $table = 'users';
```

```
protected $hidden = array('password');
```

```
public function getAuthIdentifier()  
{  
    return $this->getKey();  
}
```

```
public function getAuthPassword()  
{  
    return $this->password;  
}
```

```
public function getReminderEmail()  
{  
    return $this->email;  
}
```

```
public function nitwits()  
{  
    return $this->belongsToMany('Nitwits');  
}
```

```
public function purchases()  
{  
    return $this->hasMany('Purchase');  
}  
}
```



APPENDIX K  
MODEL OF TIER CLASS

```

<?php

// Model: 'Tier' - Database Table: 'tiers'

Class Tier extends Eloquent
{

protected $table='tiers';
protected $fillable = array('name', 'description', '
    nitwits_id','tickets_available', 'price');
public $timestamps = false;

public function tickets()
{
return $this->hasMany('Ticket');
}

public function nitwit()
{
return $this->belongsTo('Nitwit');
}
}

```

```
public function getDropdownAvailableTickets()  
{  
for($a=0; $a<=$this->tickets_available; $a++)  
{  
  
$tickets []=$a;  
  
}  
return $tickets;  
  
}  
}
```

APPENDIX L  
FRONT PAGE VIEW FILE

```
@extends('layouts.default')
```

```
@section('content')
```

```
<div class = "silver" id="container">
```

```
Welcome to Nikolay Figurin's M.S. project. This service,
    currently called ticketstand, is a ticket purchasing and
    selling platform.
```

```
This site can be hosted not only on a PaaS, as initially
    expected, but also on most other servers with Very little
    change. The service is easily expandable and well-
    organized with an MVC pattern. Some security features are
    absent that would come standard with proper hosting.
```

```
<br><b>The site runs on Laravel 4 and MySQL (and a ton of
    other tools).</b>
```

```
<br>
```

```
<div id=" threecards">
```

```
<div class=" silver" id=" frontpage-point">
```

```

```

```
Use ticketstand to purchase any event tickets on sale
```

```
directly from other users. Currently, PayPal is used as it
's the fastest & cheapest for development/testing.
```

```
</div>
```

```
<div class = "silver" id = "frontpage-point">
```

```

```

```
Use ticketstand to sell any event tickets you want.
```

```
    Alternatively , you may create free events and use the
```

```
    service to host your event info and tally up guests.
```

```
</div>
```

```
<div class = "silver" id = "frontpage-point">
```

```

```

```
This service allows you to manage all of your sales & events
```

```
    through the dashboard once you log in- see tickets sold ,
```

```
    who they're sold to , and confirm attendance.
```

```
</div>
```

```
</div>
```

```
<br><div id = "disccall">
```

```
<div id = "disclamer">
```

```
Disclaimer : This is a development server , do not use this
```

```
    outside of testing.</div>
```

```
@if (Auth:: guest ( ) )
```

```

<div class="green" id="gotobutton"><a href="/registration
">
Register Today </div></a>
@else
<div class="green" id="gotobutton" style="color:#fff">
You're logged in!<br>
Thanks for your support!</div>
@endif

</div></div>

<br>

<div id="searchfield">
{{_Form::open(array(
'method'=>'GET',
'route'=>'post.show.search'))}}
<fieldset class="blue">
<legend><h1>Search Events</h1></legend><!--h1 tag may be
an issue-->
<div id="searchfieldandbutton"><input type="Text"
placeholder="Search Events!" name="q">
<input type="Submit" value="Submit"></div><br>

{{_Form::label('searchtype', 'Search Names')}}

```

```

{{_Form::radio('searchtype', '_byname', _true)}}|
{{_Form::label('searchtype', '_Search Types');}}
{{_Form::radio('searchtype', '_bytype')}}|
{{_Form::label('searchtype', '_Search Description');}}
{{_Form::radio('searchtype', '_bydescription')}}|
{{_Form::label('searchtype', '_Search Users');}}
{{_Form::radio('searchtype', '_byuser')}}<br>
{{_Form::label('searchtype', '_Search Cities');}}
{{_Form::radio('searchtype', '_bycity')}}|
{{_Form::label('searchtype', '_Search States');}}
{{_Form::radio('searchtype', '_bystate')}}|
{{_Form::label('searchtype', '_Search Zipcodes');}}
{{_Form::radio('searchtype', '_byzipcode')}}
{{_Form::close()_}}
</fieldset>
</div>

```

```
<div class="_silver" id="_searchresults">
```

```
<h1>_Upcoming_Events</h1>
```

```
@foreach_($events_as_$event)
```

```
<div class="_searchresult">
```



```

<div id="eventinfowrap">
<div id="eventname">{{ $event->name }}</div><br>
<div id="gotopill" class="blue"><a href="/search/results
?q={{ $event->type1 }}&searchtype=bytype">{{ $event->type1 }}
</a></div>
<div id="gotopill" class="blue"><a href="/search/results
?q={{ $event->type2 }}&searchtype=bytype">{{ $event->type2
}}</a></div>
<div id="gotopill" class="blue"><a href="/search/results
?q={{ $event->type3 }}&searchtype=bytype">{{ $event->type3
}}</a></div>
<div id="eventtag">{{ $event->date }}</div>
<div id="eventtag">{{ $event->location->city }},{{ $event->
location->state }}</div>
<br><div id="eventsummary">{{ $event->summary }}</div>
</div><div id="searchresultsgotobutton">
<div id="ticketsleft"><h2><?php
$ticketleft = 0;
foreach ($event->tiers as $index => $tier):
$ticketleft = $ticketleft + ($tier->tickets_available);
endforeach;
echo $ticketleft ?> Tickets Left </h2></div><br>

```

```
<div class="blue" id="gotobutton"><a href="/events/{{$event->id}}">See event info </a></div>
```

```
</div></div>
```

```
@endforeach
```

```
</div>
```

```
</div>
```

```
@endsection
```

APPENDIX M

VIEW PORTION OF THE EDITMYEVENTDETAILS FILE

```
@extends('layouts.default')
```

```
@section('content')
```

```
<div class = "silver" id="properform" style="max-width:900px  
    ;">
```

```
<div id = "loginfield" style = "margin-left:30px;">
```

```
@if(Session::has('message'))
```

```
<div class = "green" id = "message" style="font-size:62px;"  
    >{{Session::get('message')}}<br></div>
```

```
@endif
```

```
<div class = "green" id = "gotobutton" style="float:right;_  
    margin-top:20px;padding:10px;">
```

```
<a href="/myevents">Go to My Events </a>
```

```
</div>
```

```
<div class = "green" id = "gotobutton" style="float:right;_  
    margin-top:20px;_padding:10px;">
```

```
<a href="/events/{{ $event->id }}">View Event Page </a>
```

```
</div>
```

```

<div id = "error">

@foreach ($errors->all() as $error)
{{ $error }}<br>
@endforeach

</div>
</div>

<legend><h1>Edit Event Details</h1></legend>

@if($event->images)
@foreach($event->images as $image)
<img src='{{ $_URL::to("uploads/{$event->user_id}/{$image->
filename}")_}}' style = "max-width:200px;_max-height:200px
;_display:inline-block;" />
{{ Form::open(
array(
'method'=>'DELETE',
'route'=>array('delete.event.image',$image->id)

```

```

)
) }}
{{ Form::submit('x') }}
{{ Form::close() }}
@endforeach
@endif

{{Form::open(array('url'=>"myevents/{Sevent->id}/edit", '
    method'=>'post', 'files'=>true))}}
<legend> Basics </legend>
<div id = "formcluster">
Event Name - Keep it short!<br>
{{Form::text('eventname', $event->name);}}
<br><br>
Event Types - Concert, Rock, Music, or anything Custom - For
    Searchability<br>
{{Form::text('eventtype1', $event->type1);}}
{{Form::text('eventtype2', $event->type2);}}
{{Form::text('eventtype3', $event->type3);}}
<br><br>Date & Time (<b>{{date("Y-m-d.H:i")}}</b><br>
{{Form::text('datetime', $event->date);}}
</div><br>

```

```

<legend> Location </legend>
<div id = "formcluster">
Location Name <br>
{{Form::text('locationname', $event->location->name);}}
<br><br>Street Address<br>
{{Form::text('streetaddress', $event->location->address);}}
<br><br>City<br>
{{Form::text('city', $event->location->city);}}
<br><br>State Abbreviation<br>
{{Form::text('state', $event->location->state);}}
<br><br>5-Digit Zipcode<br>
{{Form::text('zipcode', $event->location->zip_code);}}
</div><br><br><br>

```

```

<legend> Descriptions </legend>
<div id = "formcluster">
Event summary - This will be seen in the search and upcoming
views (160 Characters)<br>
{{Form::text('eventsummary', $event->summary);}}<br>
<br><br>

```

```

Event description – this will be seen on the event page<br>
{{Form::textarea('eventdescription', $event->description)
  ;}}</div><br><br><br>

```

```
<legend> Photo Upload </legend>
```

```
<div id = "formcluster">
```

```
<?php for($a = count($event->images); $a <= 2; $a++) { ?>
```

```
Upload Photo {{ $a+1 }}/3
```

```
{{Form::file('images[]')}}</div>
```

```
<br><br>
```

```
<?php } ?>
```

```
</div>
```

```
<br><br>
```

```
@foreach($event->tiers as $index=>$tier)
```

```
<legend>Tier {{{ $index+1 }}} Tickets</legend>
```

```
<div id = "formcluster">
```

```
Tier name<br>
```

```
{{ Form::text('tiername' . ($index+1), $tier->name) }}
```

```
<br><br>Tier description<br>
```



```

{{ Form::textarea('tierdescription' .($index+1), $tier->
description) }}
<br><br>Number of tickets<br>
{{ Form::text('numberoftickets' .($index+1), $tier->
tickets_available) }}
<br><br>Price of each ticket in USD<br>
$ {{ Form::text('priceofticket' .($index+1), $tier->price)
;}}</div><br><br><br><br>
@endforeach

```

```

{{Form::submit('Update_Event');}}
{{Form::token()}}
{{Form::close()}}

```

```
<br><br>
```

\*Required. Be sure to check over all information as editing and deletion is not allowed after purchases are made.

```
<br>
```

```

<div class = "red" id = "gotobutton" style= "padding:10px;_
height:_28px;"><a href="/myevents/{{ $event->id }}/delete"
target="_blank"> Remove Event </a> </div>

```

<br><br>

</div>

@endsection

## REFERENCES

- [1] G. E. Krasner and S. T. Pope, "A cookbook for using the model-view controller user interface paradigm in Smalltalk-80", *Journal of Object-Oriented Programming*, vol 1, (3), pp. 26-49, 1988.
- [2] *Top Reasons for Product Managers to Embed MySQL*, MySQL, [online] 2013, [http://www.mysql.com/why-mysql/topreasons\\_pm.html](http://www.mysql.com/why-mysql/topreasons_pm.html) (Accessed: 13 July 2013).
- [3] *MNL Services*, MNL, [online] 2010, <http://www.mnl-limited.com/services.htm> (Accessed 13 July 2013).
- [4] D. Rees, "Migrations", (Documentation), [online] 2013, <http://four.laravel.com/docs/migrations> (Accessed: 13 July 2013).
- [5] C. Guitierrez and W. Jeffrey, "Personal Identity Verification (PIV) of Federal Employees and Contractors", *Federal Information Processing Standards Publication*, Vol. 201 March 2006.
- [6] P. Mell and T. Grance, "Recommendations of the National Institute of Standards and Technology", *The NIST Definition of Cloud Computing*, Vol. 800 (145) September 2011.

- [7] W. Door, Y. Chang, H. Abu-Amara, and J. F. Sanford, *Transforming Enterprise Cloud Services*, New York: Springer, 2010.
- [8] L. Conway, "The Economics of Ticketmaster", (Planet Money The Economy Explained), [online] September 2009, [http://www.npr.org/blogs/money/2009/09/podcast\\_the\\_economics\\_of\\_ticke.html](http://www.npr.org/blogs/money/2009/09/podcast_the_economics_of_ticke.html) (Accessed: 13 July 2013).
- [9] M. Hasan, B. Sugla, and R. Viswanathan. A conceptual framework for network management event correlation and filtering systems. In Proc. IEEE/IFIP 6th Int. Symposium on Integrated Network Management, pages 233-246, 1999.
- [10] G.C. Ramsborg, B. Miller, D. Breiter, B. J. Reed and A. Rushing (eds), *Professional meeting management: Comprehensive strategies for meetings, conventions and events*, 5th ed. Dubuque, Iowa: Kendall/Hunt Publishing, 2008.
- [11] S. Radicati and T. Buckley, "Email Market 2012-2016 Executive Summary", [online] July 2012, <http://www.radicati.com/wp/wp-content/uploads/2012/10/Email-Market-2012-2016-Executive-Summary.pdf>
- [12] D. Rees, "Eloquent ORM", (Code Happy), [online] 2013, <http://codehappy.daylerees.com/eloquent-orm> (Accessed: 13 July 2013).
- [13] J. Lewis, "Forms in Laravel", (Laravel: Using Forms and the Validator), [online] April 2012, <http://jasonlewis.me/article/laravel-using-forms-and-the-validator> (Accessed: 13 July 2013).
- [14] D. Rees, "Blade Templates", (Code Happy), [online] 2013, <http://codehappy.daylerees.com/blade-templates> (Accessed 13 July 2013).

- [15] T. E. Potok, M. Vouk, and A. Rindos, "Productivity Analysis of Object- Oriented Software Developed in a Commercial Environment", *Software - Practice and Experience*, vol. 29, (10) pp. 833-847, 1999.
- [16] Changelog (Laravel 4), Laravel Community Wiki, [online] 2012-2013, [http://wiki.laravel.io/Changelog\\_%28Laravel\\_4%29](http://wiki.laravel.io/Changelog_%28Laravel_4%29) (Accessed 13 July 2013).
- [17] D. Rees, "Configuration", (Documentation), [online] June 2013, <http://four.laravel.com/docs/configuration> (Accessed 13 July 2013).
- [18] How MySQL Deals with Constraints, MySQL, [online] 2013, <https://dev.mysql.com/doc/refman/5.0/en/constraints.html> (Accessed 13 July 2013).
- [19] Composer Documentation, [online] 2012, <http://getcomposer.org/doc/> (Accessed 13 July 2013).
- [20] Database Seeding with Laravel, CD, [online] 2013, <http://laravelbook.com/laravel-database-seeding/> (Accessed 13 July 2013).
- [21] 6 Week Roadmap, Pagoda Box, [online] August 2012, <http://blog.pagodabox.com/six-week-roadmap/> (Accessed 13 July 2013).