qwikstubs

Design Specification

Prepared by The Constructors

Version 1.0

March 7, 2013

Alex Hamstra
Ben McCurdy
Brittany Berin
Jared Roesch
Kyle Jorgensen
Mentor: Colin Kelley

Table of Contents

1 High-Level Architecture Overview

QwikStubs is a real-time ticketing web service. The product is intended to make finding events and purchasing tickets effortless and fast, by leveraging the power of modern web technologies. The service is designed to link fans, promoters, and venues, so that we can provide each with an optimal experience. We looked out into the current space of ticket sale services and found them lacking in many ways. We want Qwikstubs to be about the fans, making it the most enjoyable process we can for them to purchase tickets to events.

We built our system in a few modular components, user interaction happens completely in the browser, where all views (composed of HTML, and CSS) are rendered, and dynamically updated and re-rendered by JavaScript. All the views are served by Rails, and all events in the browser are passed to, and sent from EventMachine, allowing async communication, and updates like live counters, and seat maps. All of our data is persisted to MongoDB, which is abstracted away completely by Rails.

1.1 Model-View-Controller Architecture

The three components for which this architecture is named are the model, the view, and the controller. The architecture's focus is on separating presentation of information from the representation, providing a layer of abstraction between them. This encourages software development practices such as code reuse and separation of concerns. As code is heavily factored into modules that each have their own responsibilities and behavior, they are then glued together with a set of common plumbing logic. The three components of the architecture are intended to generalize typical moving parts in a wide set of applications. This allows the general abstractions of model, view, and controller to encode the domain specific portions without needing to micromanage all the plumbing. This is usually accomplished through use of a framework which provides these general abstractions as well the connections between them.

In this pattern we divide the components like so:

- A *model* consists of data, and business logic.
- A *view* is any representation of the data encapsulated by models.
- A *controller* is the intermediate component that takes user input, consumes and produces models, and properly serves the user views.

The QwikStubs architecture centers around the MVC pattern. We use models to store all the site's data and logic(in the form of methods). Views are the visual representations of our data, and serve as the interface for the user. The controllers serve as mediators between the user and the site -- properly routing a user's actions in the views to the correct code to modify objects in the backend (database), and then update the views with new data.

1.2 Ruby on Rails_

The MVC framework we are using to build our project is Rails. It is a framework built on the Ruby ideal of DRY (Don't repeat yourself), and extended with the Rails' principle of convention over configuration, which is the idea that everything should "just work" by default. Even though these are the basic Rails conventions, one should still be able to configure and modify the framework to suit any need.

When a user visits a URL of a Rails application, that specific URL corresponds to a single "route", which then corresponds to a specific controller action. For instance, in QwikStubs, if you were to go to http://localhost:3000/**venue** then the "list" action of the venues_controller would be invoked. This would then gather information from the Venue model and send the information back to the browser through the "list" view for Venues. These components are shown in **Figure 1.**

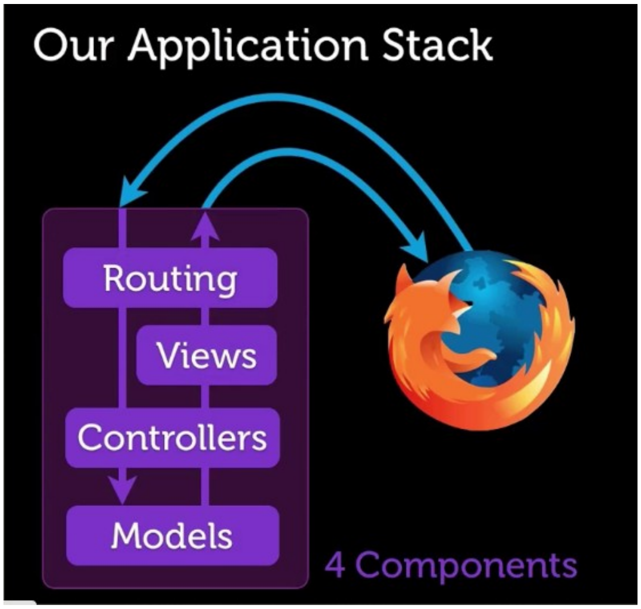


Figure 1. Rails application stack

1.3 EventMachine

QwikStubs will rely on the Ruby Gem EventMachine in order to implement our event handling in the backend, which is important for facilitating the real time updating of data for ticket sales. With a web service like this, one of the biggest issues is the scalability of the web service. In other words, our server must be able to handle an extreme amount of load in order to offer features such as realtime seating maps, and ticket counts. EventMachine is known in the Ruby community for its event-driven I/O, high scalability, and for reducing the cognitive overhead of threaded programming for the developer, allowing them to focus on the application logic instead of the plumbing logic. EventMachine's C++ core also means it provides extra performance that one can't find in Ruby.

Concurrency is baked in, for example a concurrent echo server is as easy as:

```
require 'eventmachine'

 module EchoServer
```

```
  def post_init
    puts "-- someone connected to the echo server!"
  end

  def receive_data data
    send_data ">>>you sent: #{data}"
    close_connection if data =~ /quit/i
  end

  def unbind
    puts "-- someone disconnected from the echo server!"
 end
end
```

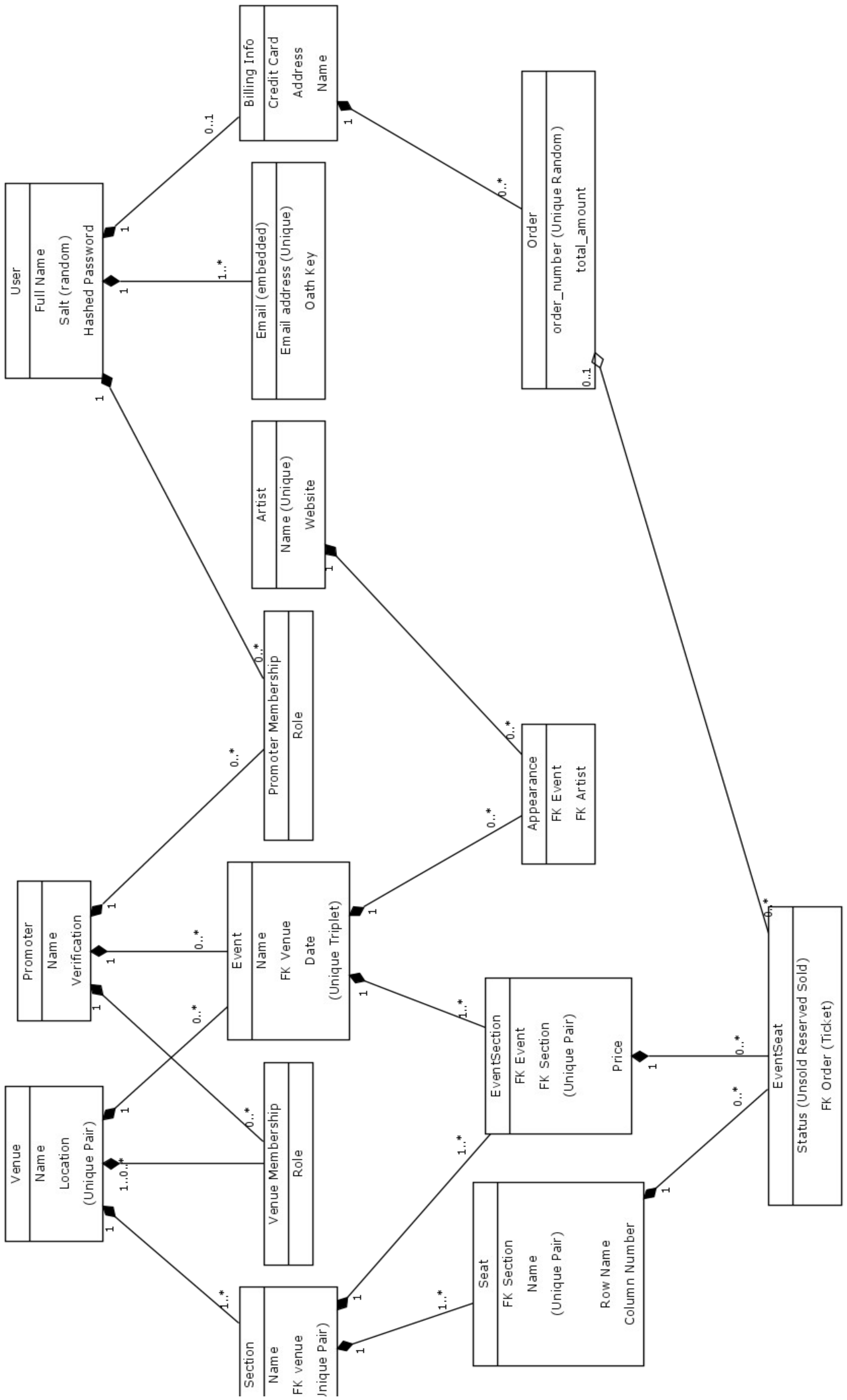
1.4 Browser

The browser will be our method for serving up the UI. We will support modern browsers to the best of our ability. UI in the browser is created from a combination of HTML, CSS, and JavaScript; the latter being our way of dynamically manipulating the design and layout of our pages. There are various challenges in getting each of those to play nicely in *all* browsers without a few hacks and custom solutions. We have sidestepped this development issue on the merit of using Bootstrap -- a CSS and Javascript framework that provides helpful defaults and built-in functionality. By using this framework, we can focus on general functionality instead of having to wrestle with the browsers. It also has the added benefit of bringing some great design to the project for free. We are also using another MVC framework, Backbone.js which enables us to mirror some our design from Rails in the browser to provide features like the auto-updating of listings, dynamic view rendering, super responsive UI elements, and more.

2 Models

The architecture for this project is comprised of 15 different models, as shown in **Figure 2.**   In the sections below, we will discuss each of these models in detail. Each model possesses a set of properties, which include both attributes (indicated with black text) and methods (indicated with green text) for interacting with these objects.

Figure 2 (below). QwikStubs UML Class Diagram


User
Full Name
Salt (random)
Hashed Password
Billing Info
Credit Card
Address
Name
Email (embedded)
Email address (Unique)
Oath Key
Artist
Name (Unique)
Website
Order
order_number (Unique Random)
total_amount
Promoter Membership
Role
Appearance
FK Event
FK Artist
Promoter
Name
Verification
Event
Name
FK Venue
Date
(Unique Triplet)
EventSection
FK Event
FK Section
(Unique Pair)
Price
EventSeat
Status (Unsold Reserved Sold)
FK Order (Ticket)
Venue
Name
Location
(Unique Pair)
Venue Membership
Role
Section
Name
FK venue
nique Pair)
Seat
FK Section
Name
(Unique Pair)
Row Name
Column Number
0..1
1
1..*
1
1
0..*
1
0..*
0..1
1
0..*
0..*
0..*
1
0..*
1
0..*
1
0..*
1
1..*
1
0..*
1
0..*
1
0..*
1
1..0..*
1
0..*
1
1..*
1
1..*
1
1..*
1

2.1 User

Properties:

- full_name
- emails
- salt
- password_digest
- first_name
- last_name
- password=(pass)
- is_auth?(pass)

The user model has a “many” relationship with emails, meaning that the Email model contains a foreign key, user_id, that references the User to which the email belongs. It was designed this way in order to support each user’s account having more than one email, allowing for all their online identities to be mapped to a single account. The User model is sparse, and only contains user metadata and password information.

There are many ways to store passwords, with some common ones being plaintext, encryption, and hashing. The first two are insecure in the presence of many advanced attacks. Allowing attackers to recover the original passwords and possibly exploit users using similar passwords on other services. This makes both of them a poor choice for a modern web application that might be susceptible to security threats. Instead, we are using hashed passwords. There are a few variations when hashing passwords, but by far the most secure option is to

  - use a salt
  - use a unique salt for each user

A salt is a set of random bits that the service combines with your password before hashing to protect against things like rainbow table attacks. By storing a unique salt for each user, we increase security and add even more overall entropy to the system.

We then store both the hashed version of the password+salt, and the salt itself to confirm the correctness of user passwords.

The User model has a few helper methods such as is_auth? which confirms the correctness of their password, full_name, and last_name to query their information, and password=, a password setter that regenerates the salt every time the password is changed.

2.2 Email

Properties:

- user
- email
- oauth_token

The email is a simple model that encapsulates the idea of an email or identity, and is always linked to a user. The email model simply has an email field, where the specific email address is stored. Since an Email belongs to a User, it has a user_id foreign key that contains the MongoDB ObjectId of the the User it belongs to. Finally it has an oauth_token, which will allow us to link email's with OAuth identity, allowing login capability via Facebook or Google.

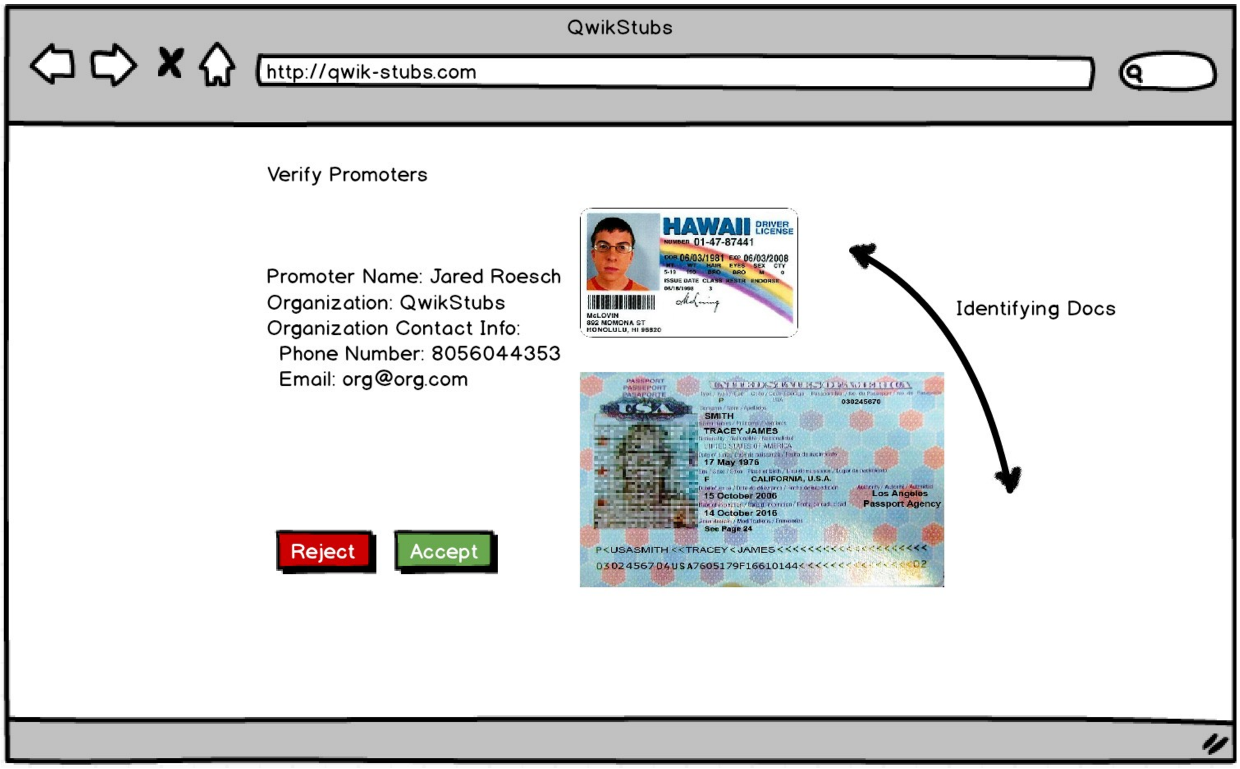
2.3 Promoter

Properties:

- name
- promoter_memberships
- events
- venue_memberships

The promoter model is used to organize the data of those who wish to sell tickets through the QwikStubs website. The promoter model "has many" events because they are capable of selling tickets to more than one event. A promoter also "has many" promoter_memberships and "has many" venue_memberships. On the QwikStubs website, a promoter is considered to be an organization. We modeled this after Github, where users can be part of an organization, and an organization can have its own repositories. Thus in QwikStubs, many users can potentially be verified as part of a promoter organization to create events and venues on the website for future ticket sales. A promoter organization, in turn, can be linked with several Venues -- an example being a sports organization wishing to sell tickets to games that take place in more than one arena.

To become a promoter, one must have a name that reflects the group which they are representing. Verification of a promoter will be performed manually using an inspection of a user's ID or credentials. We would like the approval process to be streamlined as possible with a simple UI for approving new promoters, and venues. A mockup of how a promoter will be verified on the QwikStubs website is shown below in **Figure 3.**



Verify Promoters

Promoter Name: Jared Roesch
Organization: QwikStubs
Organization Contact Info:
Phone Number: 8056044353
Email: org@org.com

Figure 3. Verification Mockup

2.4 Venue

Properties:

- name
- city
- state
- events
- sections
- add_event(event_details, layout)

The Venue model “has many" events and sections. A Venue can host countless events over time, and each Venue has a basic layout that is given by its sections.

Every venue must have a unique combination of a name, city and state in order to be added to the database, this will prevent duplication, or ambiguous references to Venues.

Venue will have some built in logic for adding an Event, through the add_event message. It will move the logic of setting up an event and all its constituent parts in one place. We will pass the necessary event details such as name, time, artists, and so on. Followed by the Venue layout we wish to select, and provide a way to save common layouts for Venues.

2.5 Event

Properties:

- name
- date
- venue
- promoter
- event_sections
- appearances

First, an Event *must* have both a name and date, with the pair being unique. Every Event model also belongs to a venue and a promoter, meaning that it has foreign keys: venue_id, and promoter_id contained in the model. A promoter is capable of creating events at venues with which they have a verified membership.

Each event can have its own seating chart comprised of different EventSections. For example, the Santa Barbara Bowl often has the pit as “General Admissions”, but also puts down seats sometimes with precise ticket numbers, so the EventSection model will handle this flexibility. Finally, each event can have appearances by more than one artist.

2.6 Section

Properties:

- name
- xpos
- ypos
- seats
- venue

A Section has a name, and a venue to which it belongs. The xpos and ypos attributes of Section are used to draw it correctly on the Canvas element in the browser. A Section also has many Seats associated with it.

2.7 Seat

Properties:

- name
- row
- column
- xpos
- ypos
- section

A Seat belongs to a specific Section, and it has a name which the user can see. The row and column attributes will be used to store the location, within a Section, of the Seat in the database. This will allow for promoters to create Seat names such as “15B”, which make sense to the fans in a venue, while still being able to keep track of the actual location of the Seat with its row and column. We will use this information to be able to write an algorithm for the auto seat selection. This way a fan can choose a “Best Seats” button with x number of seats and be able to find the best available seats for x number of people. The details of this are described in more detail in **Section 2.7.1**.

The xpos and ypos attributes are similar to those in the Section class; however, in this case, they represent the relative coordinates with respect to the Section. Our relative positioning system allows for flexibility in moving, and redrawing Sections because the layout of Seats within a Section is independent from the laying out of the Sections.

2.7.1 “Best Seat” Algorithm

The naive algorithm is obviously a linear scan looking for *n* number of seats within a section, but we feel that a smart, yet efficient algorithm is important for our real time requirements. We have a few potential algorithms, but have a few challenges along the way. On top of finding, or devising an algorithm, we must translate the algorithm to work on the database collections in place, as we can’t afford to construct objects in memory and traverse them on every request. On top of devising the algorithm the translation will be a very critical part to being performant and intelligent with seat allocation. There is one algorithm that we would like to present here as an initial way of solving the adjacent, open seat problem, and will do better than linear seat runs, as it can find seats that are adjacent but in two rows, and similar situations.

One approach that has been imagined is to begin by constructing a graph where each seat is a node, and adjacency is represented by an edge between nodes. Then take a single node that denotes being sold, and attach it to every node. This solves the case when a node becomes disconnected from the graph, by making all seats available without an adjacency relationship to another seat. When a ticket is sold remove all edges. When looking for a set of seats, we do a greedy search for the correct number of seats. There are many graph techniques for quickly finding clusters or paths, which can apply here. When we buy a set of seats we remove all edges from a node. This problem is generally NP-Complete to our knowledge, but the search space is small, and we are willing to take a greedy algorithm that produce a good solution with performance, over an optimal solution. Using a graph based algorithm is beneficial because we can represent them as adjacency lists in the database. This allows us to create an event collection when a sale begins, and modify them in place, inside of MongoDB. This is one of the major reasons why we are using MongoDB. It allows us to utilize indexing for speed of access, and avoid the cost of reconstructing the graph in memory on each request.

2.8 EventSection

Properties:

- price
- section
- event
- event_seats
- validate_event_and_section_unique

An EventSection is the class that connects an EventSeat to an Event. This class is necessary because it separates the logic of the Venue-Section-Seat hierarchy from the Event logic. It allows for the flexibility to have multiple Events at Venues, when each one of those Events might have different prices and/or sections for seats. Thus, it contains a foreign key to Event because it must represent a unique event, and it contains a foreign key to Section because that is where it gets the information regarding the layout of the Venue. It also has many EventSeats, which are to be sold at the price specified. Finally, we have a function to validate whether the foreign keys to Event and Section are a unique pair.

2.9 EventSeat

Properties:

- status
- event_section
- order
- seat

An EventSeat is basically a “ticket” in our class model. It contains the relevant information connecting an Order to an Event (via EventSection). The EventSeat model “belongs to” an EventSection and “has many” orders. Therefore, the EventSeat will contain a foreign key for the EventSection to which it belongs. An EventSeat will also contain “ticket” information by having one of three statuses -- unsold, reserved, or sold. These statuses will be used to determine and update the availability of seats within a venue in real-time. In the EventSeat model, we have created a module, Stat, that acts as an enum and assigns a numerical value to each of the three statuses. The module increases the level of abstraction when determining and establishing a seat’s status. The EventSeat is also connected to Seat via a “belongs to” relationship. This allows an EventSeat to access the relevant Seat information such as row and column number.

2.10 Artist

Properties:

- performance meta data
- members
- internal information related to events

In the Artist model, an artist “has many” appearances. An artist is capable of making an “appearance” at a single event and these appearances are not limited. Our architecture accommodates this. An artist must have a unique name in order to be stored into our database. An artist must also have a website for the benefit of QwikStubs’ users.

2.11 Appearance

Properties:

- event
- artist

The Appearance model “belongs to” an event and “belongs to” an artist. As a result, an Appearance will contain a foreign key from its specific event and another from its artist.

2.12 Billing Info

We have the basic data fields such as credit card #, billing address, and name of cardholder as placeholders in our model, but we do not want to actually handle credit card information in our system. To remove this burden, and to avoid the Payment Card Industry (PCI) requirements compliance, we are instead moving the billing process to a third party service like Stripe, or possibly something similar like PayPal, Amazon Payments, or Google Wallet.

2.13 Order

Properties:

- Order_Number
- event_seats
- Billing_Info

The Order model “belongs to” Billing Information and “has many” EventSeats. An order needs a user’s billing information in order to be fulfilled and an order can contain many tickets. Every order will be assigned a unique and random order number

2.14 Promoter Membership

Properties:

- role
- user
- promoter

The Promoter Membership model is utilized in the verification of promoters on the QwikStubs website. This model “belongs to” a User and “belongs to” a Promoter. A Promoter Membership can only be associated with one user, and likewise, one organization of promoter users.

Every Promoter Membership has a role that reflects a user’s position within a promoter organization. These roles can range from titles like “Administrator” to “General Manager,” and are determined by the group or venue that the promoters represent.

2.15 Venue Membership

Properties:

- role
- promoter

- venue

Similar to Promoter Membership is the Venue Membership model. A Venue Membership "belongs to" a promoter and "belongs to" a venue. This model serves to link an organization of promoter users to many venues. A Venue Membership also allows promoters to be given administrative roles within a Venue.

3 Controllers

From Ruby on Rails' guide to controllers:

> "A controller can thus be thought of as a middleman between models and views. It makes the model data available to the view so it can display that data to the user, and it saves or updates data from the user to the model."

In modern web development, there are many references to REST, which is an acronym for REpresentational State Transfer. What this means in a Rails context is that most application components are modeled as "resources" that can be created, read, updated, and deleted. These operations correspond to the CRUD operations of relational databases, as well as the four basic HTTP request methods: POST, GET, PUT, and DELETE.

As a Rails developer, we then have to consider our resources and determine which actions are necessary to put in their respective controllers. Not all of our models in the system need a controller, but the ones that represent objects that are stored in the database definitely need controllers. Below is a table that shows how a resource is typically mapped from routes to controller actions, using "users" as an example from QwikStubs

HTTP Verb	Path	action	used for
GET	/users	index	display a list of all users
GET	/users/new	new	return an HTML form for creating a new user
POST	/users	create	create a new photo
GET	/users/:id	show	display a specific user
GET	/users/:id/edit	edit	return an HTML form for editing a user
PUT	/users/:id	update	update a specific user
DELETE	/users/:id	destroy	delete a specific

			user

based on http://guides.rubyonrails.org/routing.html.

3.1 Application Controller

The Application Controller is the root object of the controller hierarchy. It is a Rails convention to use the application controller as a location for common logic shared between all controllers. In our application controller we will have common logic for querying the current user, checking if some is logged in, setting redirect urls, etc. All of which is functionality that is common to *all* controllers, not just one.

3.2 Users Controller

Actions:

- new => GET /users/new
- create => POST /users
- show => GET /users/:id
- edit => GET /users/:id/edit
- update => PUT /users/:id

The Users controller conforms to a subset Rail’s resource routes. That means each of its actions have an implied meaning. We have left out three of the standard actions (index, list, destroy), as they do not make sense in the context of the model. Following the traditional resource model: ‘new’ serves the UI to register a new user, ‘create’ takes the HTTP POST in order to generate a new User record in the DB, ‘show’ takes an :id and displays their user page, ‘edit’ serves the UI to update a user’s attributes, and ‘update’ takes the HTTP PUT request for updating a record.

3.3 Sessions Controller

The Sessions Controller is responsible for managing user sessions, or the concept of being logged in or logged out. This controller in particular is stripped down and only uses a small subset of Rails’ resource routes.

Actions:

- new => GET
- create => POST (a.k.a. “login”)
- destroy => DELETE (a.k.a. “logout”)

All three are part of Rail’s standard set of resource routes. The 'new' route renders the login view, where a User enters their credentials to begin a new session. The 'create' route is the route that processes the post request, generated by the submit action in new, and generates a new session, effectively logging the user in. Finally the 'destroy' route disassembles the User’s session, effectively logging the user out.

3.4 Venues Controller

Actions:

- new => GET /venues/new
- create => POST /venues
- list => GET /venues
- show => GET /venues/:id/show
- edit => GET /venues/:id/edit
- update => PUT /venues/:id
- destroy => DELETE /venues/:id

The Venue controller also conforms to the resource pattern. Meaning that the 'new' route maps to the UI for creating a new Venue, 'create' receives an HTTP POST, and generates a new Venue record in the database. 'List' shows a listing of all venues, show/:id takes a venue_id and shows a specific venue. 'Edit' serves the UI for updating a venue's information, and 'update' takes the HTTP PUT that updates the Venue record. Finally, we have destroy which allows for a promoter to remove a venue if they so choose.

When creating a Venue, the following sequence of actions occurs:

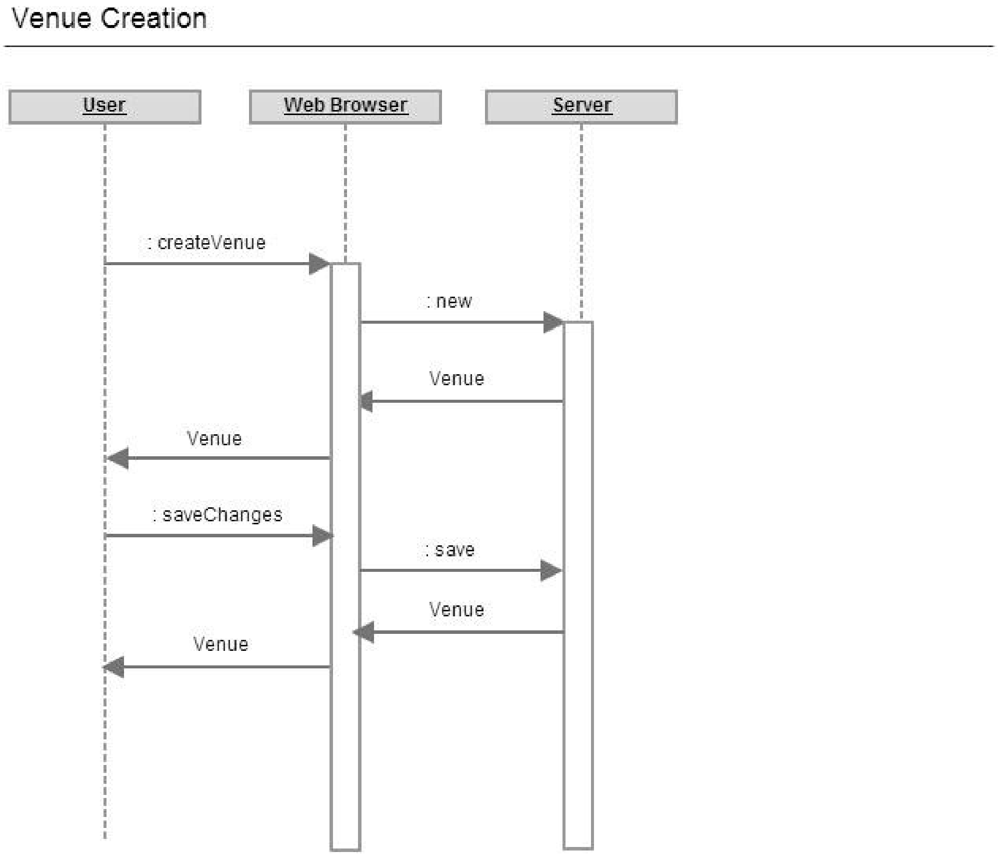


Figure 4. Venue Creation Sequence Diagram

When editing a venue to update its information, the following sequence occurs:

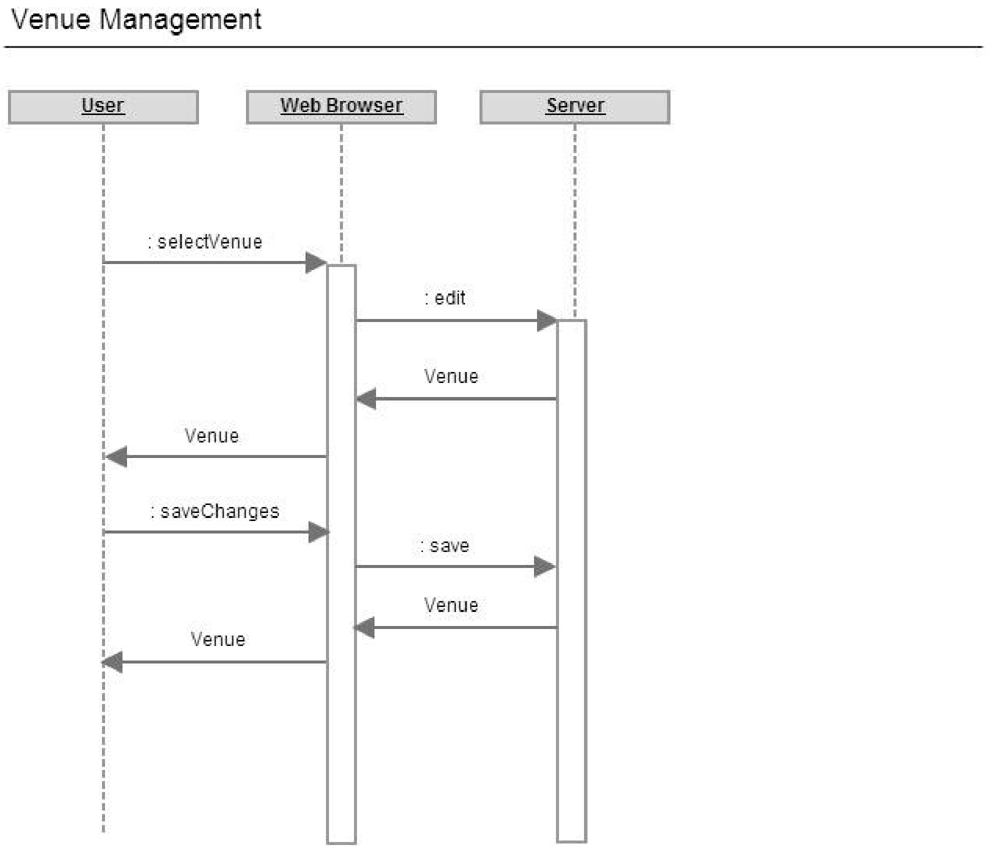


Figure 5. Venue Management Sequence Diagram

3.5 Events Controller

Actions:

- new => GET /events/new
- create => POST /events
- list => GET /events
- show => GET /events/:id/show
- edit => GET /events/:id/edit
- update => PUT /events/:id
- destroy => DELETE /events/:id

The Events controller has the same basic functionality as the Venues controller. ‘New’ leads to the UI for creating a new event. ‘Create’ receives the HTTP POST and adds the event to the database. ‘List’ shows all of the existing events in the database. ‘Show’ routes to a page with details for a specific event. ‘Edit’ leads to the UI for editing an event’s information. ‘Update’ receives an HTTP PUT and updates the record of the edited event in the database. And lastly, ‘destroy’ deletes a specific event from the database.

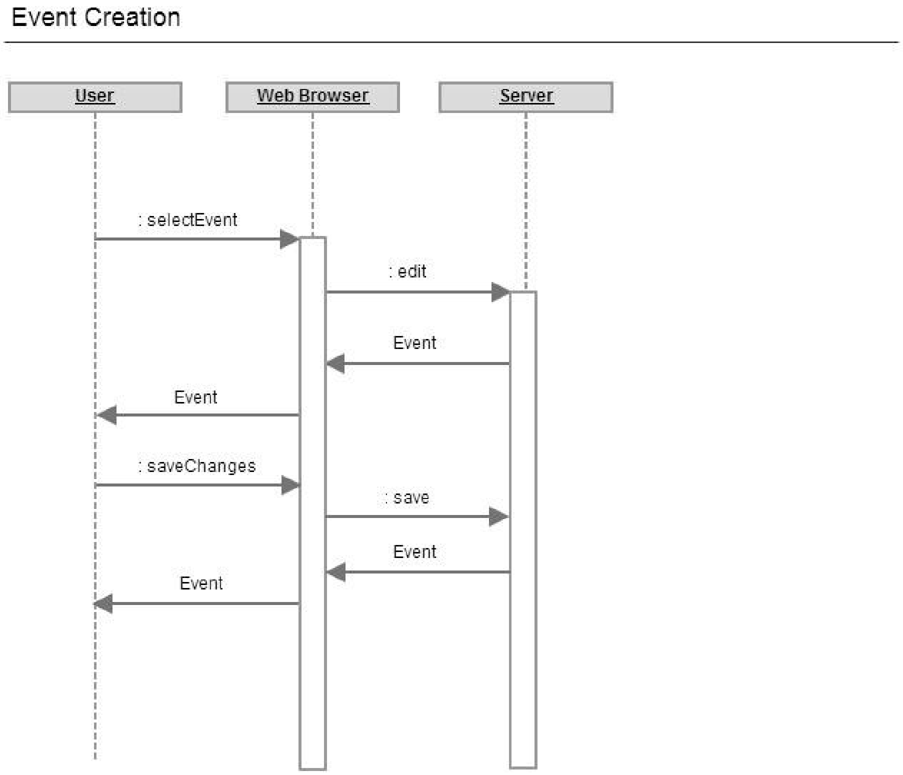


Figure 6. Event Creation Sequence Diagram

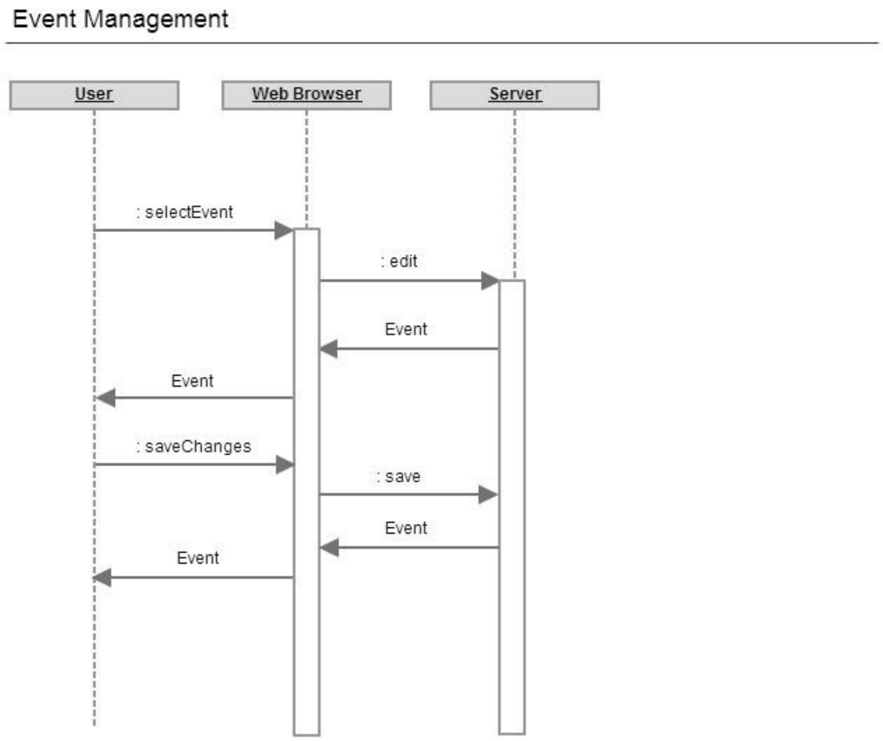


Figure 7. Event Management Sequence Diagram

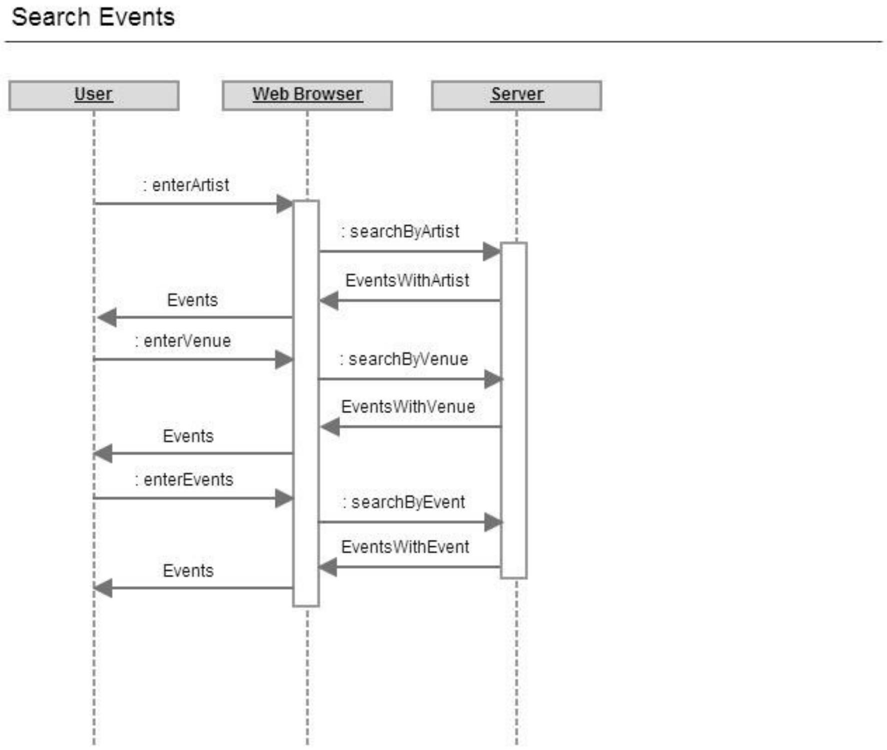


Figure 8. Event Search Sequence Diagram

3.6 Orders Controller

Actions:

- new => GET /orders/new
- create => POST /orders
- list => GET /orders
- show => GET /orders/:order_number/show
- destroy => DELETE /orders/:order_number

The orders controller is used for managing orders placed by QwikStubs' users. This controller allows a user to view, print, reserve, and purchase tickets to an event. 'New' routes to the page where a new order is created. 'Create' adds the order to the database. 'List' shows a list of all orders placed by a user. 'Destroy' deletes an order when appropriate.
Below is what occurs when a user wishes to view and print tickets.

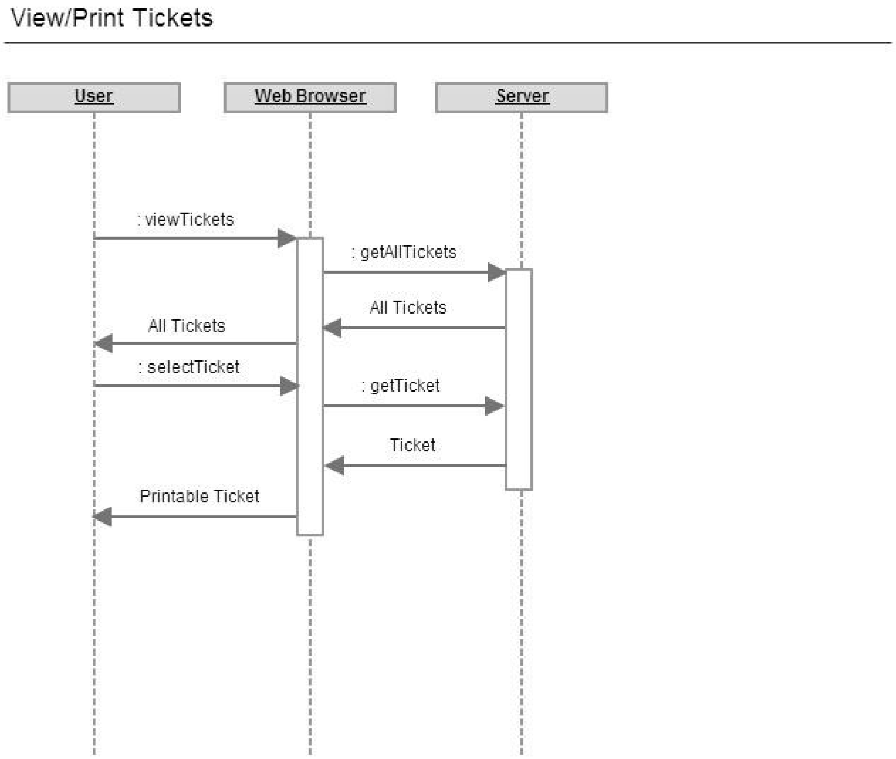


Figure 9. View/Print Tickets Sequence Diagram

Below is what occurs when a user wishes to reserve or purchase tickets.

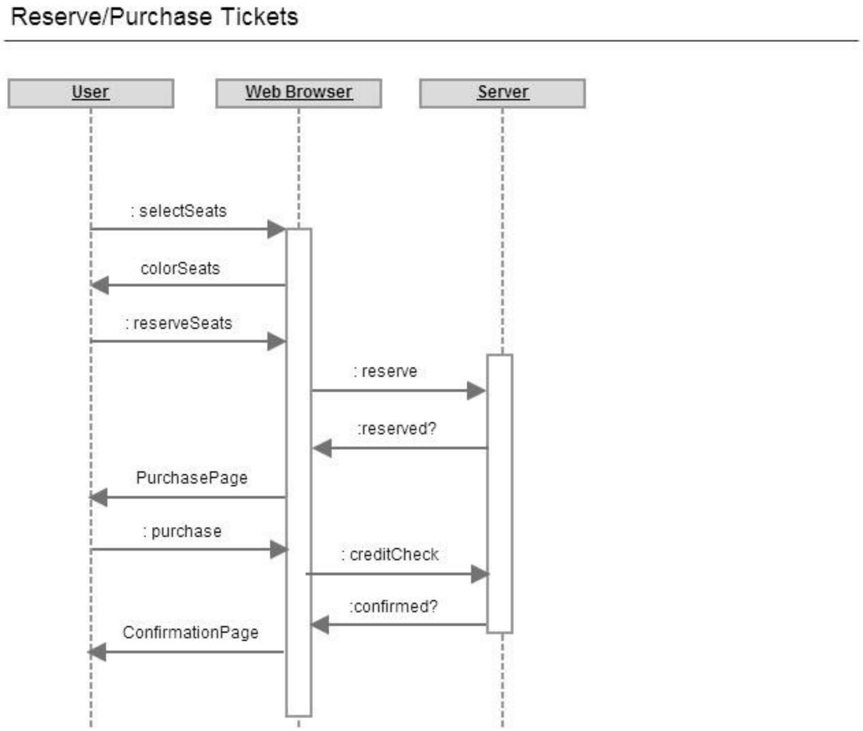


Figure 10. Reserve/Purchase Tickets Sequence Diagram

4 Views

Coffeescript is another tool we've decided to use in implementing the frontend. It provides some advantages over traditional JavaScript, as well as only being a thin compilation layer, meaning the compiled JavaScript sources are very readable. Its biggest advantage, is it prevents the use of "bad" JavaScript idioms, and features, by disallowing them completely. It is also syntactically very similar to Ruby, reducing the cognitive overhead of context switching between languages, by providing a more uniform syntax, and semantics. Coffeescript also is very readable further reducing developer effort when reading code, a must for maintainability. Finally Rails comes standard with Coffeescript, so it works seamlessly.

We also use Backbone.js in the browser which embeds another MVC framework in JavaScript, allowing us to build very dynamic pages. Backbone.js can be used to mirror the models we have in Rails inside the browser and make changes that propagate back to the database in the browser without having to reload the page. We can serve the intial data in our .erb templates so that the initial load is fast, but then can make updates, and pull more data in, depending on User interactions thanks to Backbone.js. Backbone.js also handles all the ajax calls and handles injecting templates into the page seamlessly, allowing for very smooth transitions and much faster load times. This will be really helpful when implementing the real time aspect because we can use Backbone.js to update the seating view with having the user know anything is happening.

To create uniformity throughout our pages we are using Twitter Bootstrap as are main tool for styling and layout configuration. This has many benefits, one of which being that we don’t have to write custom css for most of the styling. This saves tons of time and allows us to spend more time on features. It also have layout styling so that it automatically handles the conversions from desktop view to mobile view, giving us the flexibility to try out more ideas, rather than spending all our time making multiple views.

4.1 UI

4.1.1 QwikStubs Homepage

In **Figure 11** below, it showcases the QwikStubs homepage. The design aims for ease of access for all users -- both existing and new. Existing users are able to log in to the site directly from the homepage, while new users are able to register for the site from the same page. The site also features several events in the database that will be chosen by the user’s current location or could be sponsored events if they exist. This way we can decide what events a users see when they come to the homepage, by what events they are likely able to attend. The goal being a very enjoyable and efficient user experience.

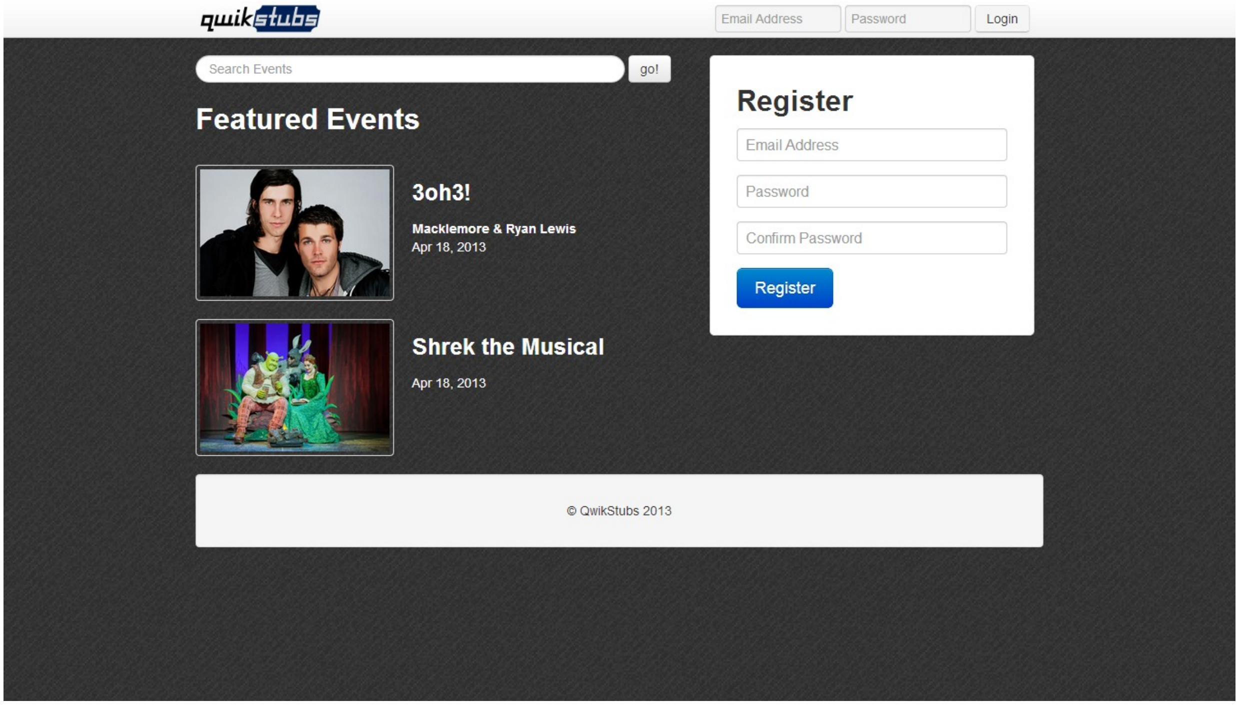
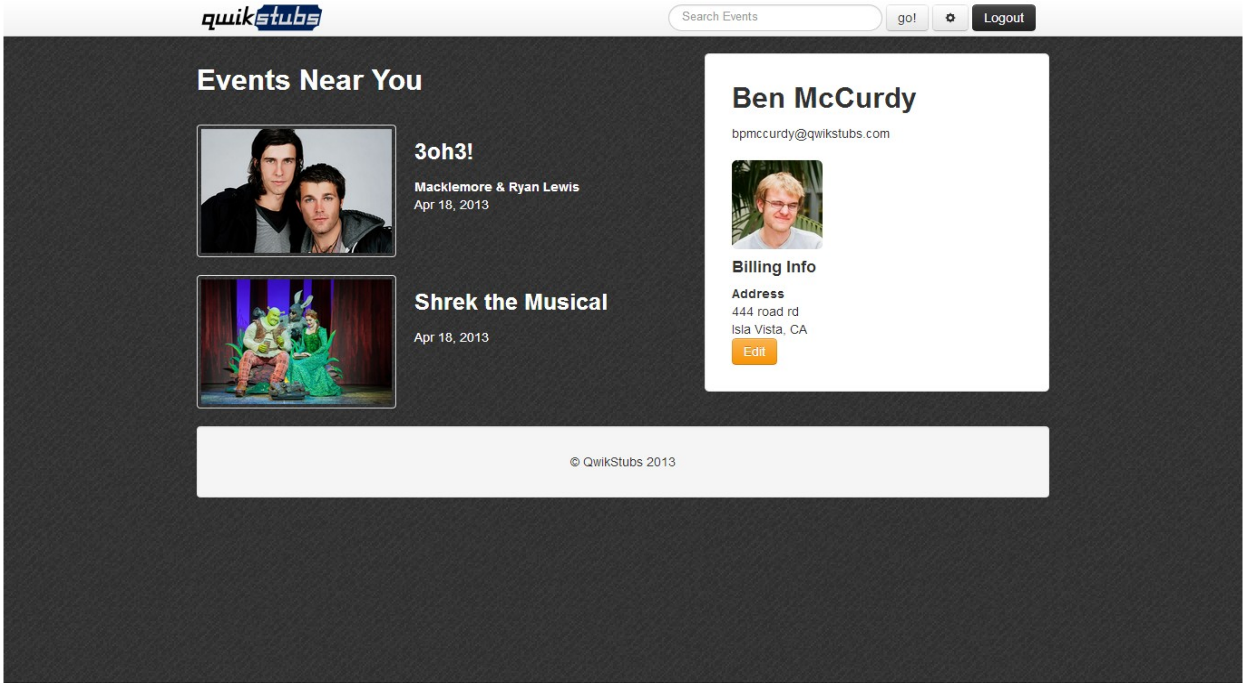


Figure 11. QwikStubs Homepage (Not Logged In)

In **Figure 12** below, we can see what the homepage would look like if you were logged in. It would contain all of the user info to the right and have events near you on the left. The search bar would also be moved to the top bar as to unclutter the rest of the page.



Events Near You

3oh3!

Macklemore & Ryan Lewis
Apr 18, 2013

Shrek the Musical

Apr 18, 2013

Ben McCurdy

bpmccurdy@qwikstubs.com

Billing Info

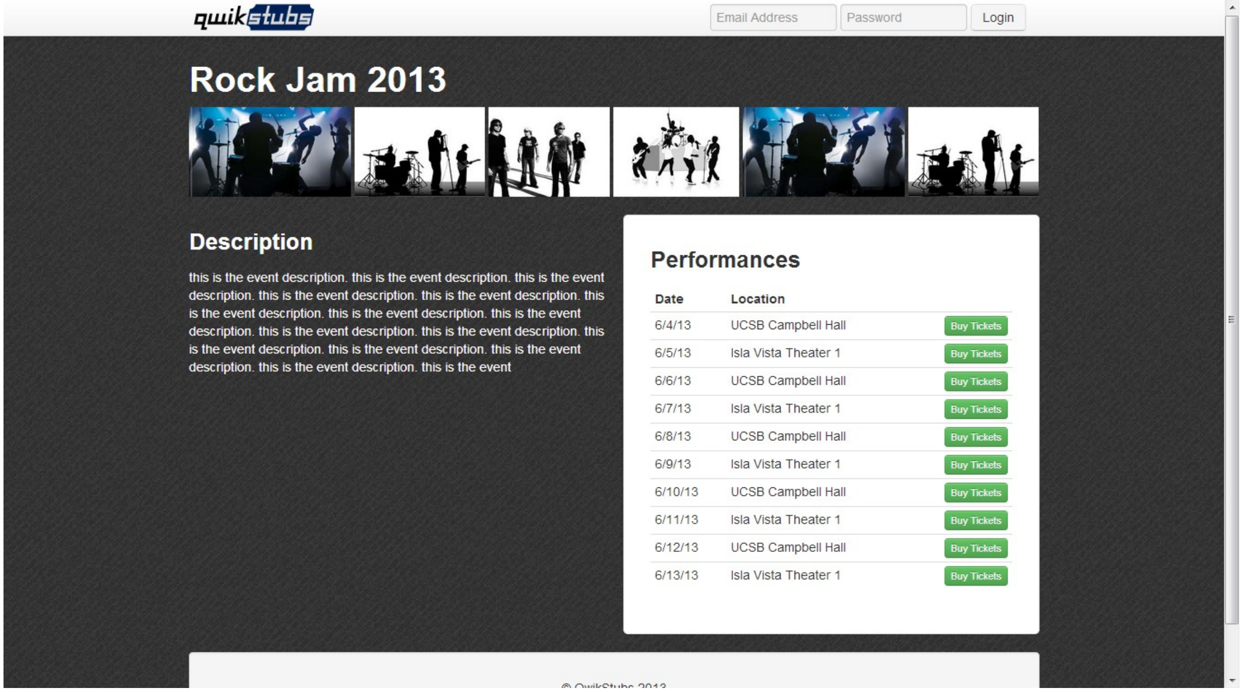
Address
444 road rd
Isla Vista, CA

© QwikStubs 2013

Figure 12. QwikStubs Homepage (Logged In)

4.1.2 QwikStubs Event Page

In **Figure 13** below, we see the QwikStubs Event page where users can view information about a specific event, as well as see when and where the event is going to take place. This way a user can decide which venue and what date they want to see their favorite band playing.



Rock Jam 2013

Description

this is the event description. this is the event description. this is the event description. this is the event description. this is the event description. this is the event description. this is the event description. this is the event description. this is the event description. this is the event description. this is the event description. this is the event description. this is the event description. this is the event description. this is the event description. this is the event

Performances

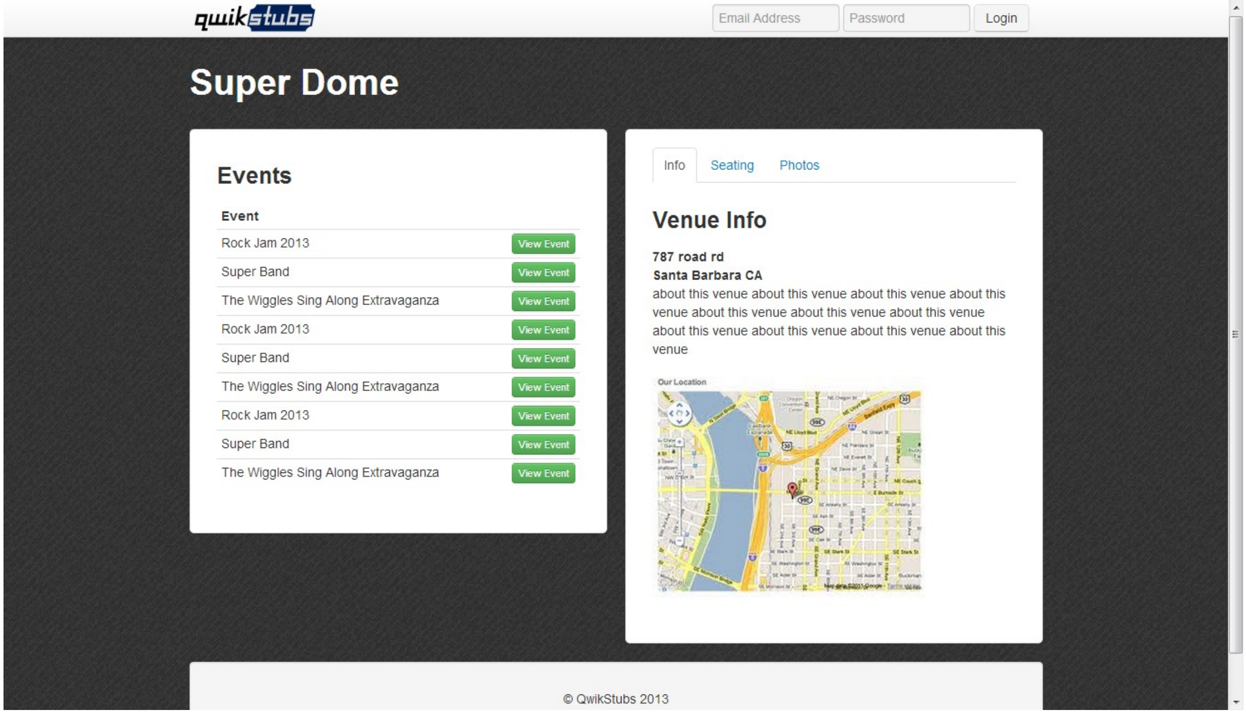
Date	Location	
6/4/13	UCSB Campbell Hall	Buy Tickets
6/5/13	Isla Vista Theater 1	Buy Tickets
6/6/13	UCSB Campbell Hall	Buy Tickets
6/7/13	Isla Vista Theater 1	Buy Tickets
6/8/13	UCSB Campbell Hall	Buy Tickets
6/9/13	Isla Vista Theater 1	Buy Tickets
6/10/13	UCSB Campbell Hall	Buy Tickets
6/11/13	Isla Vista Theater 1	Buy Tickets
6/12/13	UCSB Campbell Hall	Buy Tickets
6/13/13	Isla Vista Theater 1	Buy Tickets

© QwikStubs 2013

Figure 13. QwikStubs Event Page

4.1.3 QwikStubs Venue Page

In **Figure 14** below, we see the QwikStubs Venue page where users can view information about a specific venue, as well as see what events are taking place at that venue. From the venue page users can select an event they want to see that is at that venue and proceed to its event page. From this page, users will also be able to see the seating arrangement for the venue and a small google map and address so they know where the venue is.



Super Dome

Events

Event	
Rock Jam 2013	View Event
Super Band	View Event
The Wiggles Sing Along Extravaganza	View Event
Rock Jam 2013	View Event
Super Band	View Event
The Wiggles Sing Along Extravaganza	View Event
Rock Jam 2013	View Event
Super Band	View Event
The Wiggles Sing Along Extravaganza	View Event

Venue Info

787 road rd
Santa Barbara CA
about this venue about this venue about this venue about this venue about this venue about this venue about this venue about this venue about this venue about this venue about this venue

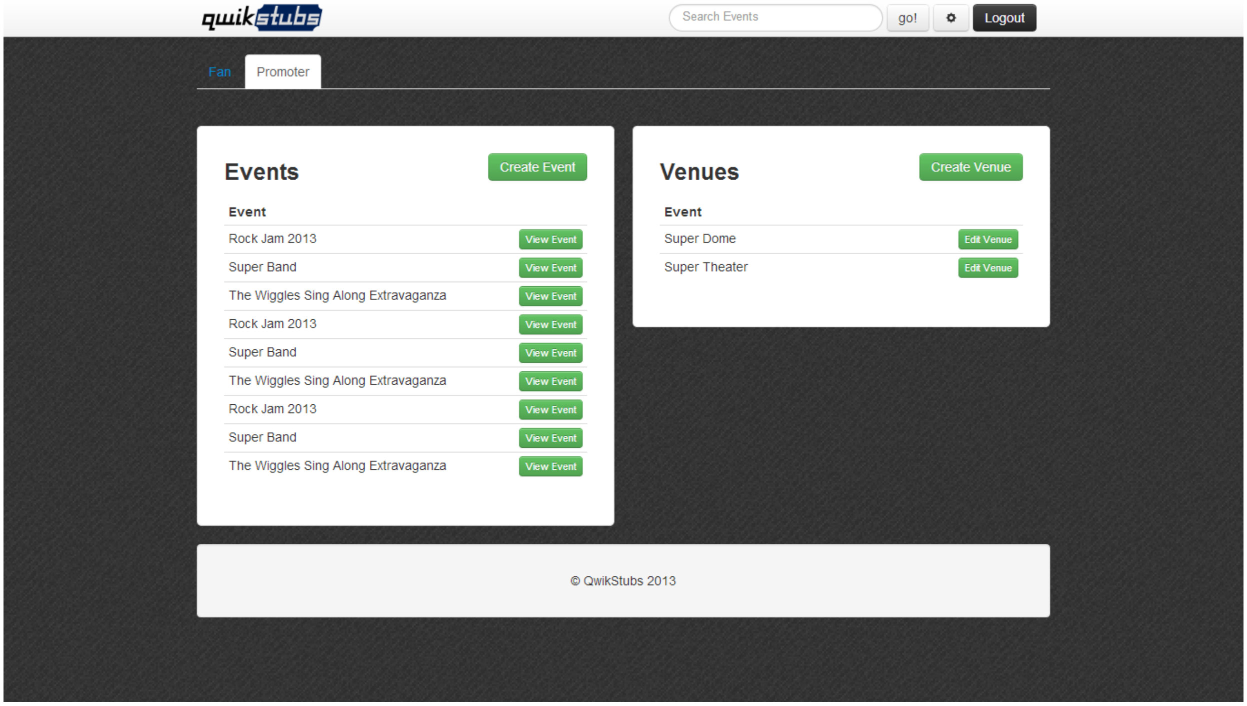
Our Location

© QwikStubs 2013

Figure 14. QwikStubs Venue Page

4.1.4 QwikStubs Promoter Page

In **Figure 15** below, we see the Promoter page. This page is the hub for a user that has a promoter membership associated with their account. From this page a promoter can select to create Venues and Events as well as view ones they already have created.



Events

Event	
Rock Jam 2013	View Event
Super Band	View Event
The Wiggles Sing Along Extravaganza	View Event
Rock Jam 2013	View Event
Super Band	View Event
The Wiggles Sing Along Extravaganza	View Event
Rock Jam 2013	View Event
Super Band	View Event
The Wiggles Sing Along Extravaganza	View Event

Venues

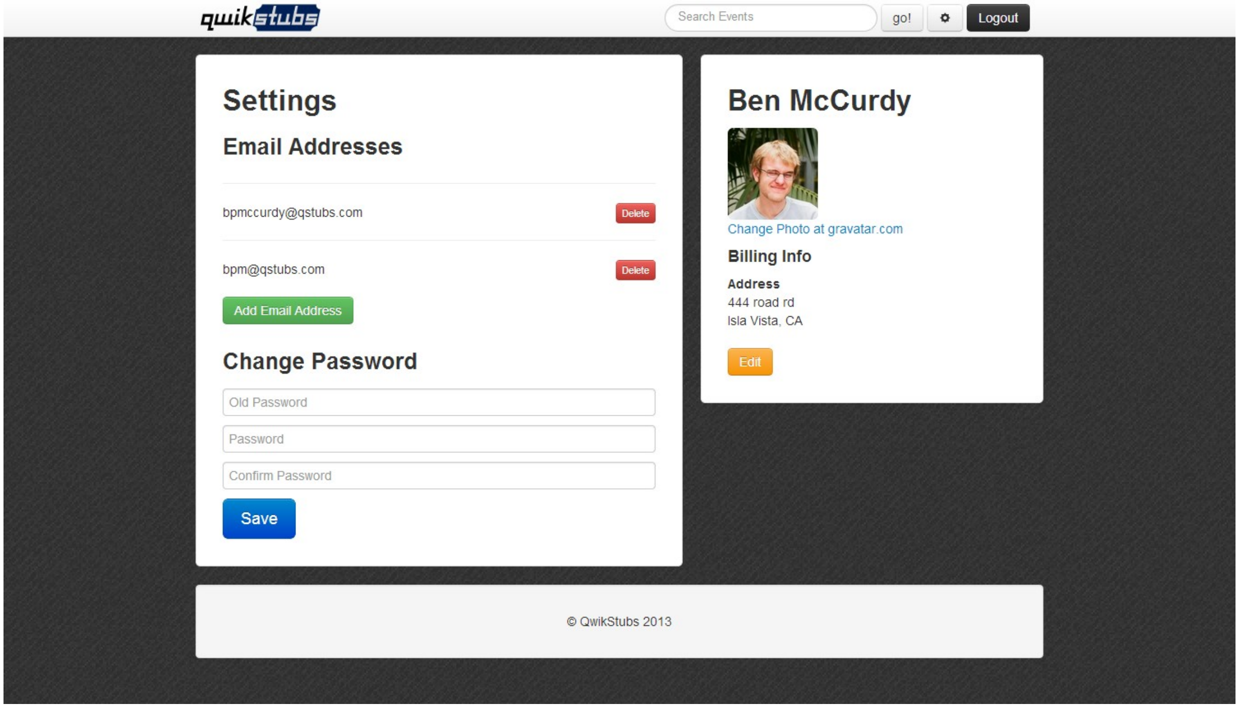
Event	
Super Dome	Edit Venue
Super Theater	Edit Venue

© QwikStubs 2013

Figure 15. QwikStubs Promoter Page

4.1.5 QwikStubs Settings Page

In **Figure 16** on the following page, we see the Settings page. This page is where the user can change all the settings associated with their account. From this page they can manage their email addresses, manage their memberships to promoters, as well as link to their avatar image provided by gravatar. This is also where a user can change their password.



Settings

Email Addresses

bpmccurdy@qstubs.com

bpm@qstubs.com

Change Password

Ben McCurdy

Billing Info

Address
444 road rd
Isla Vista, CA

© QwikStubs 2013

Figure 16. QwikStubs Settings Page

5 Testing

5.1 Unit Testing

Our goal is to have 100% test coverage for the project so we are using RSpec as a testing framework to help us achieve that goal. This is a very common way for testing to be done with Ruby on Rails and we believe will give us the best results. The unit testing allows us to test the models and controllers on a method by method basis, giving us the flexibility to make sure everything is working separately.

We are also integrating Guard: which monitors our file system, and runs the tests associated with a file automatically whenever that particular file has been changed. To go along with Guard, we are adding Spork. This allows us to keep the Rails framework running in the background, so that we don’t have to waste any time waiting for it to startup every time the tests are run.

5.2 Integration Testing

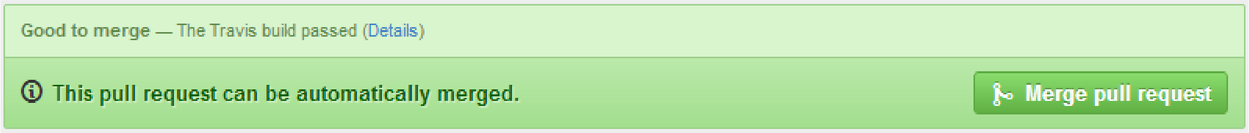
Similarly to the unit testing, we will be using RSpec as the testing framework to help cover the different interactions that happen between the methods. More specifically we will be using Capybara for end-to-end integration testing. This will allow us to test the way that the different component of the project connect and work together. Like the unit tests, we will have this integrated with Guard and Spork.

5.3 Acceptance Testing

For this project, we have written a series of user stories that determine exactly what features QwikStubs intends to successfully provide in its service. As a requirement for a story to be complete, there must be 100% test coverage. This means that the developers that choose to tackle a particular story are responsible for making sure that there is 100% test coverage on all the code that they wrote. These stories will develop alongside the project over time and serve as the tests we are trying to complete in the duration of our two week sprints.

5.4 Continuous Integration

We have set up our Github repository to use Travis CI, a continuous integration service that runs our tests every time we push updates to Github. It is a distributed build system that runs on Travis CI's servers. And to add to that, we have integrated a Travis build-status badge on our Github page that updates with the result of the tests after the latest push. If it breaks the build, Travis will send out an email to the person responsible for the broken commit and tell them to fix it. This allows us to quickly determine if our pushes have broken anything, and therefore quickly address any issues in our code. This fits in really well with having it open source because someone can fork the repo and experiment, then send a pull request with their changes and Travis will be able to tell us whether it passes all the tests or not and if it is good to merge. This would look something like **Figure 17**.



Good to merge — The Travis build passed (Details)

ⓘ This pull request can be automatically merged.

Figure 17. The Travis CI badge on our Github page for pull request