RingBase

Design Specification

March 4, 2014

Group Name: RingBase

Instructor	Chandra Krintz
Course	CS 189A
Lab Section	Wednesday 6PM
Teaching Assistant	Geoffrey Douglas
Date	March 4, 2014
Mentor	Colin Kelley

Andrew Berls	4719696	andrew.berls@gmail.com
Pete Cruz	5226196	petesta@live.com
Alex Wood	4960381	awood314@gmail.com
Shervin Shaikh	5074901	shervinater@gmail.com
Nivedh Mudaliar	4875266	nivedh.mudaliar@gmail.com

Table of Contents

1. High Level Architecture

RingBase is a real-time platform for managing an ‘ad-hoc’ call center. It allows employees of small businesses to see incoming calls in real-time, as as accept them and have the call forwarded to their mobile phone. While on a call, employees can collaborate and edit a shared view containing notes and payout information, as well as seamlessly transferring the call to another agent if need be. The product is targeted towards smaller businesses which can’t afford the six figure price tag of existing call center software. There are currently no known solutions that exist for smaller companies and it is our goal to provide a robust and affordable solution.

Our platform will make heavy use of a real-time telephony API provided by Invoca. Invoca has infrastructure connecting to the public switched telephone network (PSTN), and notifies our own servers (through a pub-sub layer) when a call comes in to a business's phone number, for example. This allows us to greatly simplify our product by not having to write any telephony-related code.

All employee interaction with the platform happens in a web browser (from a desktop computer or mobile phone), but the backend itself is composed of several different components and services. Our frontend uses views constructed with HTML and CSS, with dynamic interactions handled through JavaScript.

1a. MVC Architecture

The MVC architecture consists of three separate components, models, views, and controllers. This model encourages decoupling of the three components and separation of code. As a result, this abstraction designates responsibilities for the components to compose together to deliver a working web application.

Components:

- a *model* - contains data and business logic
- a *view* - a page of HTML and CSS displaying data in the browser
- a *controller* - receives requests from the browser, queries data from models, and renders views

1b. Ruby on Rails

We use Ruby on Rails as the web framework that serves views to the browser, as well as handling basic business logic such as account creation / login. Rails implements the MVC (model-view-controller) paradigm for clean separation of concerns, emphasizes the principles of DRY ("don't repeat yourself") and "convention over configuration" (choosing sensible defaults for common tasks). In addition, Rails will manage persistence of platform data (users and organizations) to a PostgreSQL database.

1c. Amazon Web Services

We will be using Amazon Web Services (specifically the Elastic Compute Cloud, or EC2) to host our application and manage our servers. This allows us to divide our services onto separate virtual instances which can be scaled on demand and prevent any single point of failure. In conjunction with AWS, we will be taking advantage of the Chef automation framework to help with automatic provisioning, configuration, and maintenance of our instances.

1d. Broker

Real-time events and interactions with Invoca's telephony API are handled through the 'Broker', a separate server that manages both sending/receiving/forwarding messages between browsers and Invoca respectively. The Broker utilizes a WebSocket-based server to communicate with browser clients, and the AMQP protocol to set up a publisher/subscriber (pubsub) relationship with Invoca's API. These two services work together to provide a communication hub between browser clients and Invoca. The Broker also has access to a Cassandra database shared with Invoca, a NoSQL database Invoca decided to use, which will be used to store call data.

1e. AMQP

To support publisher/subscriber relationships created by the Broker, we have instantiated a RabbitMQ server, which implements the Advanced Messaging Queueing Protocol (AMQP). This server acts as a messaging broker (independent of own Broker service) and transport layer for AMQP messages sent between our Broker and Invoca's API. Both the broker and Invoca's API can publish and subscribe to channels on the server using AMQP.

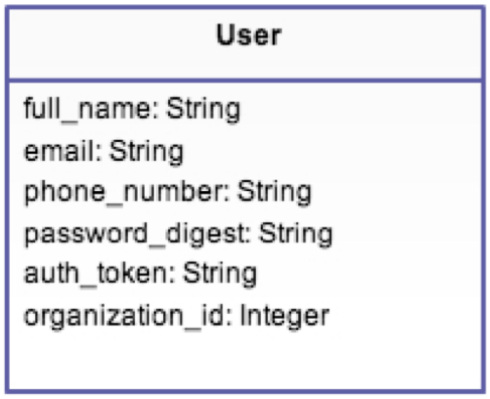
1f. Browser

The browser will provide the UI for the user. We will only be supporting Chrome, Firefox, and iOS Safari, as these browsers support the WebSockets standards which we use extensively. Our UI is created from HTML, CSS, and JavaScript. Bootstrap (CSS) will help us by providing fully customized, built-in containers while JavaScript will dynamically manipulate pageviews. In addition, we will be avoiding certain browser incompatibilities since we are only supporting relatively modern browsers.

2. Models

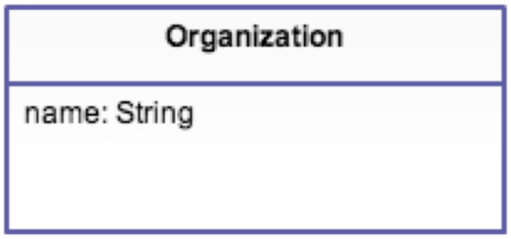
2a. User

The User model represents all users on the platform, including both agents (employees) and their managers. Each user is associated with at most one organization.



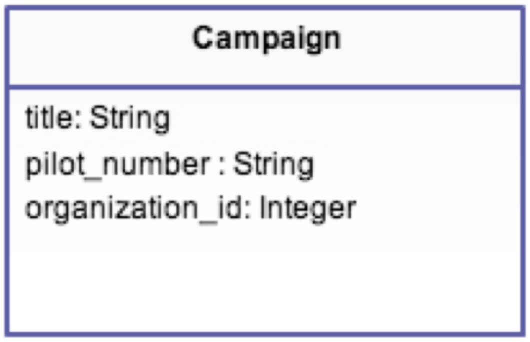
2b. Organization

An Organization represents a business signed up on the platform. It stores the name of the business, and has many users (employees) associated with it (foreign key on the users table).



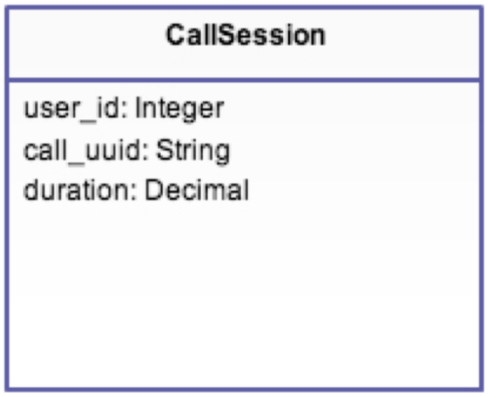
2c. Campaign

The campaign model represents the combination of a title and pilot_number, which is the number that customers will be calling into. Invoca’s servers will then receive the incoming call and forward a message to our servers.



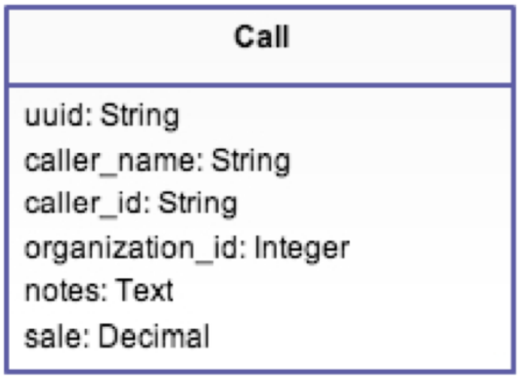
2d. Call Session

A call session represents an agent speaking on the phone with a customer for some period of time. There may be multiple call sessions associated with a single call, for example if an agent transfers the call to another agent midway through.



2e. Call

Our Rails application does not actually have a call model or calls table. Instead, all data about calls (incoming, in progress, and completed) is stored in a Cassandra database shared with Invoca. When a call comes in to Invoca's servers, they will write a record to Cassandra and then notify our application via an AMQP message.



3. Controllers

Our Rails application has a very small number of controllers, as most interactions happen either in our event Broker or our AngularJS frontend.

3a. UsersController

The Users controller is responsible for handling events related to user account creation and profile editing. This controller ensures that the following parameters are required for users in registration: full name, email, password, organization name, and organization pilot number (which is used to automatically create a Campaign).

UsersController Actions

Action	HTTP Method	Path
`new`	`GET`	`/users/new`
`create`	`POST`	`/users`
`index`	`GET`	`/users/:id.json`

3b. SessionsController

The sessions controller handles user authentication and login. Users enter their email and password in the login form (the `new` action), and the `create` action verifies their identity before redirecting them to the main dashboard (or sending them back to the login form with an error message). When a user logs out we delete the cookies in their browser and return them to our welcome page.

SessionsController Actions

Action	HTTP Method	Path
`new`	`GET`	`/login`
`create`	`POST`	`/sessions`

`destroy`	`DELETE`	`/logout`

3c. ApplicationController

The Application controller does not serve any content or interactions directly, but instead acts as a base class for all other controllers and implements various helper methods. For example, it implements functions that prevent anyone who is not logged in from accessing any of the pages from our website except for the main landing page. It also implements functionality to set cookies in the user's browser to store login state. Finally, it adds helpers to find the currently signed in user and their organization, which are used throughout the application as well as providing their data to the Angular frontend.

Helper methods

Method Name	**Description**
`must_be_logged_in`	Require that a user be logged in for a set of actions
`current_user`	Get the currently signed in user
`current_organization`	Get the current user's organization
`signed_in?`	Check if the user is signed in
`login_user`	Set a cookie in the browser
`reject_unauthorized`	Redirect a user back to the login page

4. Frontend

4a. AngularJS Application

We use Rails to serve our JavaScript content, which is our main AngularJS application. This frontend makes HTTP requests to the Rails/PostgreSQL backend to display and manage data such as the currently signed user and the organization they belong to. The frontend also has a two-way communication with the Broker through WebSockets. Messages are sent to Angular when phone calls are received, ended, or updated. Angular event listeners then notify the Broker when an agent clicks to pick up, end, and update calls using the same messaging system. The great power in our frontend is in our ability to update the display of call status/notes/etc to the user without having to refresh on any of the pages.

I. Controllers

The `PhoneCtrl` uses the `Communicator` services to subscribe to messages from the Broker. It then updates the model according to the messages received. It also grabs the current user and organization from the window variable so that it can display the right information specific to that user.

The `CallCtrl` is connected to our call view, it deals with adding notes, changing the sale value and storing the duration for a specific call. Once these values are changed it sends the updated information to our Cassandra database.

The `ModalInstanceCtrl` brings up the modal view when an agent wants to transfer a call. It uses the Agent service to fill itself with all the agents in the current organization.

II. Services

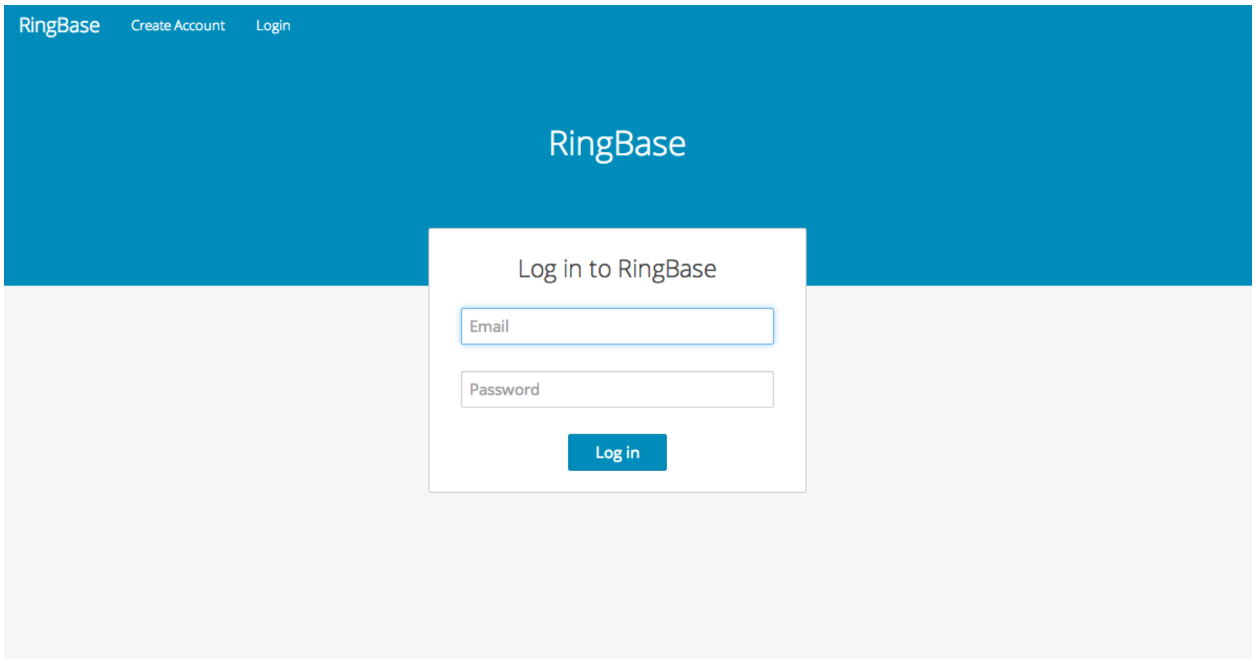
The `Agent` services makes HTTP requests to Rails to get values from the Postgres database. It gets information about all the agents for the current organization.

The `Communicator` services utilizes web sockets to stay connected to our Broker. It subscribes to incoming messages, deals with error handling of those messages, and also sends messages to the Broker when information needs to be updated.

4b. Views

I. Login

The login page is where agents enter their credentials to sign in to the RingBase platform. This form submits data to the SessionsController, which checks whether the username and password match anything already in the RingBase system. If there is no matching account, the form will display an "Invalid email or password" message, otherwise they will be signed in and redirected to their dashboard.



II. Manager Sign Up

The Manager Sign Up page will enable businesses managers to create an organization account and designate themselves as a manager. This will give them additional capabilities within the platform, such as supervising their agents, aggregate reporting capabilities, and editing data for any call in the platform.

III. Manager Invitation

Instead of employees creating accounts directly on the platform, managers have the capability to 'invite' employees to join using their email. After entering a number of emails, the platform sends a message to each employee with a link they can use to enter their information and become a member of the organization.

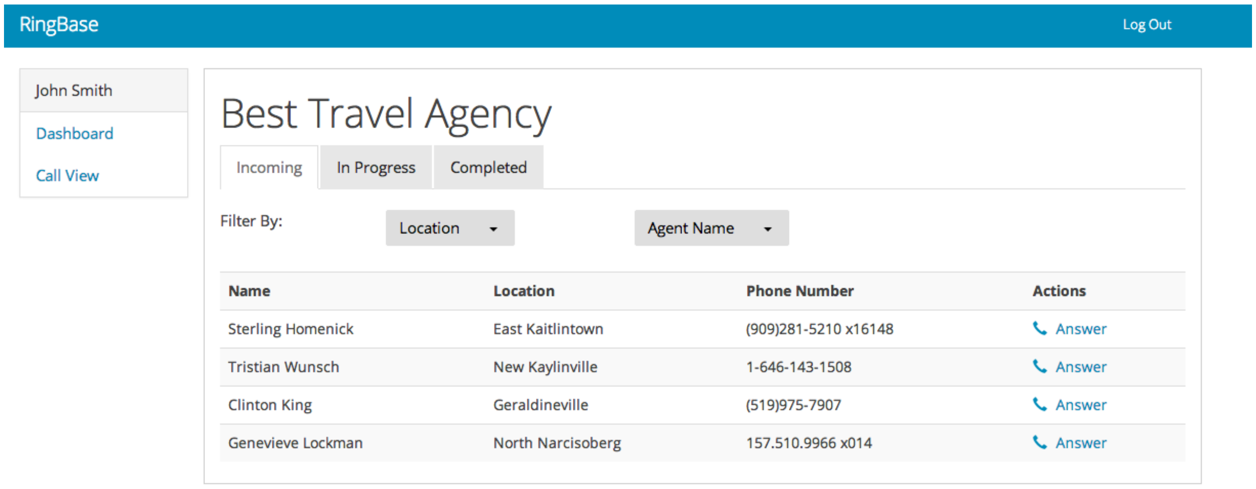
IV. Agent Join

This is the form employees will be taken to after clicking the invitation link in the email sent by managers. It contains fields for standard information such as name, email, and password, and automatically adds the agent to the proper organization after submitting.

V. Call Dashboard

The dashboard is the central information hub for agents. It communicates with the Broker to provide a real-time display of incoming calls and data such as caller location, caller ID, and status to each agent. From the dashboard, agents can choose to answer any call or view notes for an in-progress or completed call

.

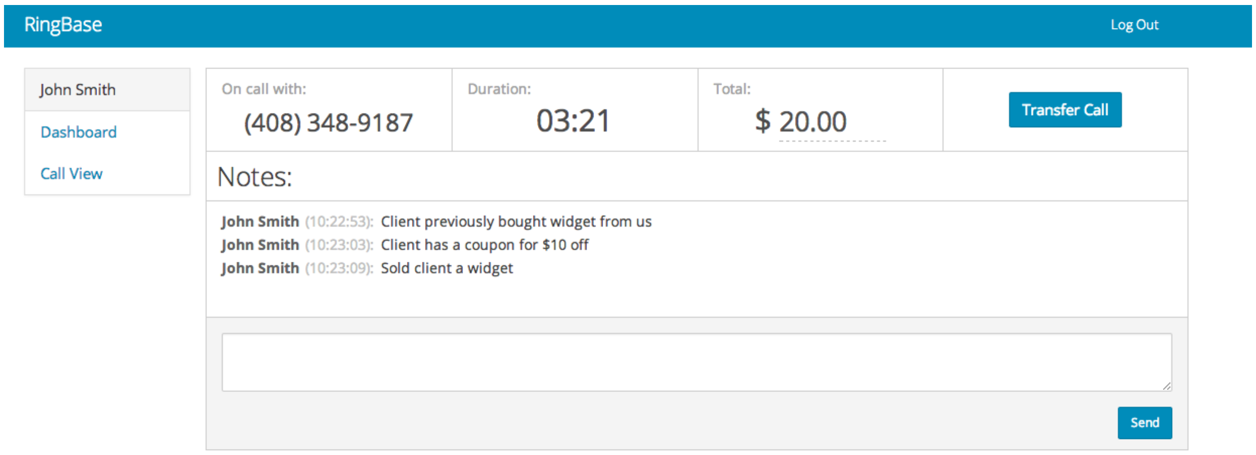


Best Travel Agency

Name	Location	Phone Number	Actions
Sterling Homenick	East Kaitlintown	(909)281-5210 x16148	Answer
Tristian Wunsch	New Kaylinville	1-646-143-1508	Answer
Clinton King	Geraldineville	(519)975-7907	Answer
Genevieve Lockman	North Narcisoberg	157.510.9966 x014	Answer

VI. Call View

The call view is the main display once an employee has accepted a call and is speaking on the phone. On this page, the employee can enter notes and annotations about the call, a total sale amount, track the duration, and transfer the call to another employee if need be. To transfer calls, a modal is displayed which shows all organization agents as well as their status (available or busy) for the employee to select from.



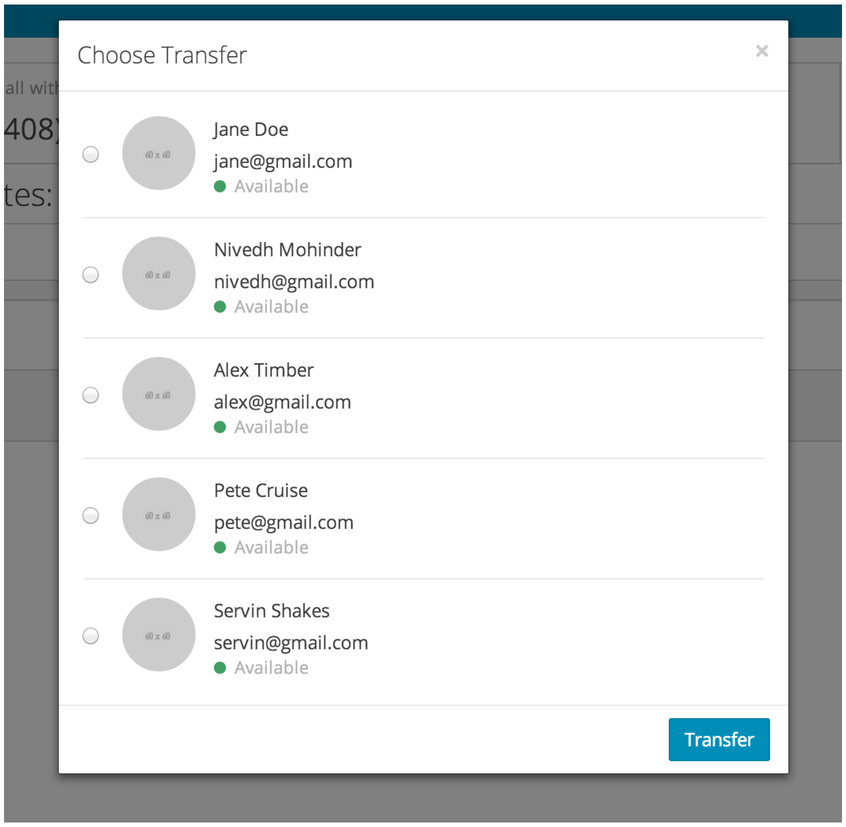
On call with: (408) 348-9187

Duration: 03:21

Total: $ 20.00

Notes:

John Smith (10:22:53): Client previously bought widget from us
John Smith (10:23:03): Client has a coupon for $10 off
John Smith (10:23:09): Sold client a widget


Choose Transfer

Jane Doe
jane@gmail.com
Available

Nivedh Mohinder
nivedh@gmail.com
Available

Alex Timber
alex@gmail.com
Available

Pete Cruise
pete@gmail.com
Available

Servin Shakes
servin@gmail.com
Available

The call transfer modal

5. Broker

The Broker is the server that acts as a communication hub between browsers and Invoca's API.

5a. EventMachine

The Broker relies on the EventMachine framework to provide asynchronous event-driven I/O. EventMachine implements the reactor pattern, which enables fast processing of many concurrent requests. This forms the core of the Broker's interactions with browser clients as well as Invoca's API. Connections to these endpoints are instantiated within EventMachine's "run loop", which accepts incoming requests and messages and dispatches them to the proper service handler. Inside a single EventMachine loop, a Goliath-backed SocketServer is instantiated to communicate with the browser, and a concurrent connection is established with a channel on our AMQP server to publish and subscribe to Invoca’s messages.

5b. Goliath / SocketServer

Goliath is a non-blocking Ruby web server framework powered by EventMachine. It uses asynchronous IO operations, but leverages Ruby fibers (lightweight cooperative concurrency primitives like threads) to let programmers write ‘linear’ code instead of having to deal with multiple levels of callbacks. We used Goliath’s built in WebSocket module in our Broker to build the ‘SocketServer’, which manages real-time interactions with browser clients using the WebSockets protocol. The SocketServer keeps track of currently-connected agents, receives incoming events (such as requests to accept a call), and broadcasts messages to connected peers as necessary (such as a notification that a call has been accepted). The SocketServer uses JSON as a data serialization format, using a simple event schema that the server and Angular frontend agree on.

5c. AMQP

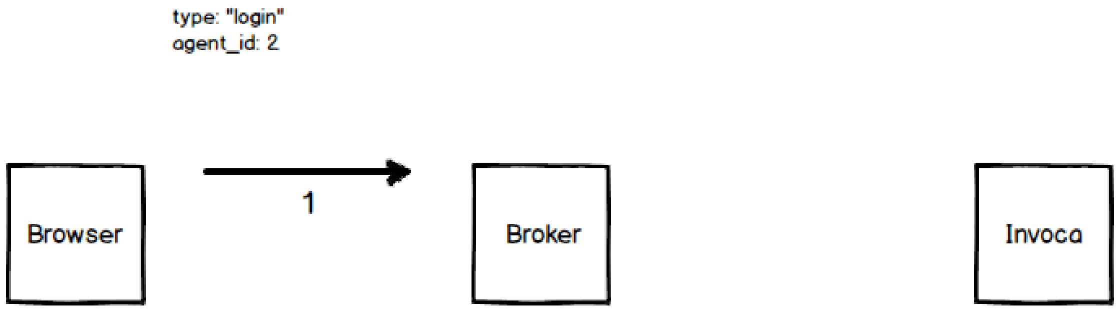
Our broker supports AMQP messaging with the RabbitMQ server with the ruby-amqp client gem. The gem allows the broker to initialize a connection, then either subscribe to a channel on RabbitMQ with a queue, or publish to that channel with an exchange. The broker, with an established publisher/subscriber relationship with the browser via the WebSocket API and Goliath, also subscribes to Invoca’s API over AMQP using RabbitMQ as an intermediate. With these two relationships, it is able to publish messages to one end on behalf of the other, bridging the gap between them.

5c. Event Schema Diagrams

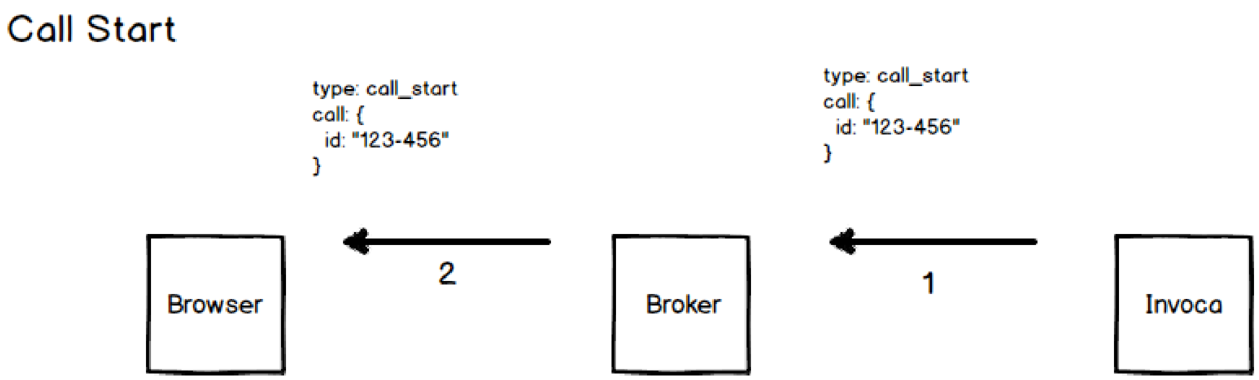
The following diagrams represent the schema and direction of all messages passed between communication endpoints in our platform. “Browser” represents an employee interacting with the platform in their browser, and encapsulates the AngularJS frontend. “Invoca” represents Invoca’s telephony API, which also has an AMQP pubsub subscription set up, and “Broker” is our Broker server which sits in the middle of the two.

The “Login” event occurs after an agent has authenticated through Rails. It is essentially a handshake with the Broker so that real-time messages can be sent and received.

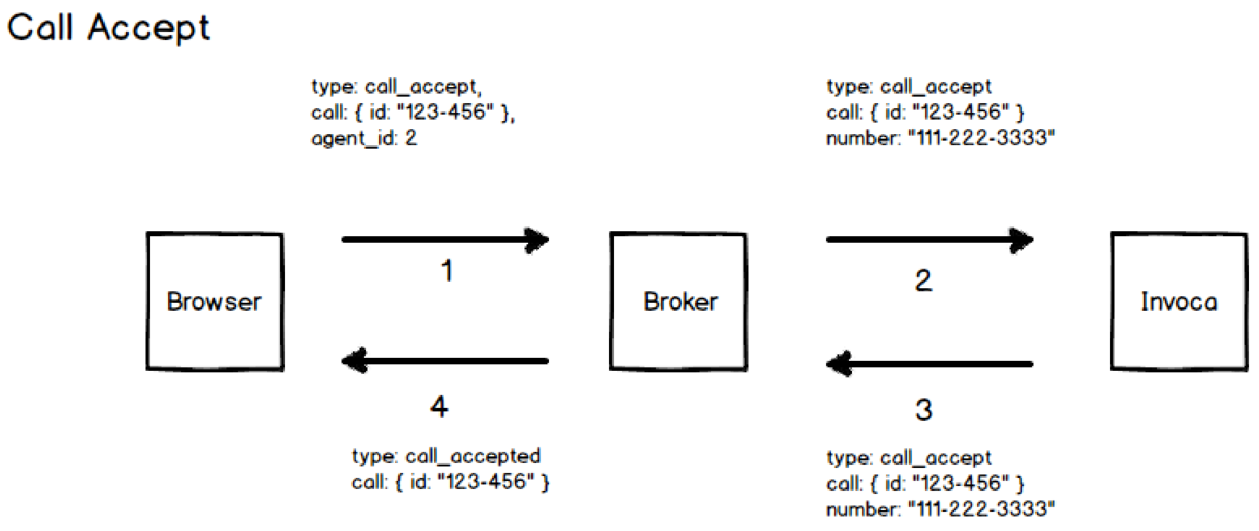
Login



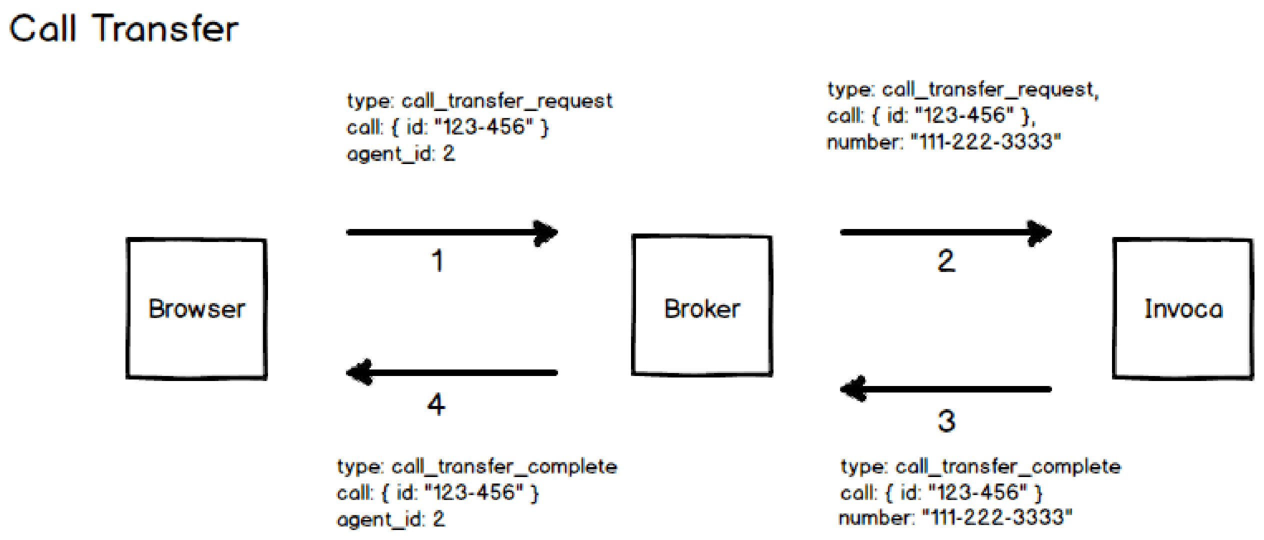
“Call Start” is sent by Invoca when a call is coming in to the business (i.e., a customer has dialed the phone and it’s ringing)



“Call Accept” represents a request by an employee to accept an incoming call and have it transferred to their phone. This is a two-phase action - the message will be propagated through the Broker to Invoca, who will handle transferring the call to their mobile device, and then send back an acknowledgement will will propagate back and signal the browser to properly update the display on receipt.



“Call transfer” represents a request by an employee to transfer the call to another agent. It is a two-phase action similar to Call Accept, in which Invoca will transfer the call and then send an acknowledgement to the browser to update the UI.



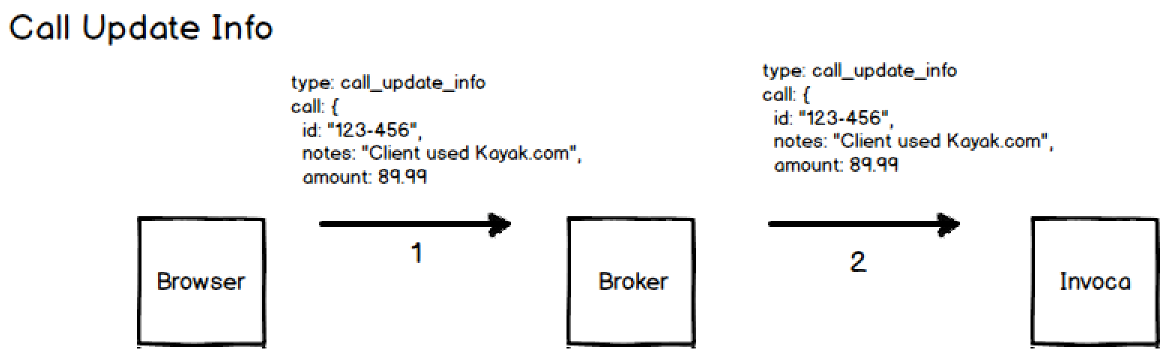
```
type: call_transfer_request
call: { id: "123-456" }
agent_id: 2
```

```
type: call_transfer_request,
call: { id: "123-456" },
number: "111-222-3333"
```

```
type: call_transfer_complete
call: { id: "123-456" }
agent_id: 2
```

```
type: call_transfer_complete
call: { id: "123-456" }
number: "111-222-3333"
```

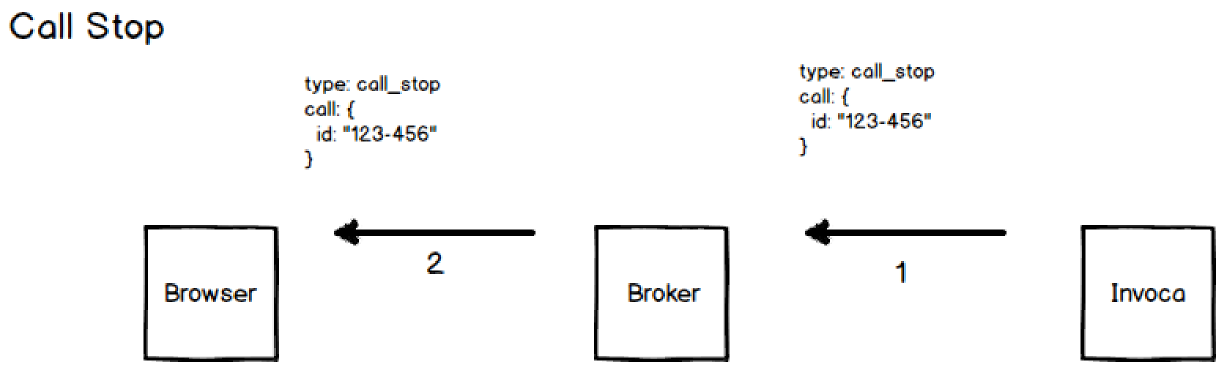
This message is propagated whenever an employee updates call data such as notes or sale information. The Broker handles writing the updated data to Cassandra, and sends a notification to Invoca over AMQP for symmetry (Invoca may choose to ignore the message).



```
type: call_update_info
call: {
  id: "123-456",
  notes: "Client used Kayak.com",
  amount: 89.99
```

```
type: call_update_info
call: {
  id: "123-456",
  notes: "Client used Kayak.com",
  amount: 89.99
```

“Call Stop” is sent by Invoca anytime a call is ended for any reason (this includes customer hangup as well as agent hangup). It signals the browser to update displays as appropriate and redirect the employee back to the main dashboard.



6. Testing

6a. Backend - rspec

We will be using rspec as our testing framework for our Ruby backend. It is based on the principles of Behavior-Driven Development and features a rich domain-specific language (DSL) for writing expressive specs. In addition, it includes a mocking/stubbing framework which will allow us to stub out external dependencies in our system. We are also using the Guard gem, which is a command-line tool which watches our filesystem (i.e., spec files), and automatically runs specs when they change to facilitate rapid development.

6b. Frontend - Karma

To test our Angular front end we will be using Karma as our automated tester. Once it is started, it watches all of our Angular folder and whenever there is a file change it runs all of the tests to make sure they still pass. We're using Karma to do unit testing on our Angular modules, for example our services that communicate with the Broker and Rails. Karma also provides us with a great tool to do end-to-end (e2e) testing. Similar to Selenium tests, it loads up a browser and clicks through the website to make sure our user-browser interaction works as intended.

7. Infrastructure

7a. CI - Travis

Both our Rails application and our Broker (kept in separate repositories) have been configured to use TravisCI as a contiguous integration service. Travis automatically runs our entire test suite on every push, and alerts us if any tests fail. This gives us the ability to know exactly when and where any regressions occur, and when it is safe to merge a pull request (i.e. the build is “green”). We have also added a Travis badge to our open-source repositories which displays the current build status, so that any visitor knows if the build is currently passing or failing.

Sample badge:

7b. Coverage - Coveralls

Our Rails application uses Coveralls for test coverage metrics. Coveralls hooks into Travis, and tells us what percentage of our application code is covered by tests. This gives us a rough indication of how well-tested and reliable our code is at any time, as well as providing an incentive to add more tests to increase our coverage and confidence in our code.

Sample badge: