Software Design Description for Mashbot

George D'Andrea Andrew Gall Josiah Kiehl Cody Ray Vito Salerno

February 15, 2010

Revision History

Name	Date	Reason for Changes	Version
George D'Andrea, Andrew Gall, Josiah Kiehl, Cody Ray, Vito Salerno	16 February 2010	Revised from Reviews	1.1
George D'Andrea, Andrew Gall, Josiah Kiehl, Cody Ray, Vito Salerno	17 January 2010	Initial Version	1.0

Contents

Chapter 1

Introduction

1.1 Purpose

This document serves to expand upon the requirements document into implementation details and technology choices. This document should be referenced when specific features are being implemented.

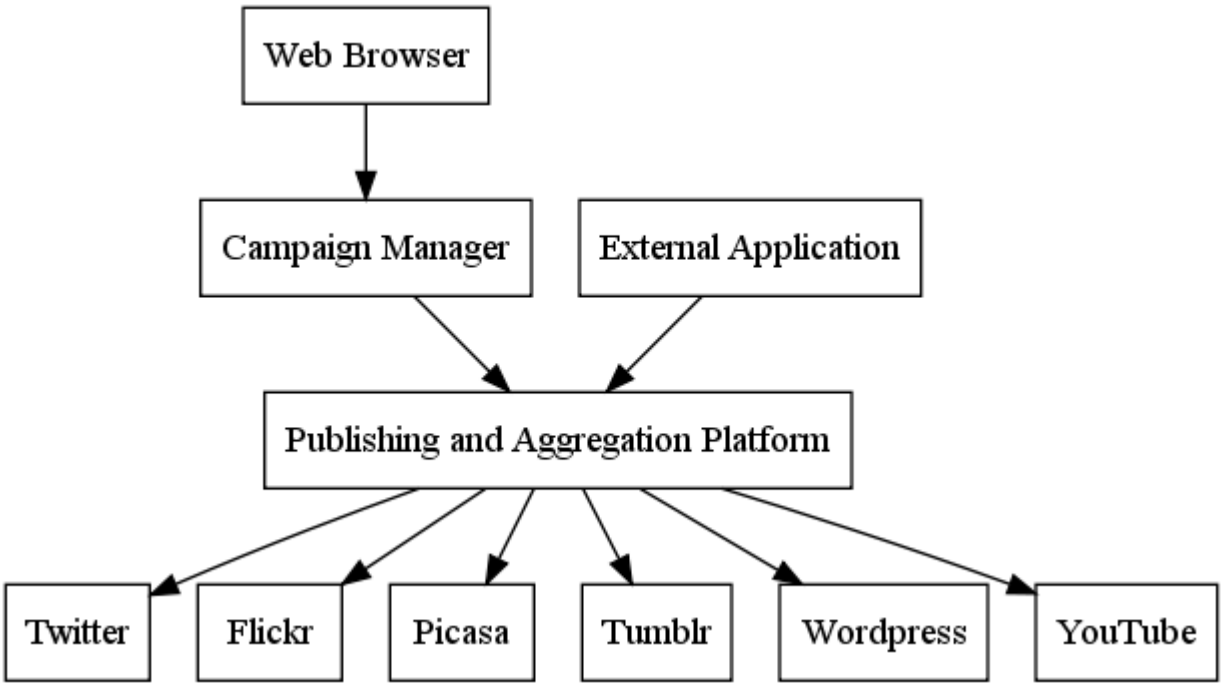
1.2 Scope

This covers the general architecture of Mashbot, as well as the design decisions used to apply that architecture via various appropriate technologies, libraries and frameworks.

1.3 Definitions, Acronyms, and Abbreviations

- API — Application Programmming Interface
- CRUD — Create Read Update Delete
- CSS — Cascading Style Sheets
- HAML — XHTML Abstraction Markup Language
- MVC — Model View Controller
- SASS — Syntactically Awesome Style Sheets
- XHTML — Extensible Hypertext Markup Language

1.4 Context Diagram



Chapter 2

Architecture

2.1 Overview

Mashbot will be implemented using a strict Model-View-Controller architecture. This is augmented by the inclusion of the Publishing and Aggregation Platform, the purpose of which is to abstract the interaction with external service APIs from the application as a whole, thus allowing a pure MVC architecture to be implemented, increasing maintainability, flexibility and extensibility.

2.2 Architecture

- Campaign Manager
  - Data Layer / Model
  - Presentation Layer / View
  - Business Layer / Controller
- Campaign Workers
- Publishing and Aggregation Platform

2.3 Service-Oriented Architecture

Mashbot will be implemented as two distinct yet related services. The Campaign Manager will handle the interaction between the user and the data the Campaign Manager is concerned with, where the Publishing and Aggregation Platform will handle the interaction between external service APIs and the Campaign Manager.

2.4 Survey of Technologies Used

2.4.1 Campaign Manager

- Presentation Layer
  - HAML — HTML replacement markup language, for building web layout structure.
  - SASS — CSS replacement stylesheets, for applying visual styles to the layout built in HAML.

  - jQuery — JavaScript library which provides cross-browser compatibility as well as streamlined Ajax request handling.
  - Google Chart API — Public service provided by Google which generates many different kinds of charts and graphs.

- Business Layer
  - Ruby — Dynamic programming language.
  - Rails — Web application framework written in Ruby which provides a concise Model-View-Controller architecture.
  - Heroku — Rails engine which provides enhanced production deployment via Rails compilation, a fast readonly filesystem, and horizontal scaling.

- Data Layer
  - ActiveRecord — Component of Rails which provides the Active Record pattern of data access, creating data model objects and relationships for interacting with resources in a database.
  - MySQL — Fast and free relational database which plugs into Rails without effort.

2.4.2 Publishing and Aggregation Platform

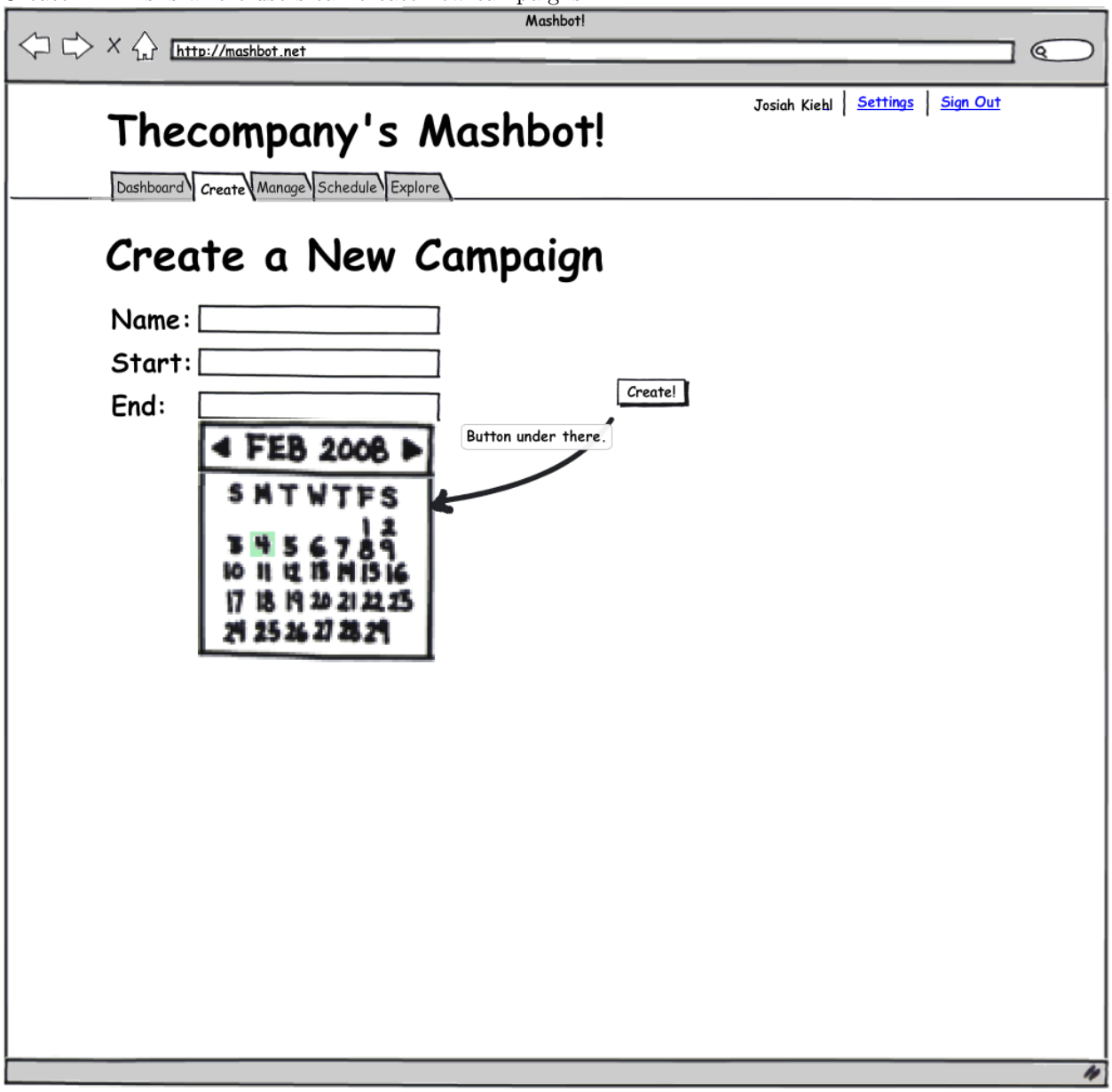
- Java. The team has a broad base of experience with Java and some team members have experience developing an enterprise application on the J2EE. Additionally, the frameworks available are very robust and nice for building web services on top of.
- Web Service Framework - Spring MVC. We chose this platform because it offers a high degree configurable with zero code change. Important options like defaults and the order of handlers on receiving a request is completely configurable in XML.
- Plugin Framework/Manager - Java Plugin Framework.

2.5 Presentation Layer Components

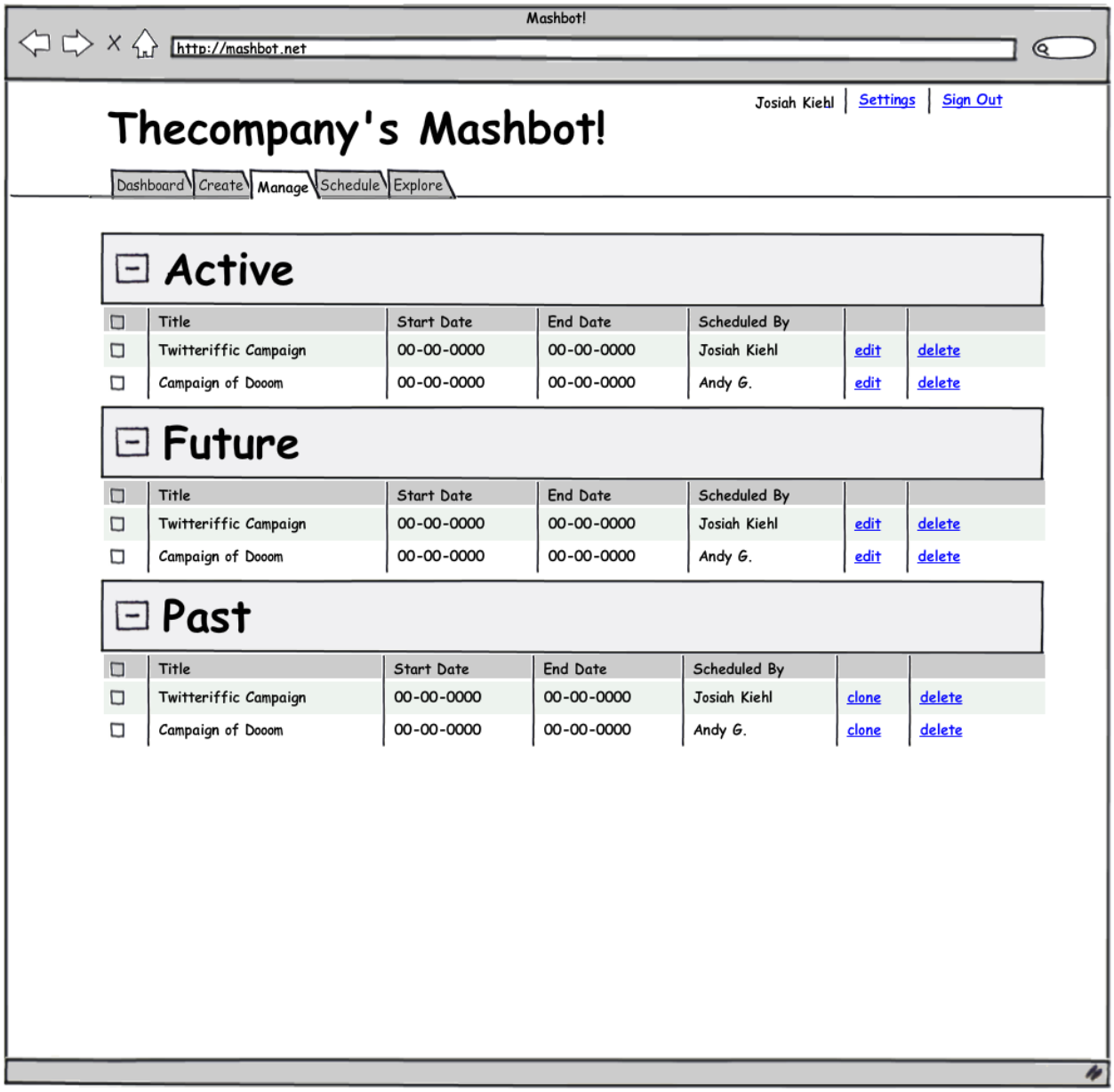
2.5.1 Campaign Views

Campaigns are accessed via the Create and Manage tabs on the primary navigation tabs. Create is for the Create view, Manage is for List, Show and Edit.

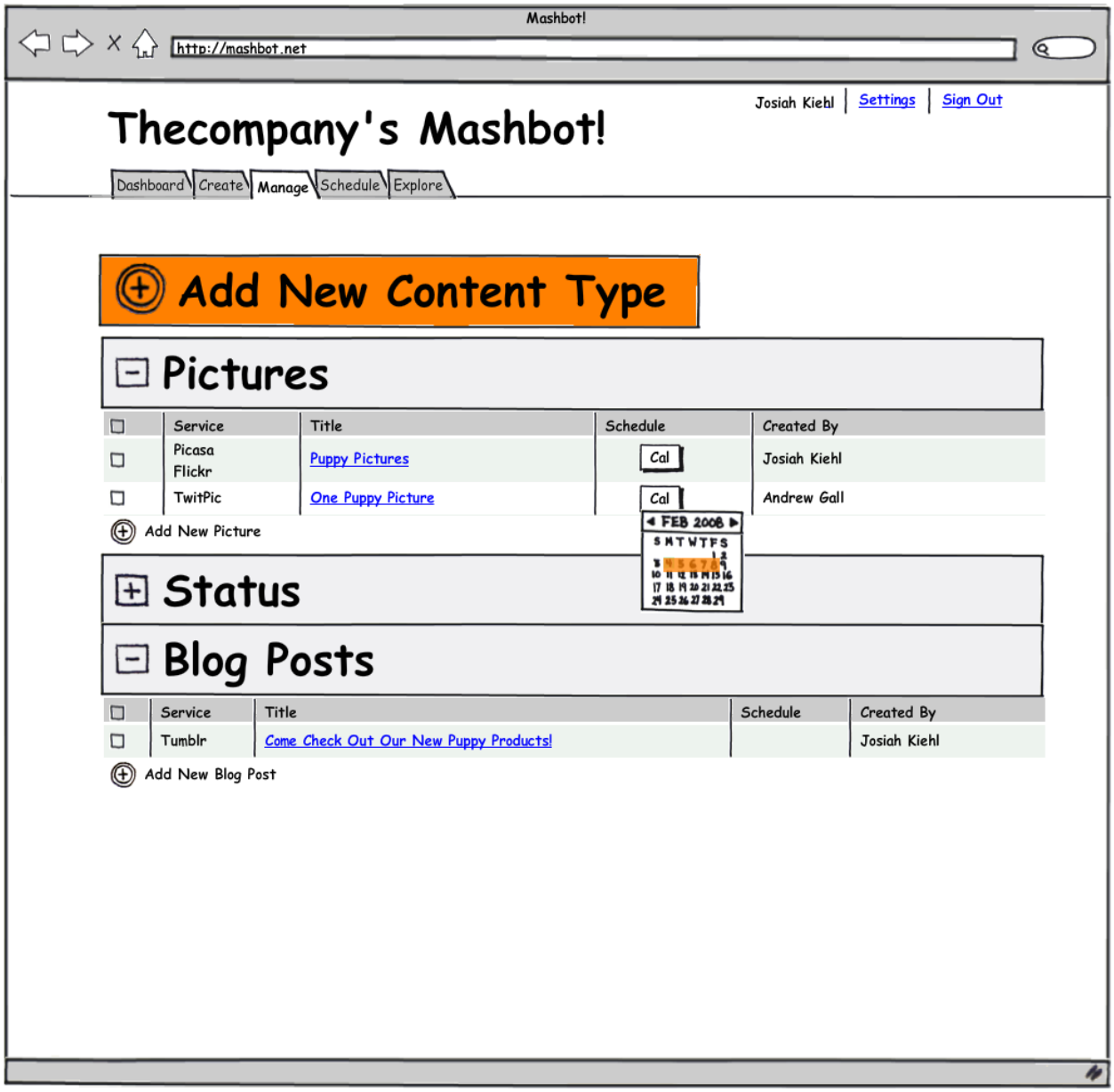
- Create — This is where users can create new campaigns.



- List — This is where users can view, update or delete existing campaigns.



- Show — This view is what is shown when the user wants to view an existing campaign via the Show view. This is also where the Content pieces will be listed.



- Edit — This is virtually the same view as Create, however this will be prepopulated with the existing content of the given Campaign.

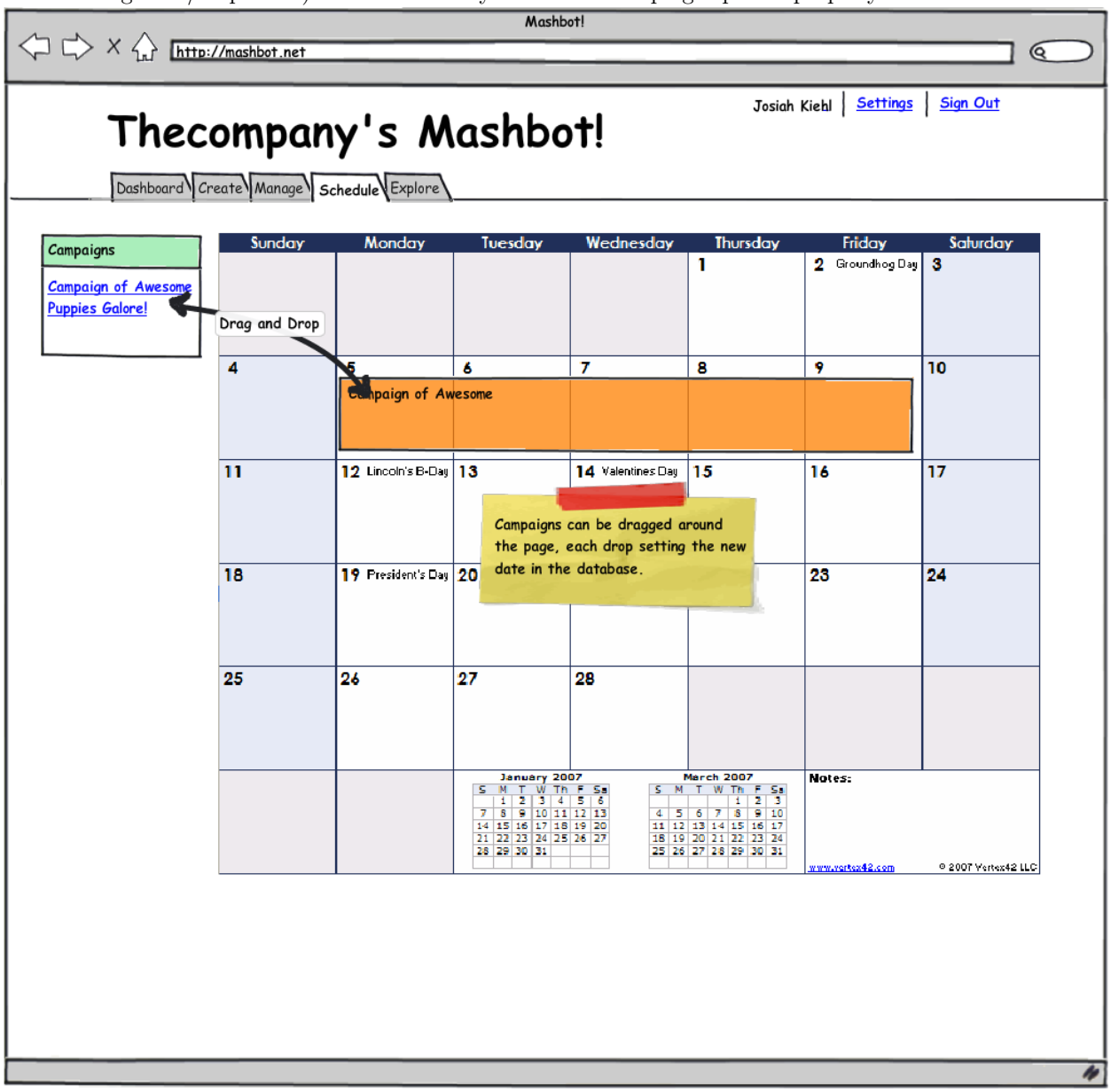
2.5.2 Content Views

Content pieces are included inside Campaigns. These views are accessible via the Show view of a Campaign for the corresponding Campaign id.

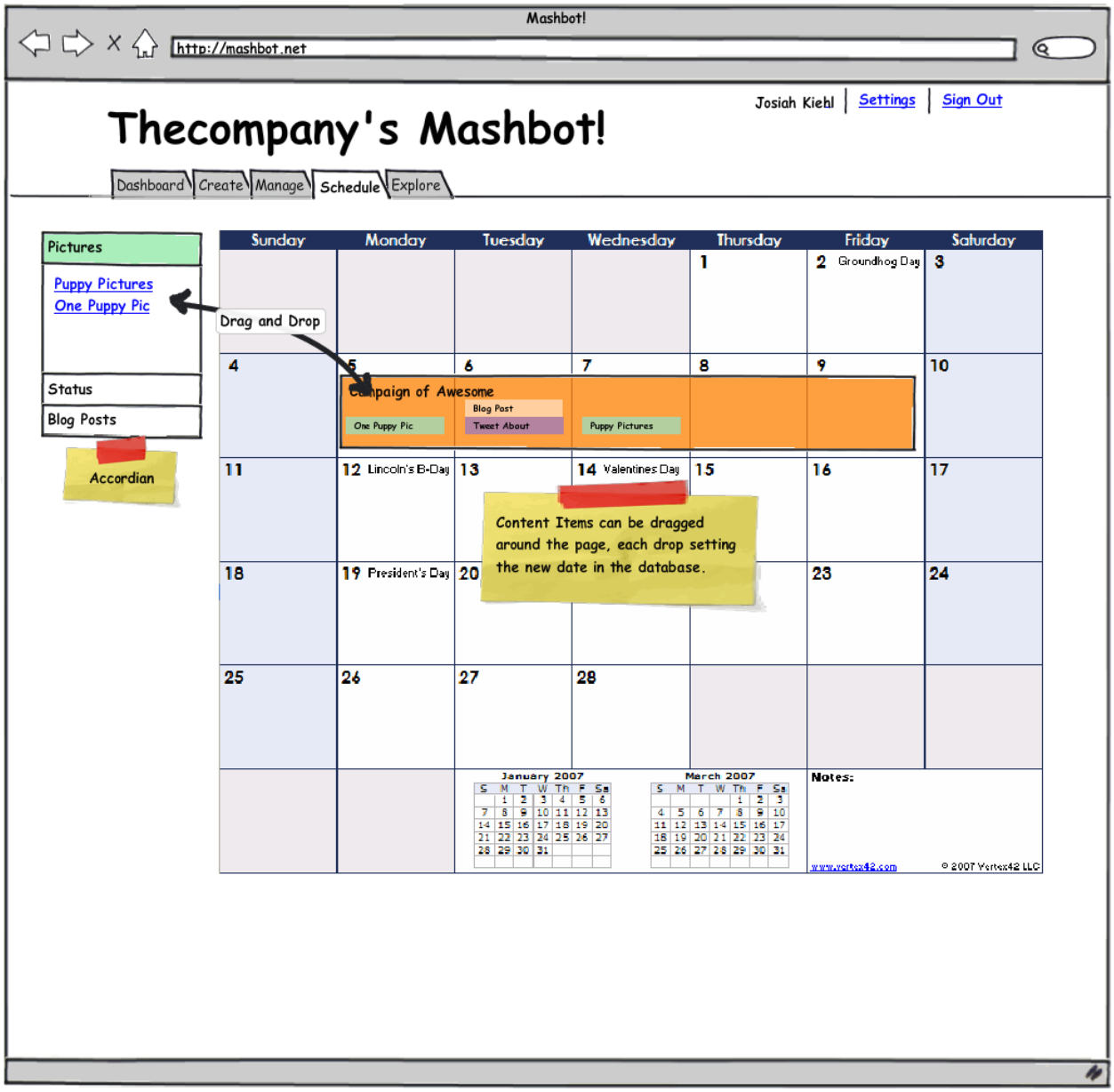
- Create — When on the Show view of a given campaign, the user can enter the Create view for Content.
- Show — This is how the user previews the Content they have created.
- Edit — This is virtually the same view as Create, however this will be prepopulated with the existing content of the given Content.

2.5.3 Scheduling Views

- Primary Scheduling View — consists of a list of Campaigns available to be scheduled (ie: they do not have existing start/stop dates) as well as already scheduled Campaigns placed properly on the calendar.

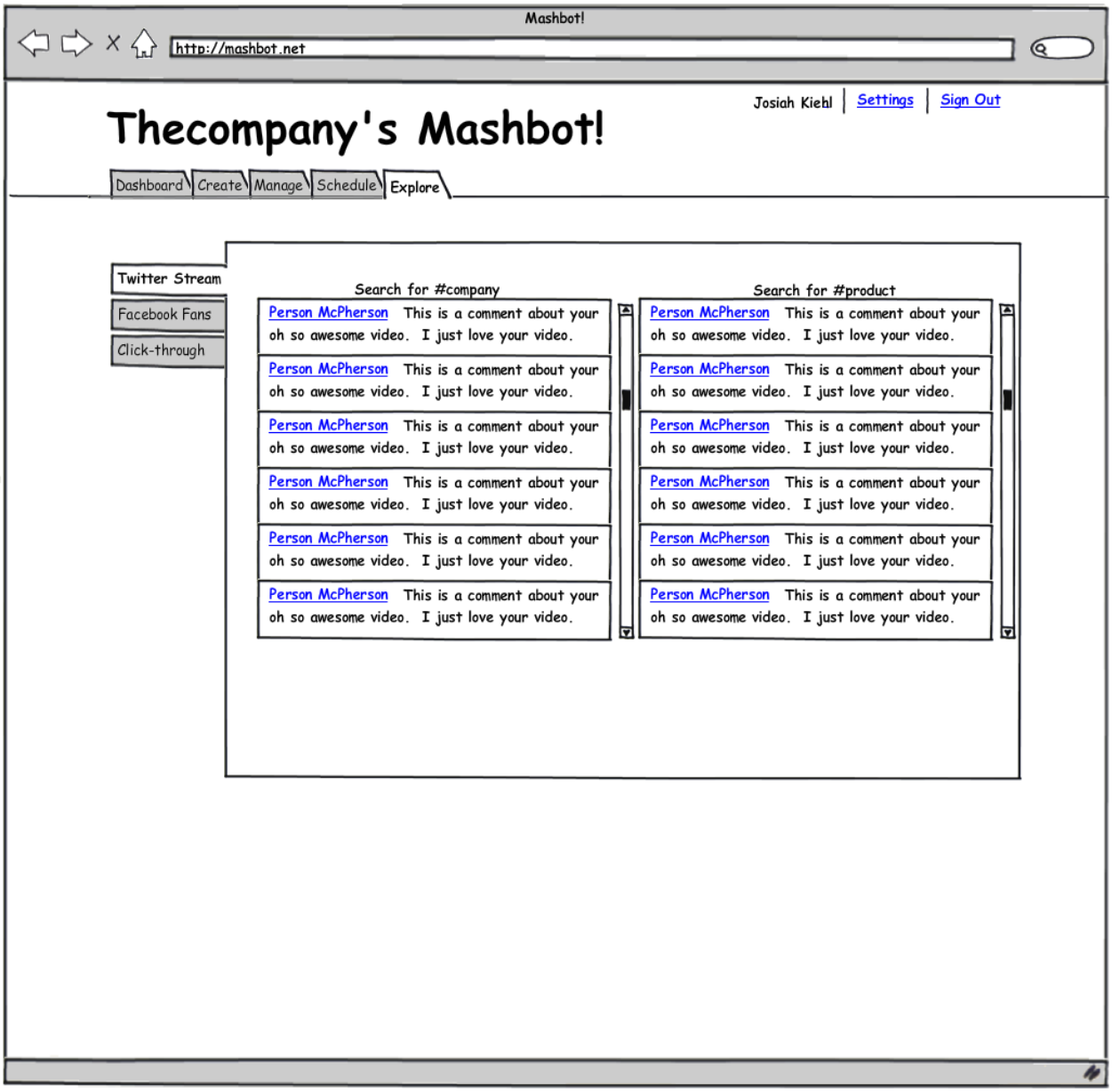


- Content Scheduling View — similar to the Primary Scheduling View, however the items available to be scheduled here are the individual content pieces of the Campaign. This is accessed via selecting the Campaign from the calendar, or via the List Campaign or Show Campaign views.

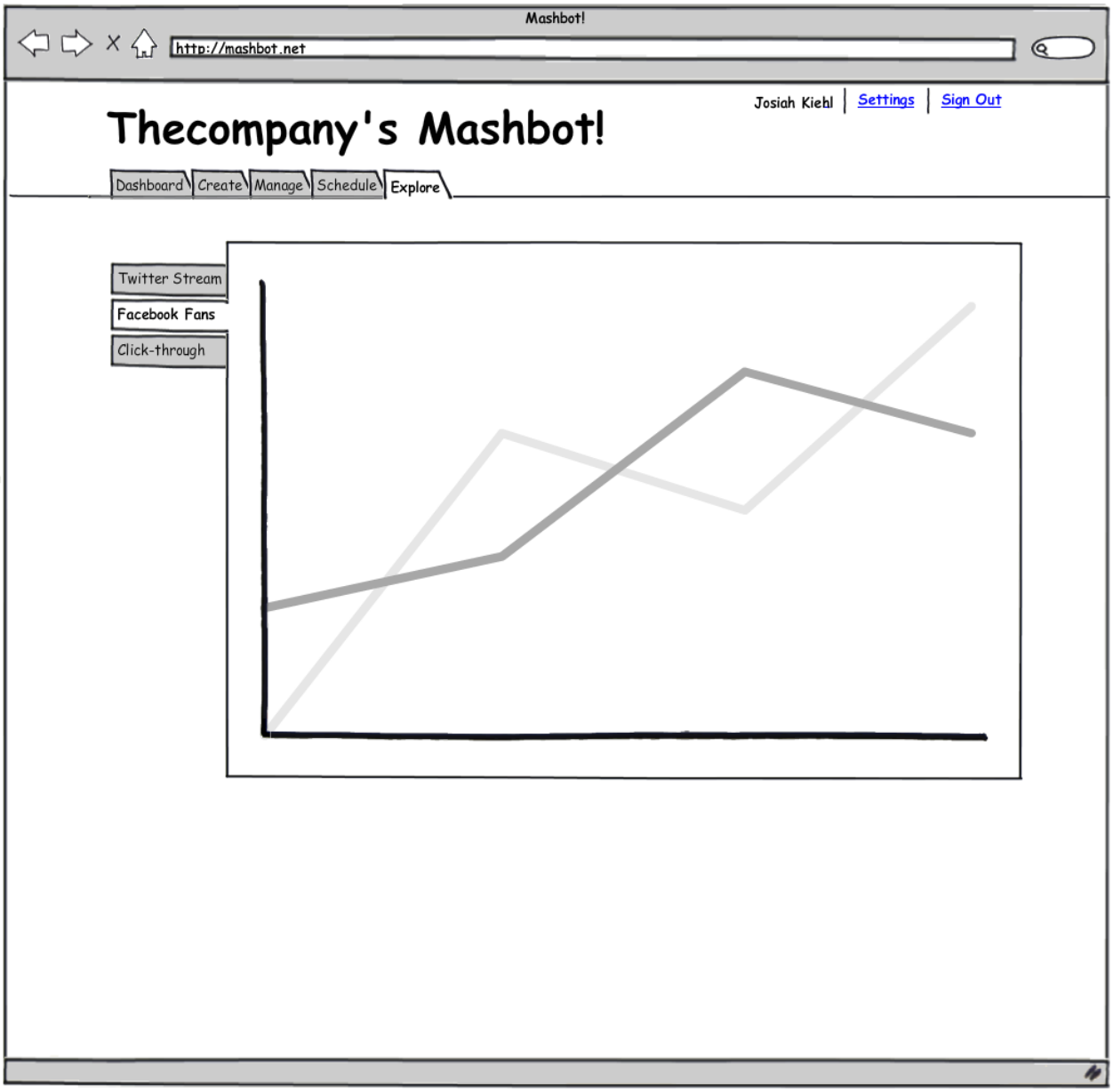


2.5.4 Explore View

Here the user will have several available "Insight Views." These are dependent on which plugins exist in the Publishing and Aggregation Platform, however there will be some provided by the Campaign Manager alone. These will provide charts that are layerable, such that multiple charts can be seen on the same graph.



- Plugin Independent
  - Clickthrough tracking — Any time a link is generated via Mashbot, it is given a special redirecting URL that will allow Mashbot to track how many times the link has been clicked.
  - Rate of publishing — How often does the user tweet/blog/etc. This will most likely be used to correlate frequency with user engagement.
- Plugin Dependent
  - Facebook Fan tracking — A line chart of how many fans the user's fan page has.



– Twitter Follower tracking — A line chart of the number of twitter followers the user's Twitter account has.
– Number of times retweeted — A line chart of the number of times a tweet of the user's has been retweeted.

2.6 Business Layer Components

2.6.1 Session and Authentication

At the user layer, the authentication engine will use a secure password / token based system. In addition to the default username/password option, Mashbot will also support login through OpenID, Facebook Connect, and Twitter/OAuth.

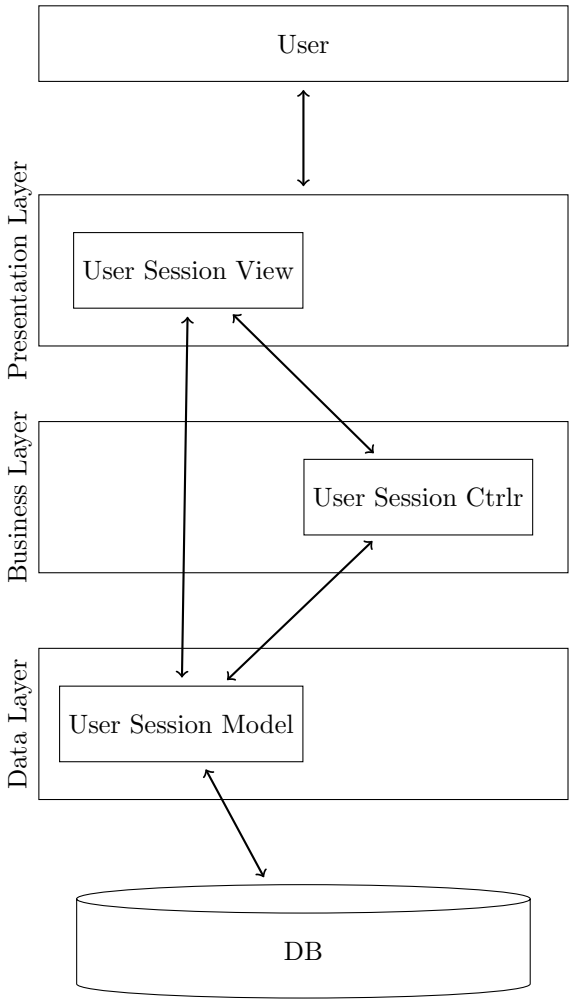
The Mashbot authentication system will provide three different kinds of security tokens that tasks may need, rather than providing users with a token for each task (reset passwords, etc.). The tokens are the "keys" to do the following:

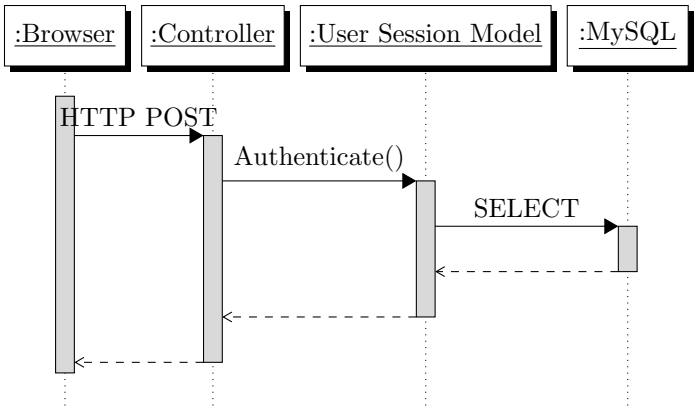
- Persistence token: Used internally, it is stored in your cookies and sessions to persist the user. This is much more secure than plainly storing the users id.
- Single access token: Use this for a private feed or API access. For example, `www.example.com?user_credentials=[single access token]` grants access but does NOT persist.
- Perishable token: Great for authenticating users to reset passwords, confirm their account, etc.

Additionally, all cryptographic routines will be provided by external, third-party verified modules to ensure that their implementation is correct and secure. The authentication framework itself will be largely provided by a widely-used open source Ruby library known as authlogic.

Log In

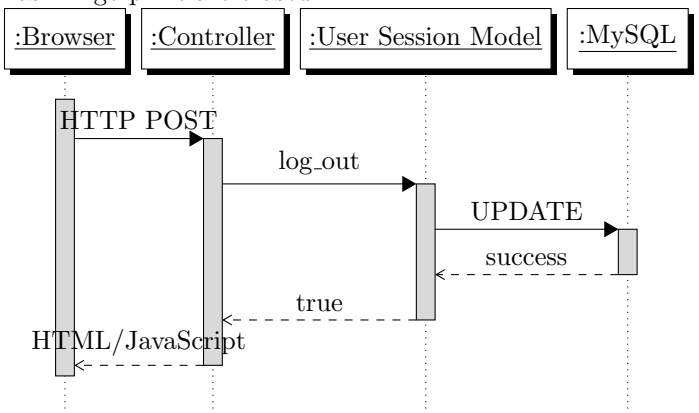
For password-based authentication, the password will be processed through a one-way hash function (SHA512 by default) and only the hashed password will be stored in the database. This mechanism avoids the potential of password recovery if the database becomes compromised.





Log Out

During the logout process, the session is destroyed. Both the session cookies and the corresponding server-side hash fingerprint are erased.



OAuth

At the API layer, Mashbot will use the OAuth standard. OAuth is an open protocol to allow secure API authorization in a simple and standard method from desktop and web applications. OAuth allows users to share their private resources (e.g. photos, videos, contact lists) stored on one site with another site without having to hand out their username and password. OAuth allows users to hand out tokens instead of usernames and passwords to their data hosted by a given service provider. Each token grants access to a specific site (e.g. a video editing site) for specific resources (e.g. just videos from a specific album) and for a defined duration (e.g. the next 2 hours). Thus OAuth allows a user to grant a third party site access to their information stored with another service provider, without sharing their access permissions or the full extent of their data.

OpenID

OpenID is an open, decentralized standard for authenticating users which can be used for access control, allowing users to log on to different services with the same digital identity where these services trust the authentication body. OpenID can replace the common login process that uses a login-name and a password, by allowing a user to log in once and gain access to the resources of multiple software systems

Session Handling

Mashbot uses a cookie-based session store and avoids storing confidential information in its session. In order to prevent attacks by client-side cookie modification, sessions have a SHA512 fingerprint attached and are hashed with a secret stored on the server. Using a cookie-based session increases performance, avoids the "server stickiness" issue associated with horizontal scaling, and increases performance due to an avoided database query on every page load.

2.7 Data Layer Components

2.7.1 ActiveRecord

ActiveRecord is a library which maps database tables to objects. It allows the columns of tables to be accessed as attributes of a Ruby object. Additionally, it handles all of the relationships between tables as well as validation for input.

2.7.2 MySQL

MySQL is a fast, free relational database management system solution.

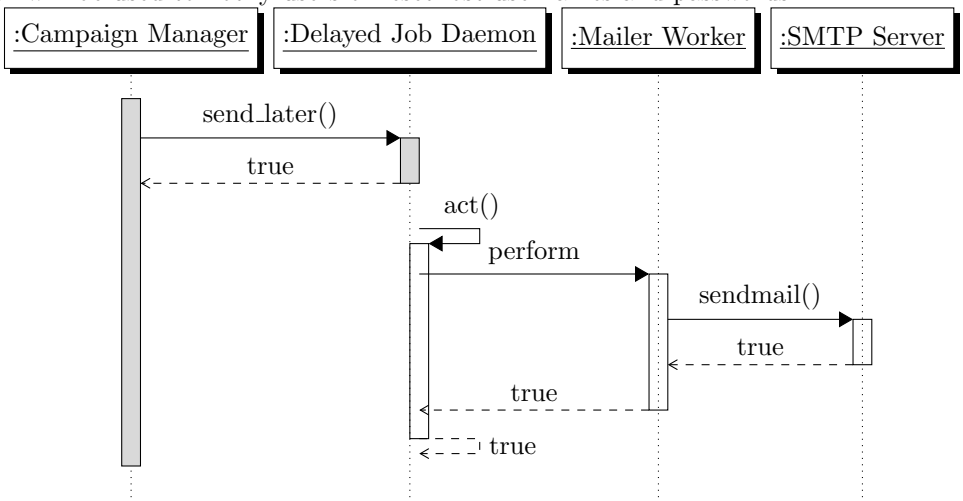
2.8 External Components

2.8.1 Publishing and Aggregation Targets

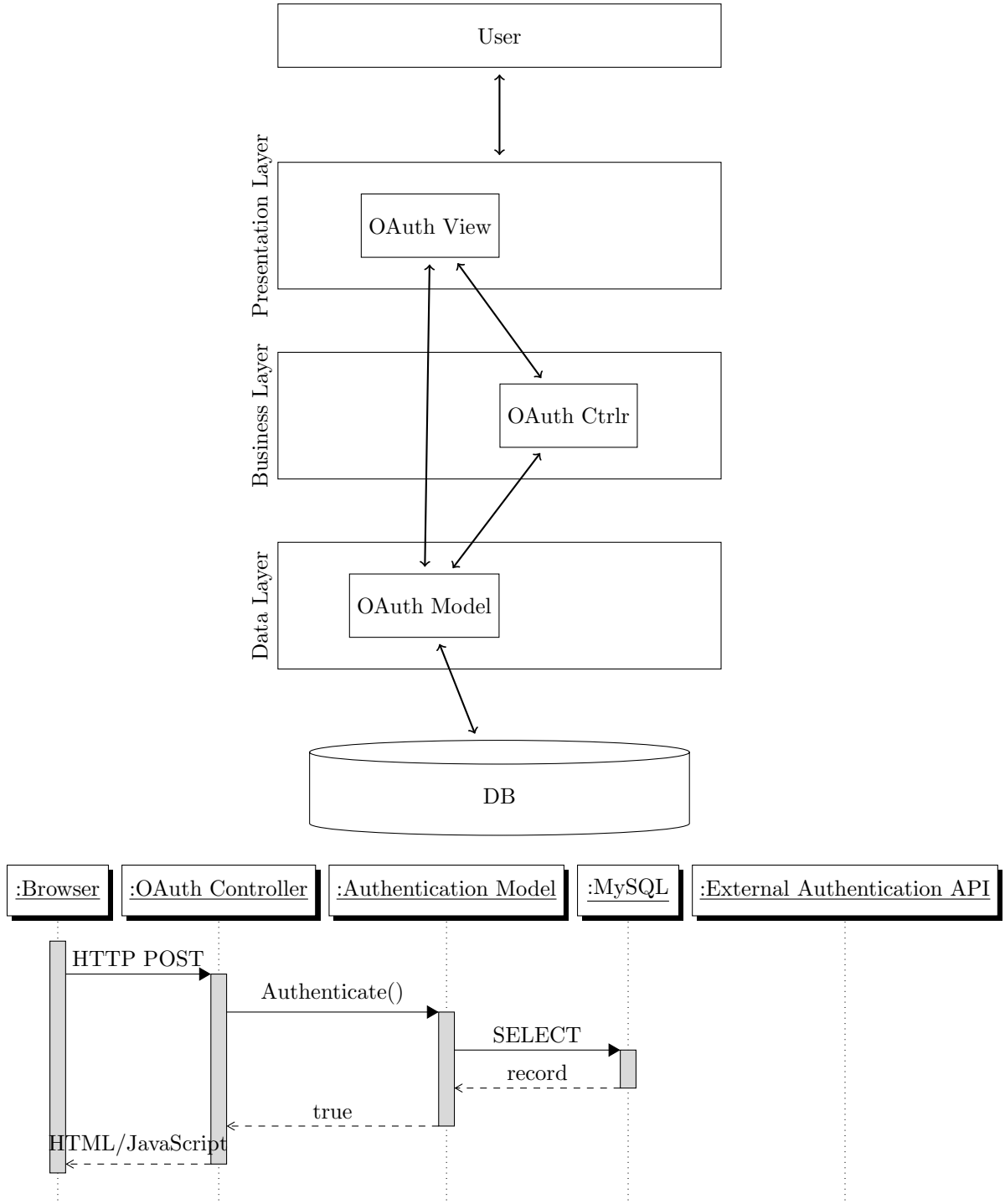
- Flickr
- Picasa
- Twitter
- Tumblr
- Wordpress
- YouTube

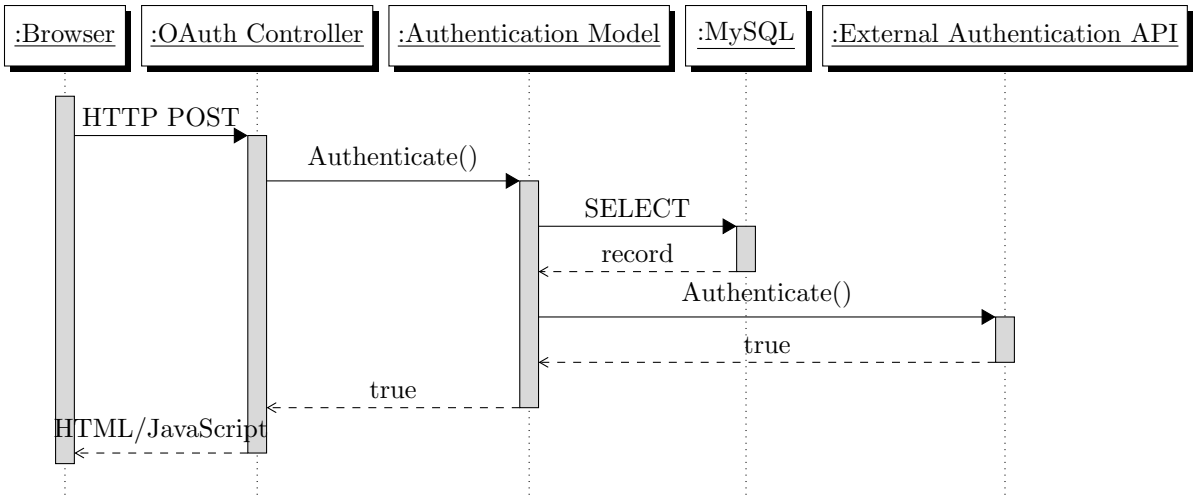
2.8.2 Email/SMTP Service

Email will be used to notify users of reset lost usernames and passwords.



2.9 External Authentication via OpenID

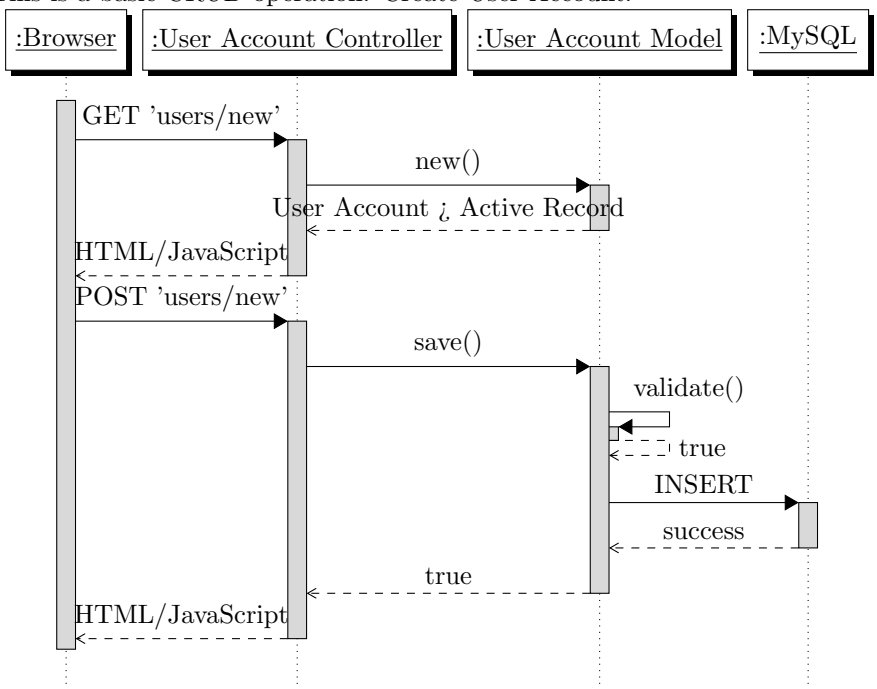




2.9.1 User Accounts

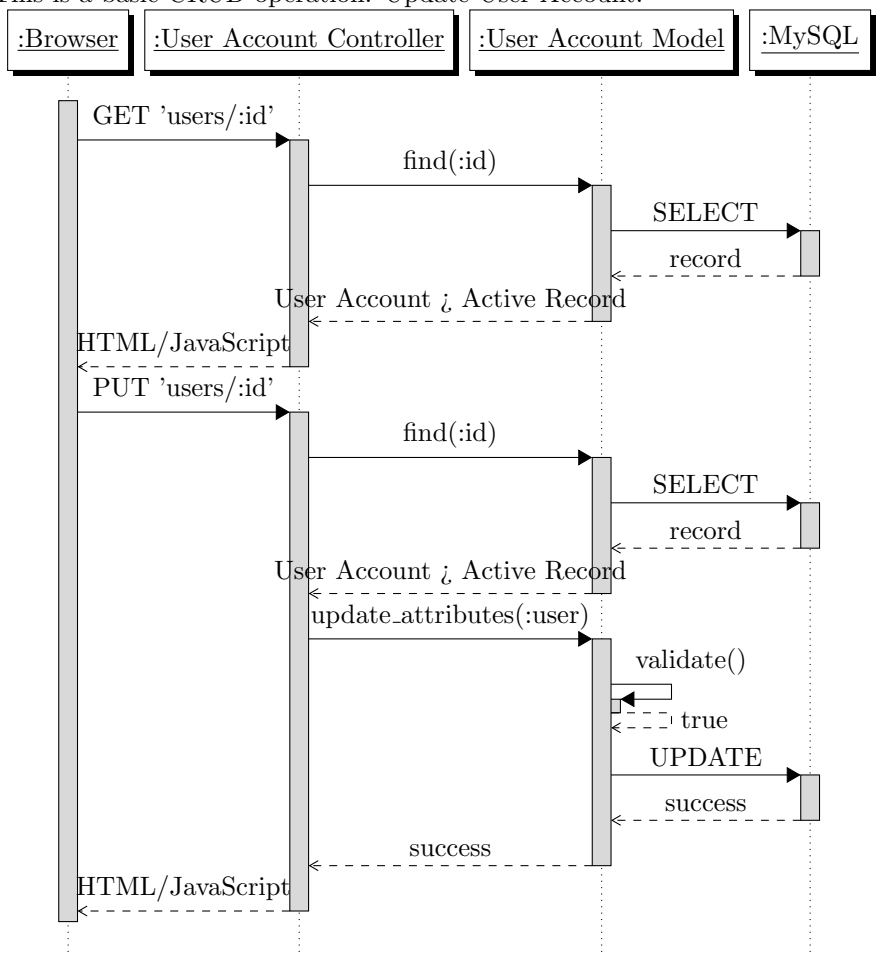
Create User Account

This is a basic CRUD operation: Create User Account.



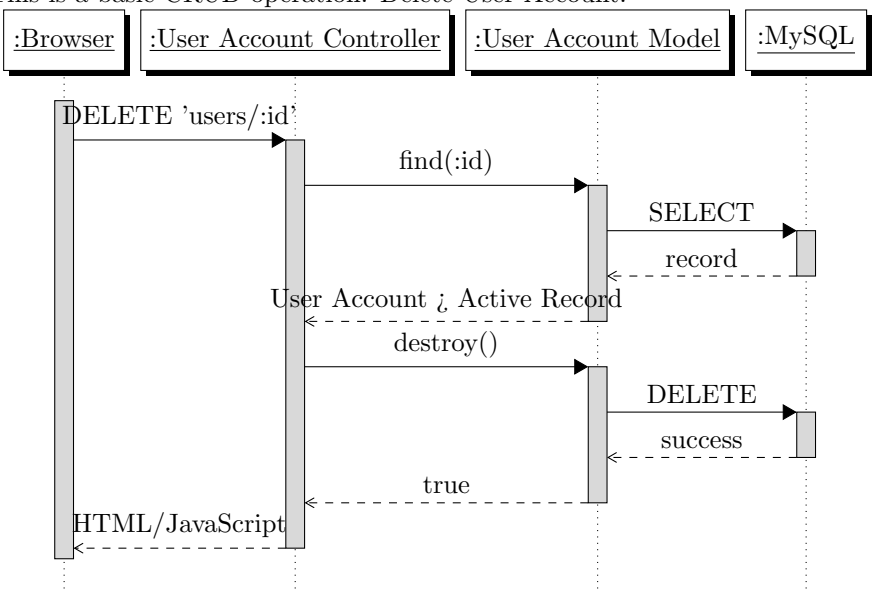
Update User Account

This is a basic CRUD operation: Update User Account.



Delete User Account

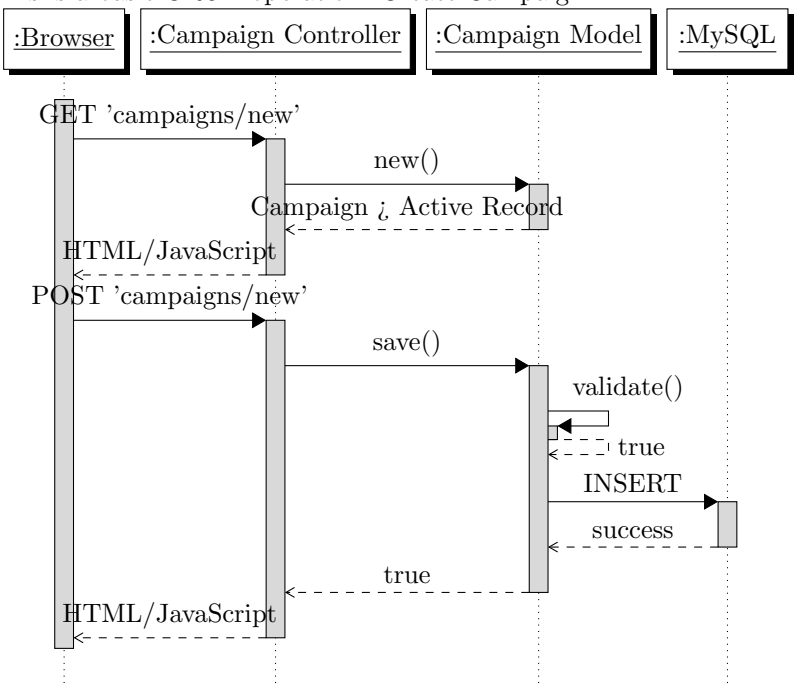
This is a basic CRUD operation: Delete User Account.

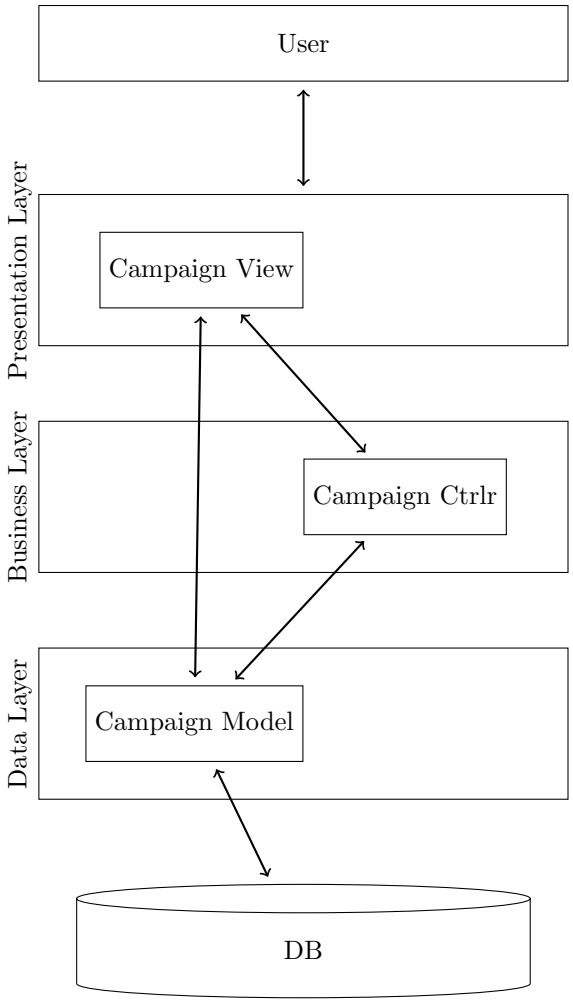


2.9.2 Campaigns

Create Campaign

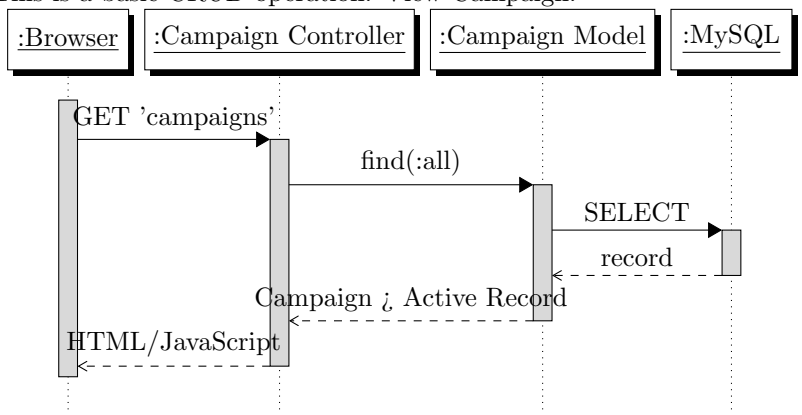
This is a basic CRUD operation: Create Campaign.



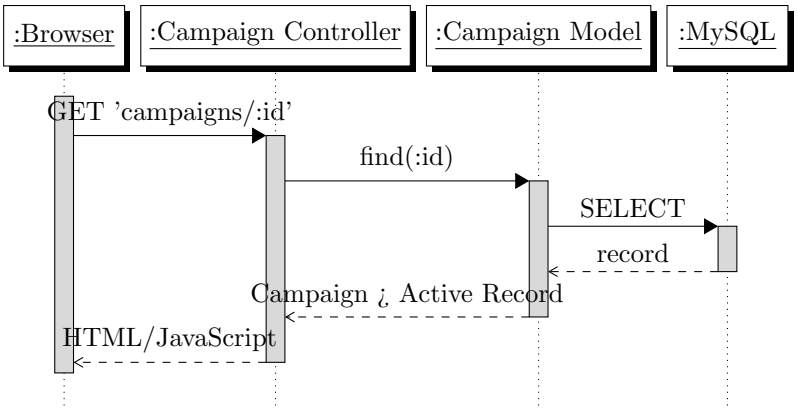


View Campaign

This is a basic CRUD operation: View Campaign.

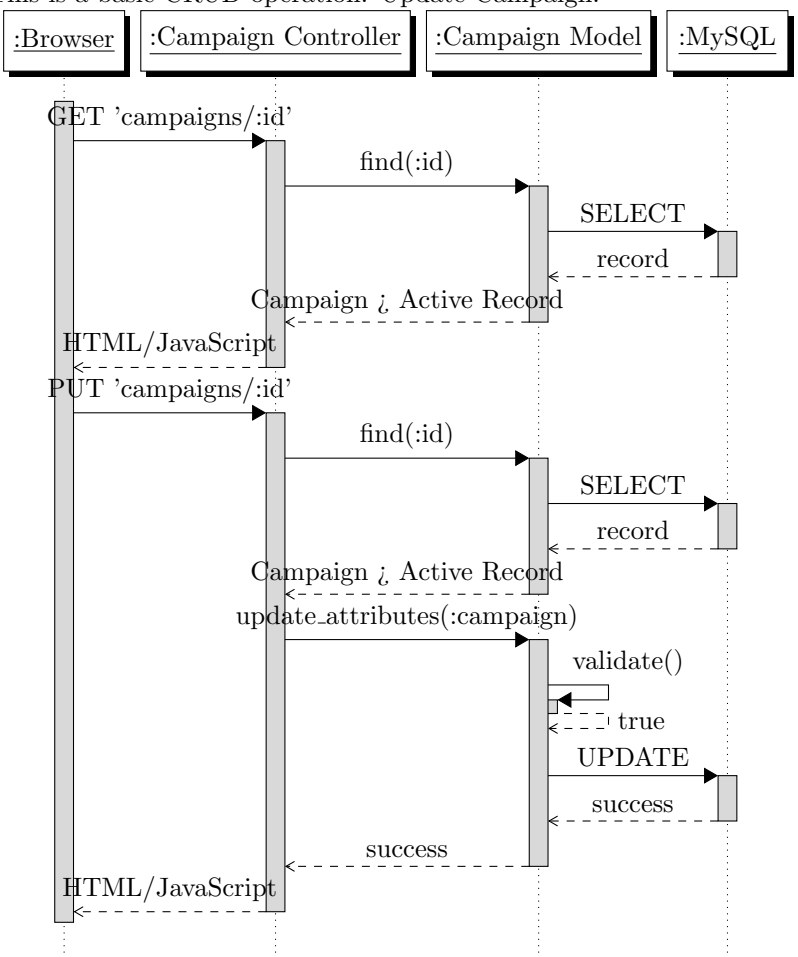


This is a Read operation, similar to the above, but for one Campaign.



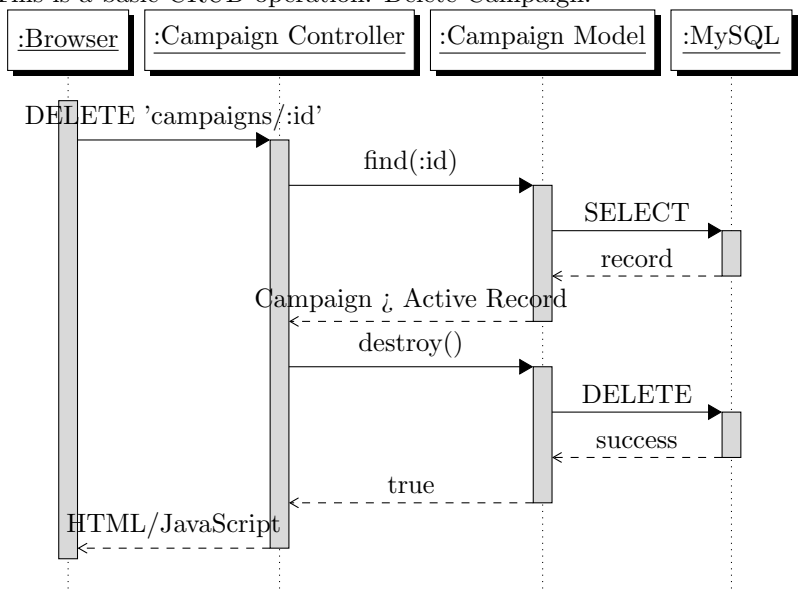
Update Campaign

This is a basic CRUD operation: Update Campaign.



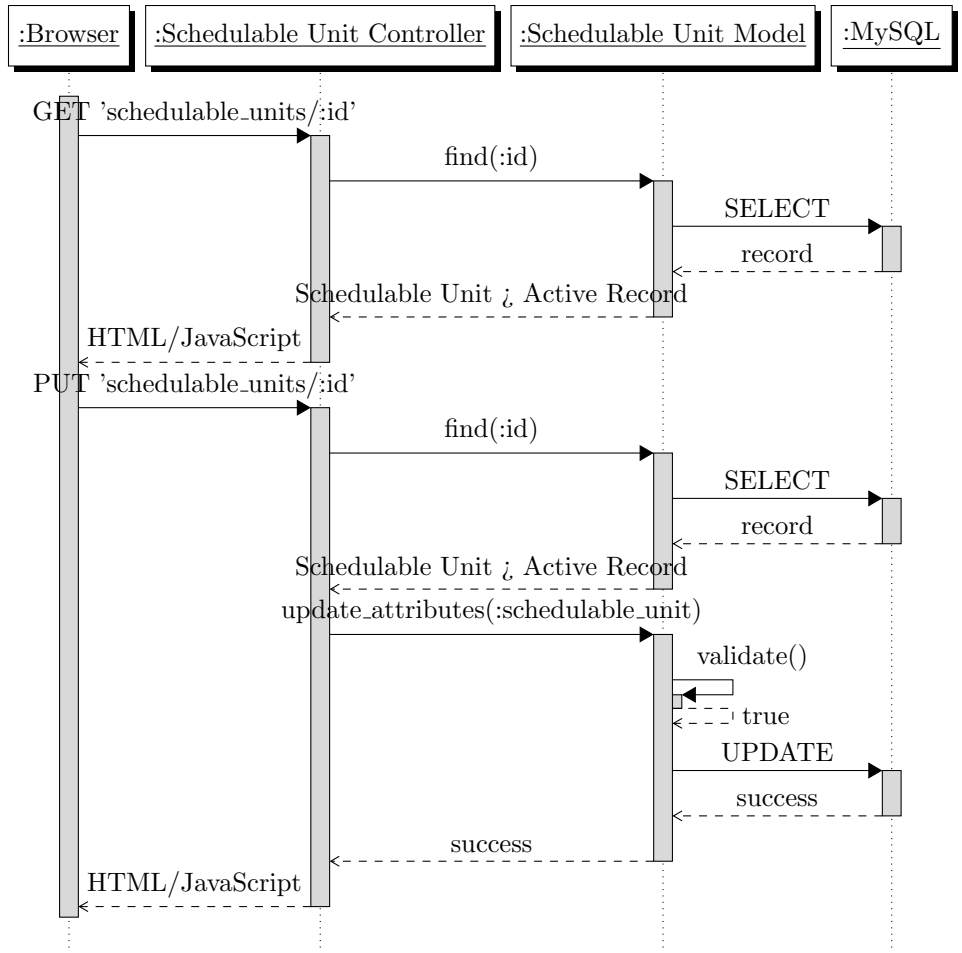
Delete Campaign

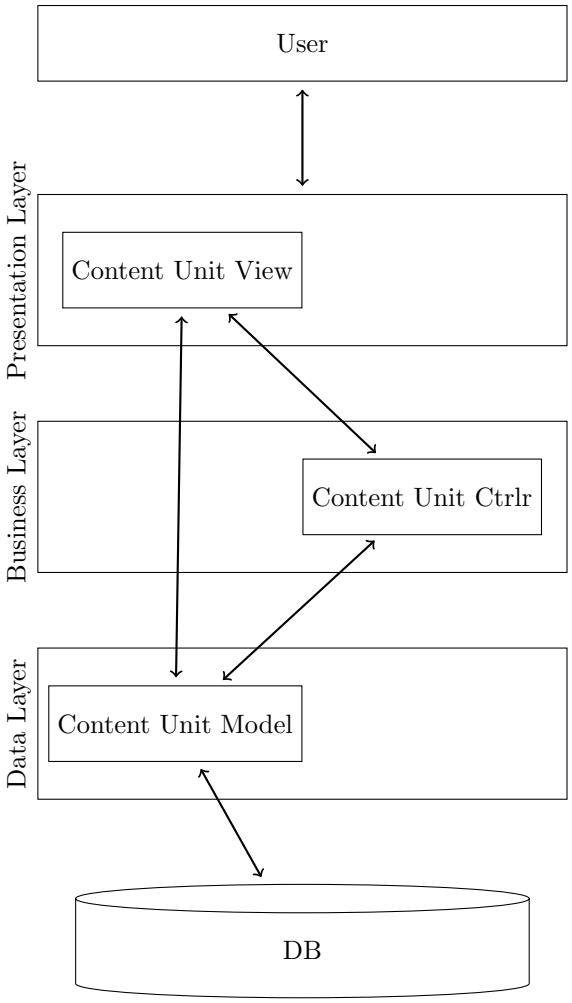
This is a basic CRUD operation: Delete Campaign.



Schedule Campaign

This is a basic CRUD operation: Schedule Campaign.

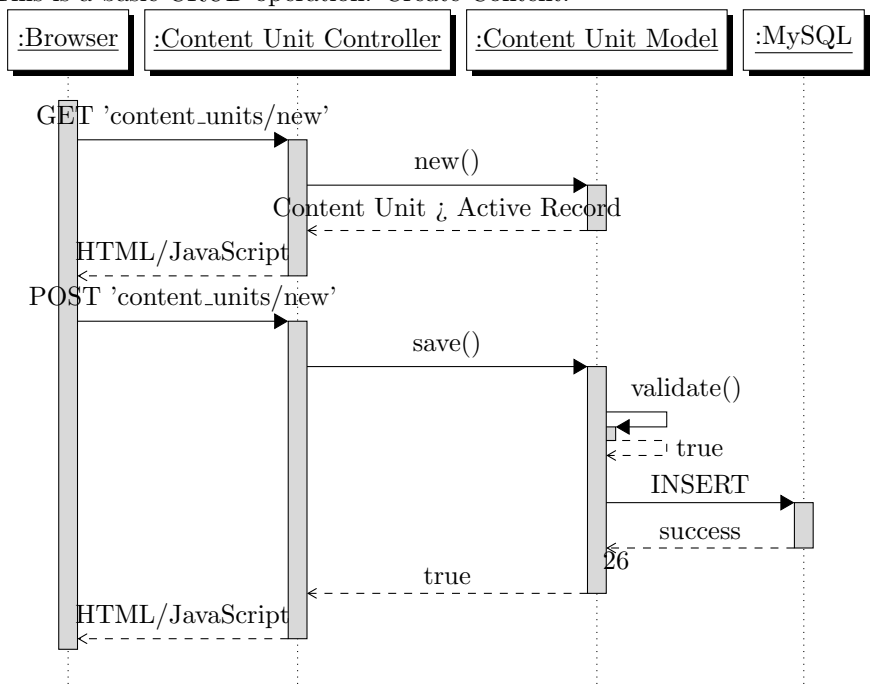

:Browser
:Schedulable Unit Controller
:Schedulable Unit Model
:MySQL
GET 'schedulable_units/:id'
find(:id)
SELECT
record
Schedulable Unit ¿ Active Record
HTML/JavaScript
PUT 'schedulable_units/:id'
find(:id)
SELECT
record
Schedulable Unit ¿ Active Record
update_attributes(:schedulable_unit)
validate()
true
UPDATE
success
success
HTML/JavaScript



2.9.3 Content Units

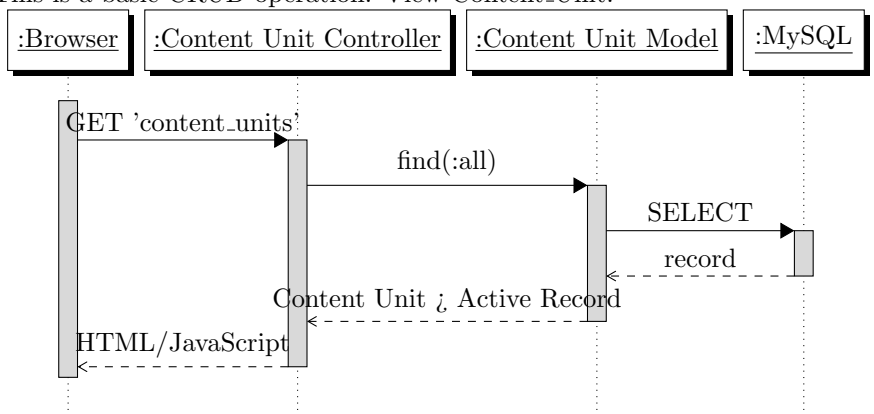
Create Content Unit

This is a basic CRUD operation: Create Content.

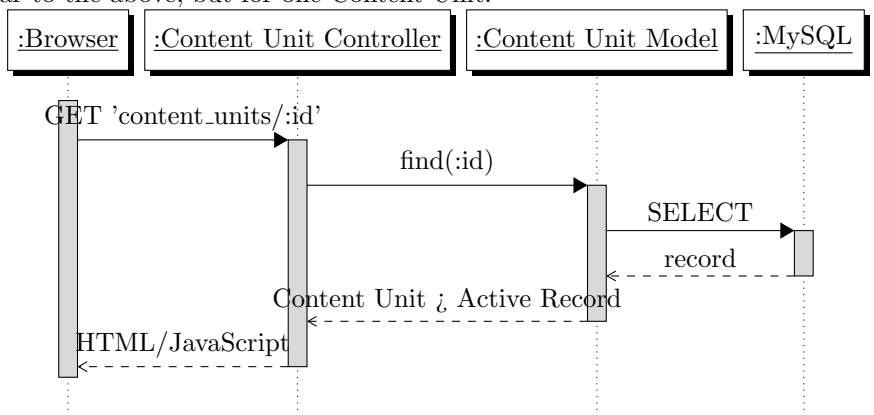


View Content Unit

This is a basic CRUD operation: View Content_Unit.

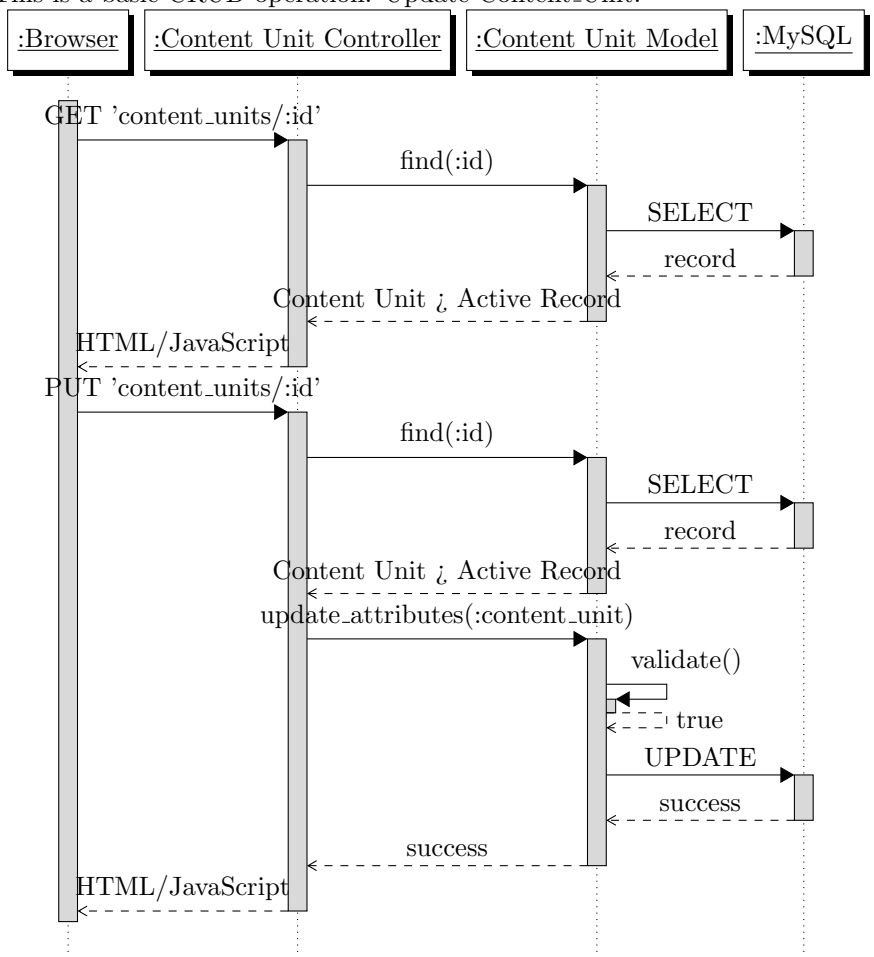


This is a Read operation, similar to the above, but for one Content Unit.



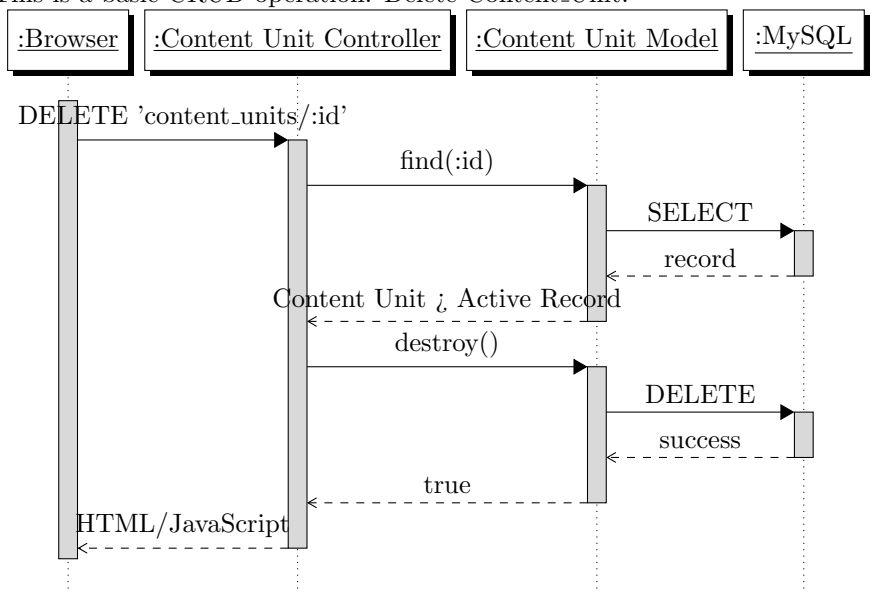
Update Content Unit

This is a basic CRUD operation: Update Content_Unit.



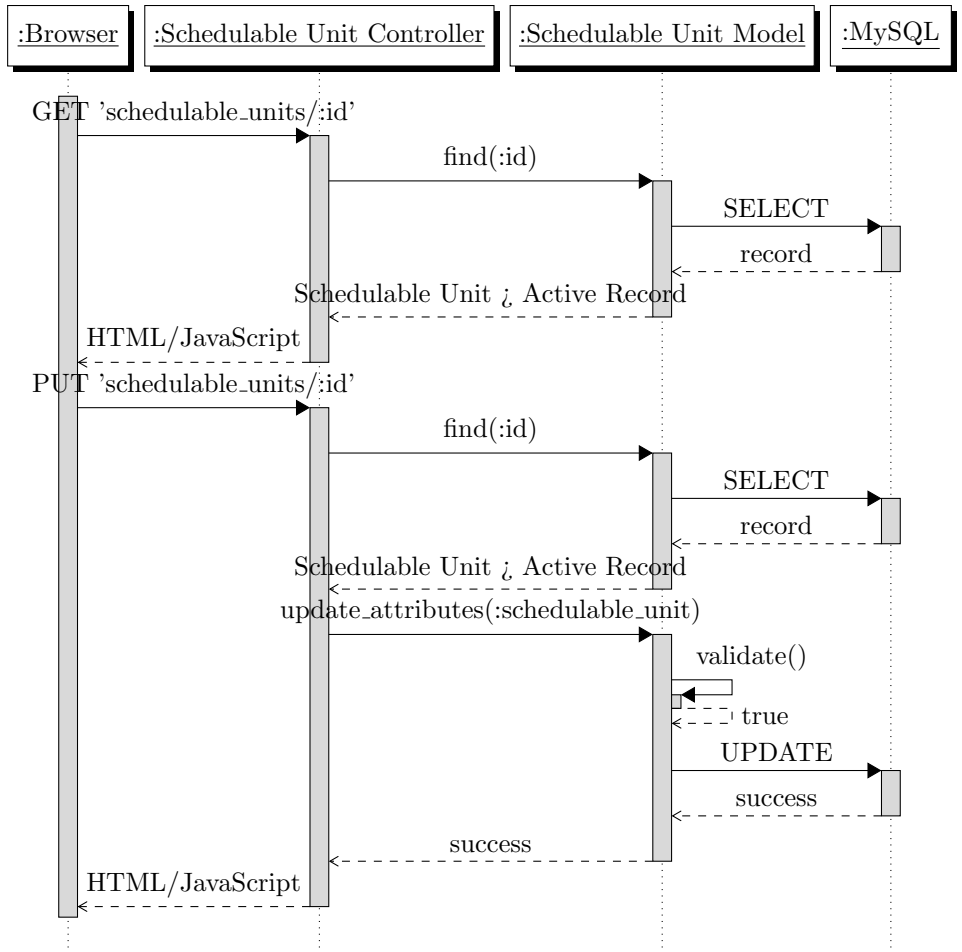
Delete Content Unit

This is a basic CRUD operation: Delete Content_Unit.

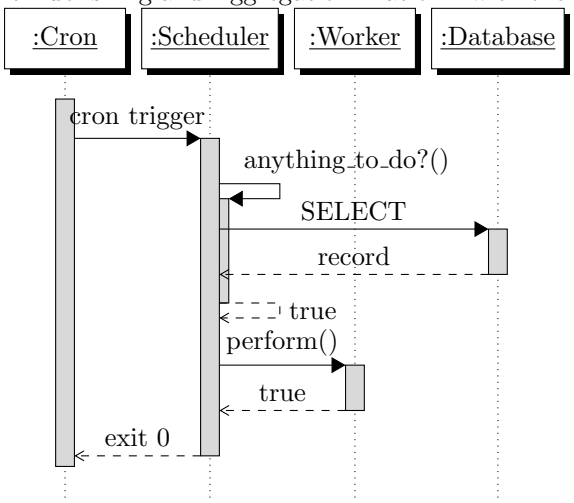


Schedule Content Unit

In order to change the scheduled time of anything that is schedulable, the same process as an update is carried out.



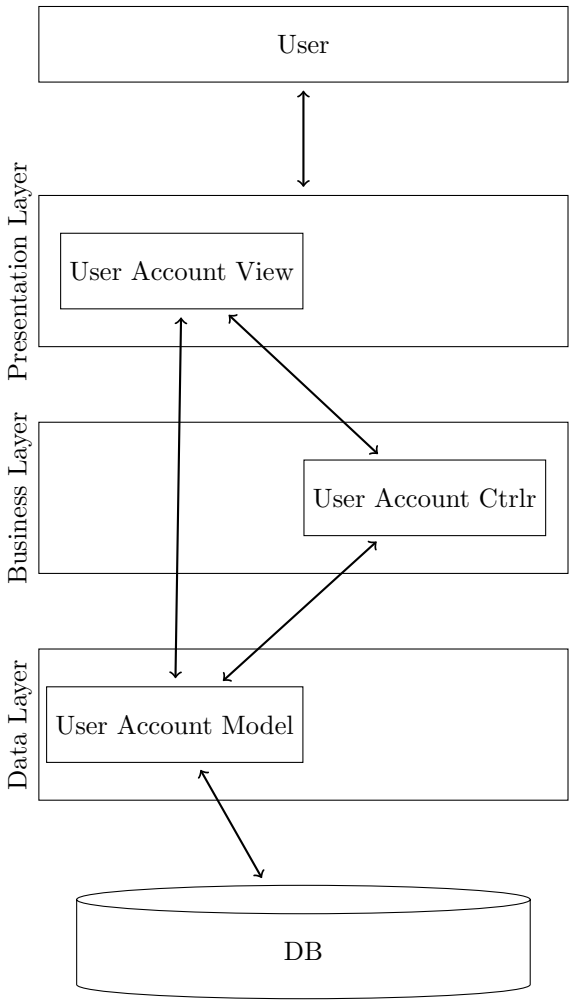
After content is scheduled, it can be picked up by the Scheduler, triggered via Cron. Every 30 minutes, Cron will run the Scheduler, which will look in the database for active Campaigns. Within those active campaigns, each piece of content will be checked to see if the go-live time is now or past. If it's now or past, the scheduler calls the Publishing and Aggregation Platform with the content needed for a push to the given

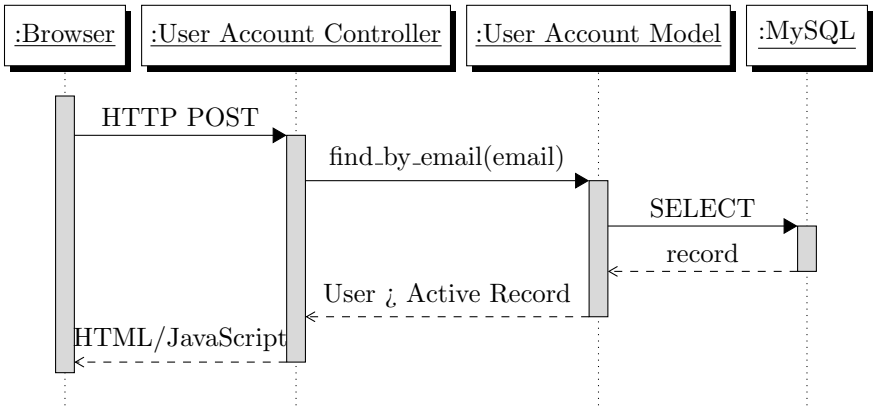


External Service.

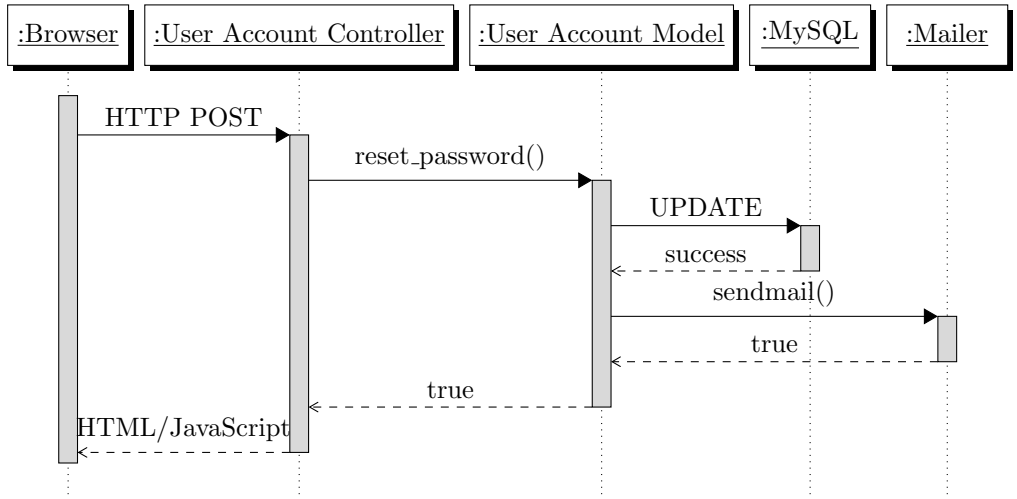
2.9.4 View Metrics and Statistics via Explore Panel

2.9.5 Lost User Name

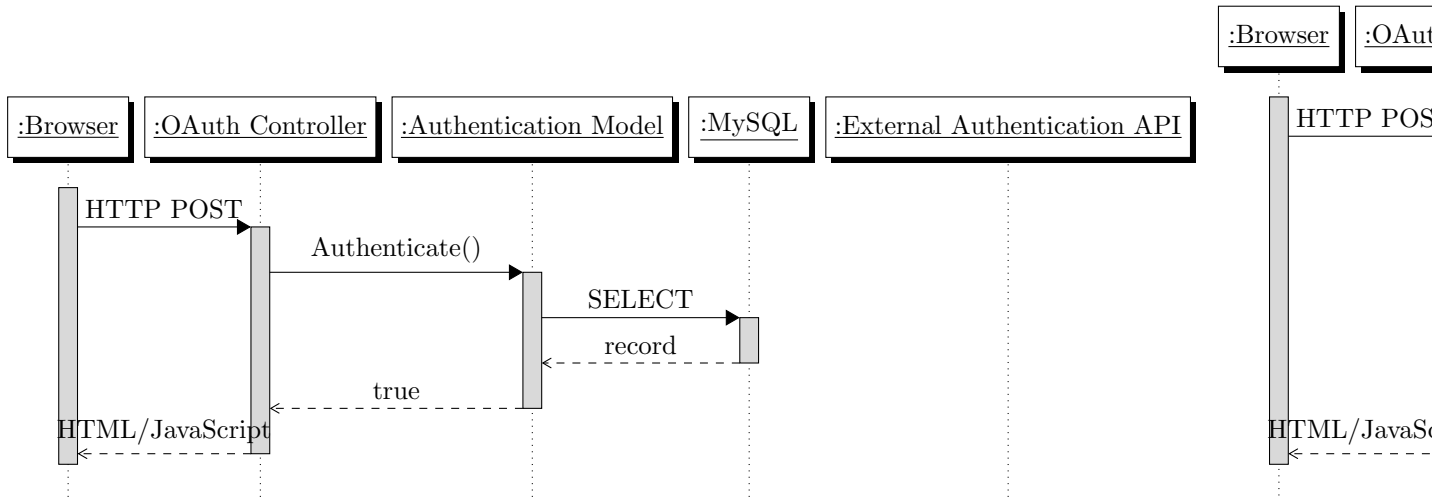




2.9.6 Lost Password



2.9.7 External Authentication via OpenID



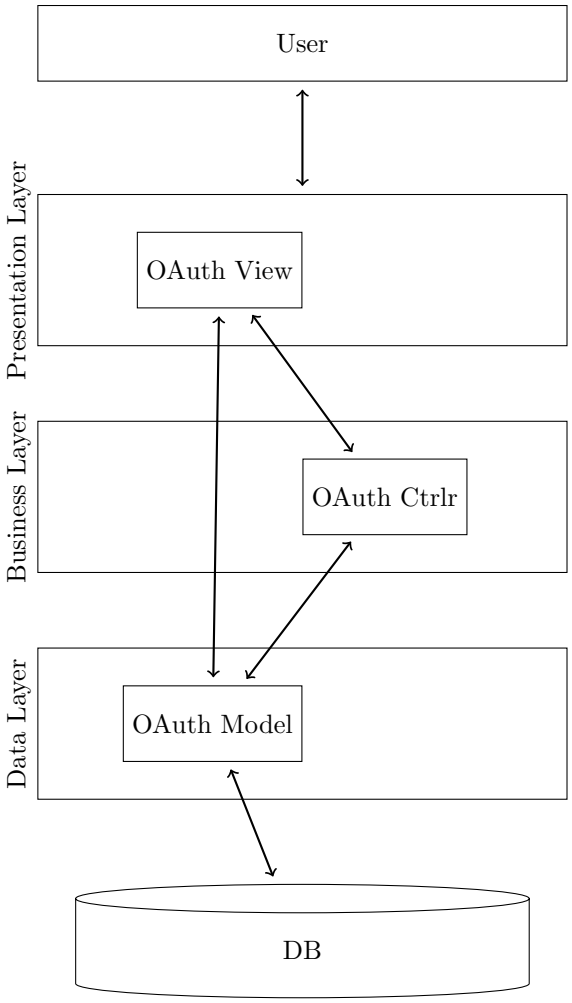
2.10 Data Layer Components

2.10.1 ActiveRecord

ActiveRecord is a library which maps database tables to objects. It allows the columns of tables to be accessed as attributes of a Ruby object. Additionally, it handles all of the relationships between tables as well as validation for input.

2.10.2 MySQL

MySQL is a fast, free relational database management system solution.

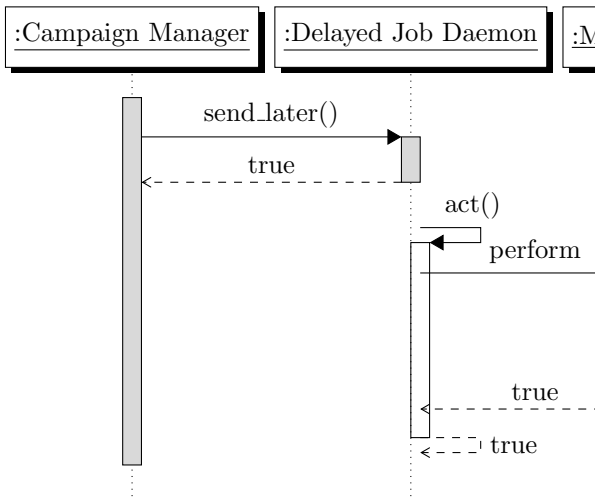


2.11 External Components

2.11.1 Publishing and Aggregation Targets

- Flickr
- Picasa
- Twitter
- Tumblr
- Wordpress
- YouTube

2.11.2 Email/SMTP Service



Email will be used to notify users of reset lost usernames and passwords.

Chapter 3

Publishing and Aggregation Platform

3.1 Object Model

3.1.1 Overview

Objects exist in Mashbot for the purposes of identification and manipulation. Objects are not stored in the Publishing and Aggregation Platform, but are merely passed through its pipeline.

Objects contain information general to that object itself and also information specific to one or more services, so as to identify that object on a service, or distinguish similar fields in different services.

Semantically speaking, an object is anything that can be manipulated by people and computers in a web context. Examples of objects are pictures, blog posts, status updates, IDs, people, et al. Not all of these are supported by Mashbot initially, but could be added through the plugin architecture.

3.1.2 The Object

The object comprises:

- Type
- Service Associations
- Properties
- *(Optionally)* Primitive value

Type

An object has type associated with it. Type is service agnostic: multiple services may support handling the same type of object, and services may support handling multiple object types.

Service Associations

Each object may be associated with one or more services. This reflects the idea that an object, as manipulated by Mashbot, exists in parallel with an object stored in some form on each associated service.

Associations tell the Publishing and Aggregation Platform where to look for an object, that is, which plugins to use to index and pull a given object.

Adding/removing an association: Adding an assocation is "pushing" an object to that service. Removing an association is "deleting" an object from that service.

These operations are defined by plugins in terms of both *behavior* and *requirements*. Requirements may be complex and hierarchical, specifying recursive requirements for values and sub-objects within a given object.

Properties

Properties represent an unordered collection of child objects belonging to the object in question. Child objects are not mandated to be of a unique type or value, and thus cannot be addressed by type or value. Instead, objects can be queried for child objects, given criteria about the child objects themselves.

For instance, an object may be associated with two different services, and thus have two different identifiers. That object, therefore, will have one child object ID associated with the first service, and a second child object ID associated with the second service. (But the numbering here is arbitrary)

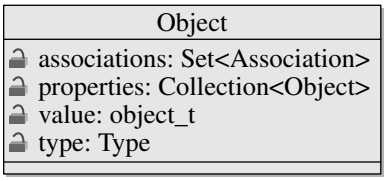
This isn't to say that child objects cannot be associated with more than one service: if two entities refer to the same thing, they should be the same object. For instance, an object that has a name might have the same name on two different services, so the object would have a single child object for name, and both objects would be associated with both services.

Primitive value

A primitive value is a value with which the object can be interchanged without losing or gaining semantic information.

For instance, IDs and names are object types that would frequently have primitive values. An ID could have a primitive value *1000*, and a name could have a primitive value *Bob's Photo*.

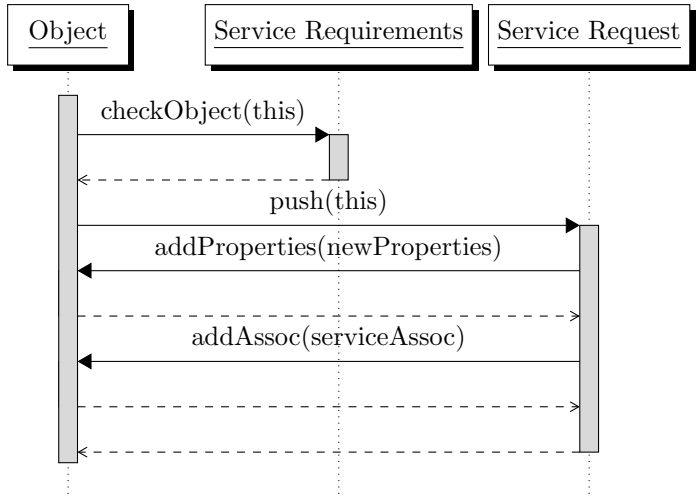
3.1.3 Class Diagram



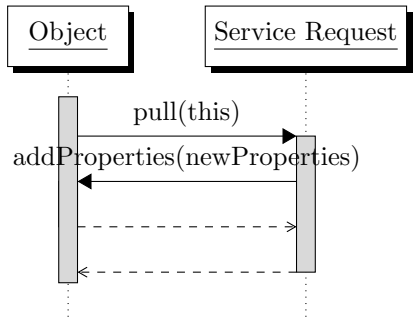
3.1.4 Sequence Diagrams

(Only showing sequences between object and service.)

Pushing to a Service



Pulling from a Service



3.2 API Design

3.2.1 Request

A request to our API will have the following fields:

- OAuth Information
  - Realm - A Protection Realm
  - Consumer Key - A key for identifying the customer
  - Access Token - A token generated using the parameters and the Consumer Secret
  - Nonce - A randomly generated string given to all requests sent with the same timestamp.
  - Timestamp - The number of seconds since January 1st, 1970.
  - Signature Method - The signature method that the user used to sign the request
  - Version - Version of OAuth that you are using.
- Operation - A string containing the operation as detailed below.
- Content - The content to be passed to the appropriate Content Processors. This contains all the information encompassed by our Object Model.
- Third Party Authentication Data - The third party authentication data for sending third party requests. It will contain a tree whose hierarchy will contain the following levels.
  1. Service Name
  2. User Name
  3. Authentication - Contains either the authentication token for this user or a password

3.2.2 Operation

The operation is the type of action which you wish to perform on our service. This partly defines what type of information is required in the request. However, these fields are not inherent and in fact, the information that is required in any request is defined by any plugin which supports a given content type. The operation space is completely unlimited but the four operations below are planned to be the main operations supported by the Mashbot project.

Push

In this operation, the user will provide all the necessary information for a piece of content to be posted to the services specified. Assuming success, they will be returned identifying information for the content on each service.

Delete

In this operation, the user will provide identifying information for items which they wish to be deleted from the services identified.

Edit

In this operation, the user will provide identifying information for already posted material and an edited version of the original content which the user wishes to be replaced on services for which information has been provided.

Pull

In this operation, the user will provide identifying information for a previously posted piece of content which will be retrieved from the specified services. In the case that the specified content is not identical between the services, each unique piece of content will be returned.

3.2.3 Content Types

The content type determines what type of content we are performing an operation on. Like operations, they partly define what information is required for a request. Also like operations, the ones mentioned below are not the only ones Mashbot will support. Any service plugin can define a new content type and determine what fields are required for a successful post. We will also allow for different ways of specifying the content. For example, a user may upload an image and use that to post a picture. However, if a service supports it the user can submit a URL pointing to a photo instead. The four main types of content we expect to deal with are the following.

Blog Post

A blog post will generally have two components, a title, and a body. Both of these will be text fields, but the assumption is that users can embed HTML if they would like more content rich posts.

Picture

A picture post's required components are expected to change a little depending on the service to which it is posted. We expect most plugins to require (or at least support) the actual image, a title, description, and album to which the image should be posted.

Status

A status is expected to be the simplest type of content. It only requires a single text field, which is the status.

Video

Video should have the same required fields as a picture, exchanging the image for a video. A service may also require one or more preview image(s).

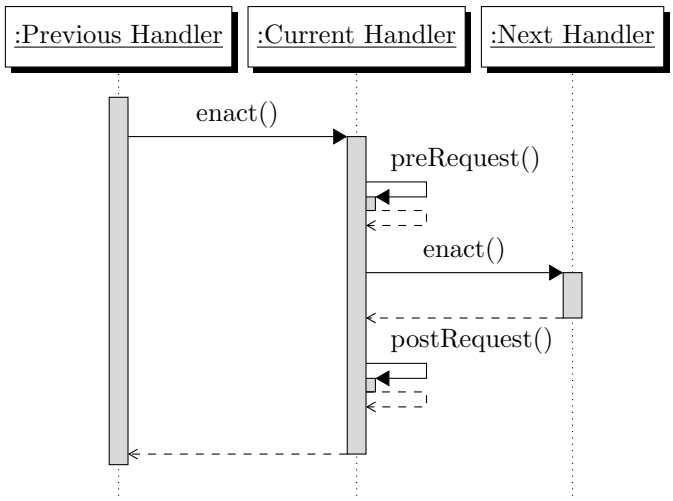
3.3 Server Design

3.3.1 Platform

We plan to build this server on an existing web service framework. This means that everything up until the point where execution and the incoming request is passed to the handler chain when a request is made will be handled by this existing framework. Most configuration will be carried in XML.

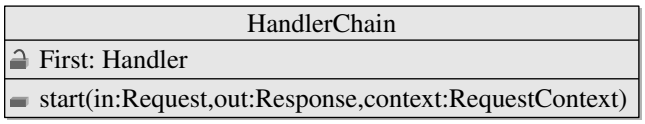
3.3.2 Architecture

The internal architecture follows the handler chain pattern. The HandlerChain class holds a linked list of handlers. As demonstrated below:



3.3.3 HandlerChain

This class embodies the linked list that is the handler chain. It will the pass the request information to the first Handler and then that handler will call each subsequent handler in the chain. This class will also handle initializing the linked list itself as it will take as an argument an arraylist of Handlers. It also unless to segregrate the functionality of linking all of the Handlers together outside of the Handler classes instead of hard coding into the code what the next handler will be for each.



3.3.4 Handlers

Handler

This is an interface which describes what methods must be provided in order for a class to be considered a handler. A Handler is a class which can take in a request, perform some preprocessing, perform the actual operations, and then perform post processing before returning have made any necessary changes to the response.

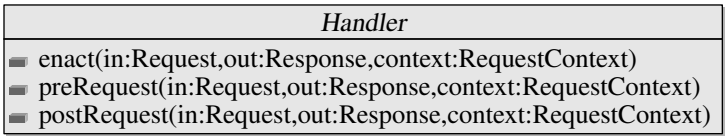


Figure 3.1: Interface: Handler

ChainableHandler

A chainable handler is one which can be placed in a sequence of handlers with the knowledge of who is next in line. This means that it will perform preprocessing and postprocessing, before and after calling the next handler in the chain respectively.

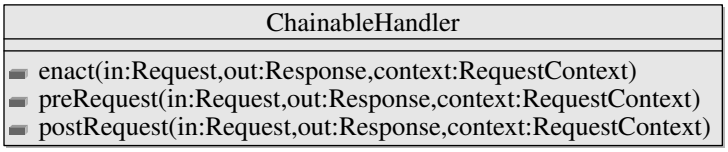


Figure 3.2: Class: ChainableHandler

AuthenticationHandler

The authentication handler will take the authentication information from the Request object and after having performed authentication of the user will insert the unique user identifier into the Context object passed up the Handler chain.

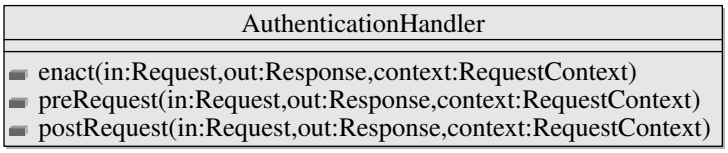


Figure 3.3: Class: AuthenticationHandler

SerializationHandler

The serialization handler extracts the raw data from the Content field and creates a new Content object from that containing all the data. The information contained in this is encompassed by the object model. See the object model section for more information. It then places this Content in the application context on its way in. On the way out, it will take the Content object from the Context. Then, it will reserialize it and place it in the Response.

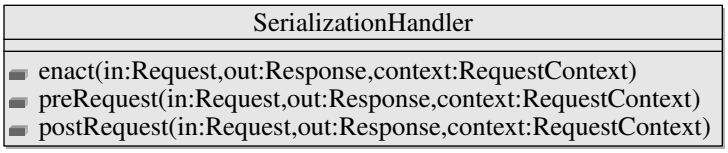


Figure 3.4: Class: SerializationHandler

RequiredDataVerificationHandler

This handler queries the plugins which handle the operation and content type which has been specified by the request with the fields that are present to make sure that all requisite data for that request is present. If all required data for any one service is not present, the request fails at this step. Otherwise, the request continues through the system.

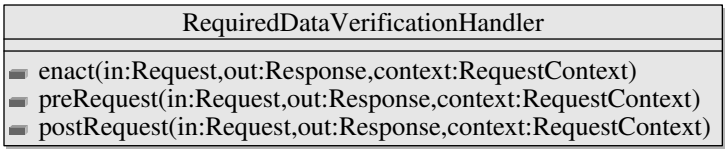


Figure 3.5: Class: RequiredDataVerificationHandler

ContentProcessingHandler

This handler loops through and calls the plugins which correspond to the operation, the content type, and the services which have been specified in the request passing them the content blob of the request. The plugin will return the blob. All plugins must conform to the ContentProcessor Interface.

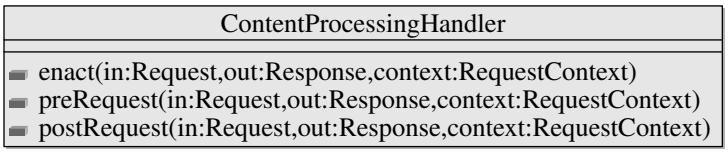


Figure 3.6: Class: ContentProcessingHandler

Overall

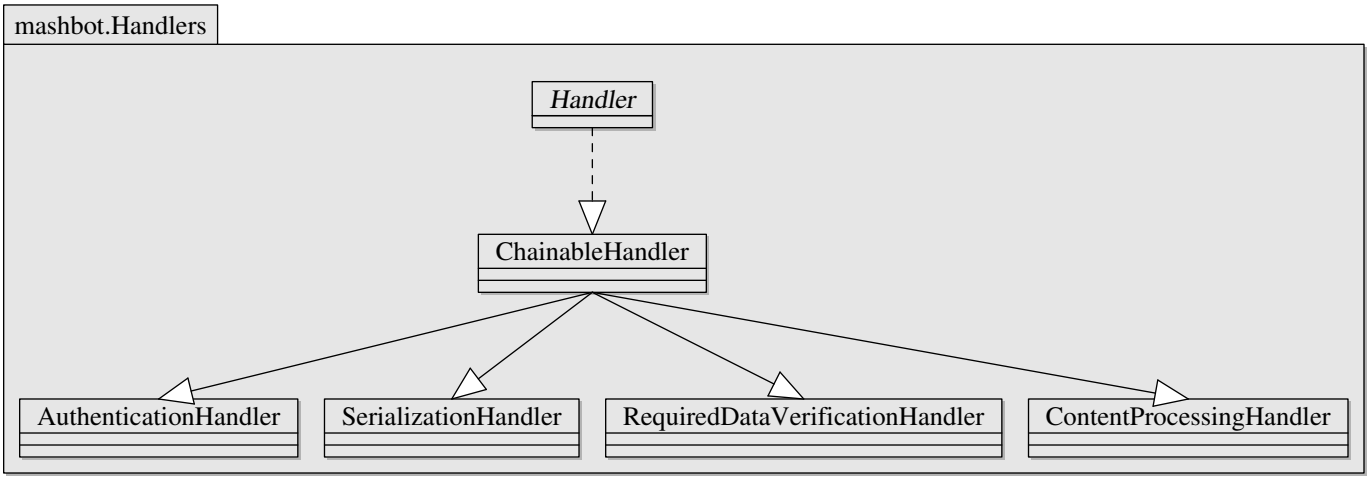


Figure 3.7: Overall

3.4 Sequence Diagram

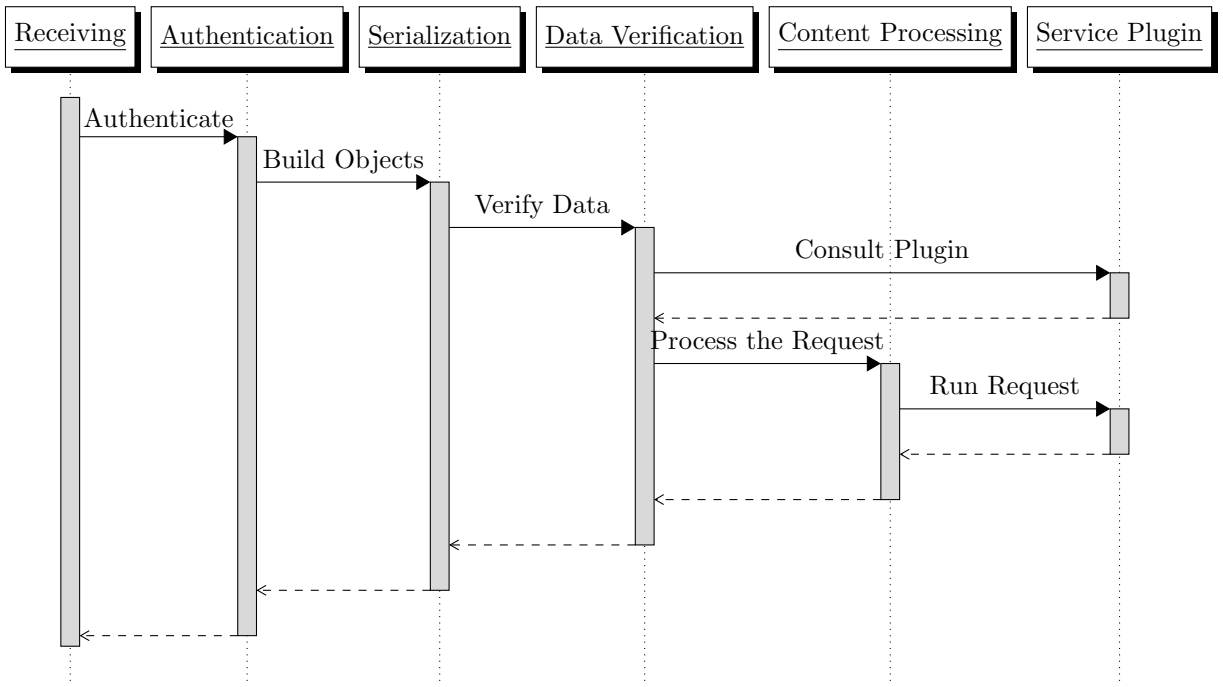


Figure 3.8: Sequence Diagram for Handling a Request

3.5 Plugin Design

3.5.1 Basic Design

The ContentProcessor expects to find a list of plugins available to it which will be categorized by service, operation, and content type. It will choose the plugins to used based on the content type of the incoming request and then pass the content, operation, and content type to the plugin. The plugin will then perform the appropriate action and return the content with the expected data filled in. The lists of available plugins will be maintained by a plugin manager will be built on top of a plugin framework.

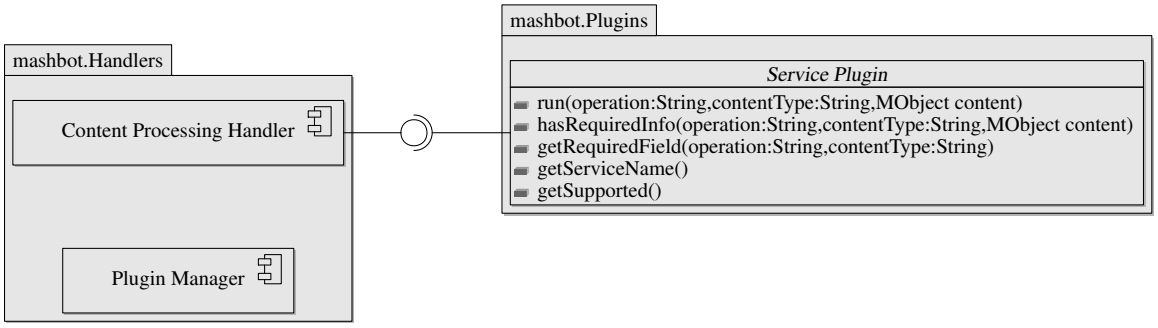


Figure 3.9: Plugin Architecture

3.5.2 ServicePlugin Interface

This is the interface that a Service Plugin provide in order to be used by the Content Processing Handler. It provides two basic functionalities. First, it must verify if the user provided by a user is adequate. Second, it must be able to be able to accept information and then perform the expected action.

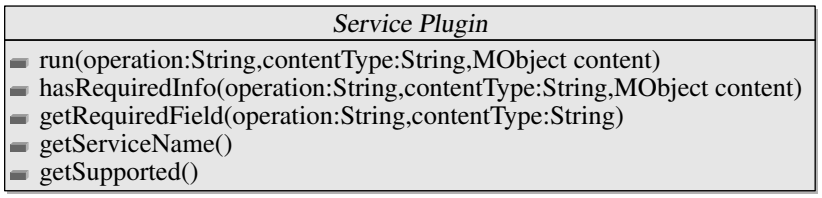


Figure 3.10: Interface: ServicePlugin

3.5.3 Service Plugins

Flickr

The only thing the Flickr API requires for a photo upload is the photo itself. Optional parameters include a title, description, tags, and access control parameters. The API also allows user to edit or delete photos and add photos to albums. We plan on providing at least these functions.

Picasa

Picasa provides a similar API to Flickr, except we must provide an explicit album for a picture to go into (although there is a default album).

Twitter

The only real object we need to deal with in twitter are tweets (status updates). Twitter allows for the posting of tweets and deletion of tweets. It also offers the ability to retweet (copy another user's tweet). We will provide access to all these actions. Twitter also provides a number of API calls which will be useful for our tools for analytics

Tumblr

Tumblr allows a variety of post types, which include "regular," "photo," "quote," "link," "conversation," "video," and "audio." Many of these post types contain fields that accept HTML. We will support all these post types. Tumblr also provides edit and delete API calls.

Wordpress and Blogger

Wordpress provides a variety of web APIs, but we will use the one which is compatible with Blogger. It allows for posts with only a title and content fields. The content field accepts HTML. It also provides the ability to edit and delete posts.

YouTube

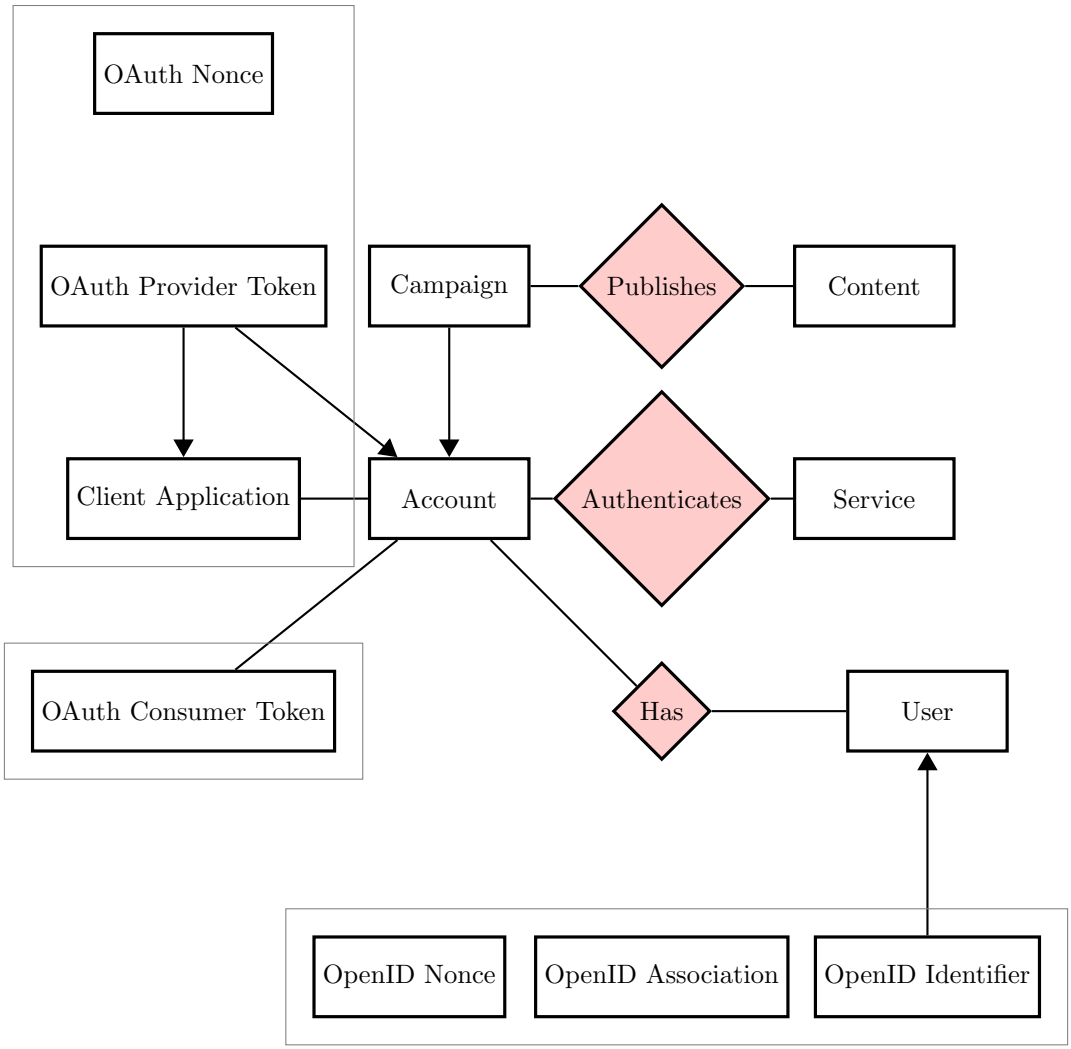
Youtube videos require a title, description, category and keywords in addition to the actual video data. YouTube supports both direct uploads and browser based uploads. Our plan is to only support browser based uploads. YouTube also supports the modification and deletion of videos, which we will support.

Chapter 4

Database Design

4.1 Summary

An entity-relationship diagram depicting the data model is shown below, which uses the E-R notation described in the book Database System Concepts, by Silberchatz et al.



4.2 Advantages of Design

- Decoupled Publishing Platform
  - Allows publishing capabilities to be extended without change to the Campaign Manager.
  - Provides asynchronous publishing so HTTP requests do not time out when pushing a large amount of content.
  - Allows scheduled items to get fired off, even if the Campaign Manager is not fully running.
- Extensibility
  - Easy to add support for new social media services
  - Easy to add support for new content types
  - Easy to add support for new operation types
- Request pipeline is easily modifiable
  - Easy to add new authentication types

  - Simple to add pre-processing steps
-

4.3 Disadvantages of Design

- Support for new content types will force additional development if generic file uploads are not sufficient to handle publishing the content.
- Generality of our design may create performance bottlenecks in the future
- We will be limited by the capabilities of the web service framework we choose.
- The campaign manager design presupposes a certain amount of familiarity with social media concepts