SE31520 Developing Internet Applications: CS Alumni Assignment

Samuel Jackson
University Of Aberystwyth
slj11@aber.ac.uk

December 6, 2014

1 Introduction

This report documents the resulting project for the SE31520 module. For this assignment I chose to create a RESTful client application to communicate with the CS Alumni site. The RESTful client is implemented in the Python language as a basic command line utility. The client uses the requests [1] to create HTTP requests to the CSA site. The command line interface has been developed using the click library [2]. The basic functionality of the client application replicates almost all of the existing functionality available through the CSA site through the command line client application. I have made very little changes to the server side of the site.

The major changes regarding the existing work is the addition of a lot more testing for both controllers, enabling SSL support and adding OAuth 2.0 [3] support via the Doorkeeper gem [4]. The client and server together implement support for the resource owner password credentials grant [5] including refresh token support. One final change to the server side is the addition of some routing aliases for the resource operations on the server for convenience.

The rest of the report is structured as follows: In section 2 briefly outlines the uses cases for both the CS site and client. Section 3 gives an overview of the architecture of the system for both the CS alumni server and the client application. Section 4 outlines my approach to testing both components of the system and section 5 provides an analysis of the project; drawing conclusions on the work produced and what could have been done in hindsight.

2 Requirements

The user stories for the CS alumni site have already been outlined in the requirements document provided as part of the assignment brief. The use cases are essentially the same for both the CS alumni site and the RESTful client application. Broadly speaking there are three different kinds of users. Guests are unregistered users who can only view the website home page and the login page. In the client application a user must login before they can perform any actions on the site. Registered uses who are not admins can perform view and update details about their account. Admins can perform all the actions of a regular user, but can additionally carry out CRUD operations on all users. Furthermore admins can also view, create and delete (records of) broadcasts to all users of the site through a variety of media. The use case diagram in figure 2.1 summarises the major actions that can be performed from via the site and the RESTful client.

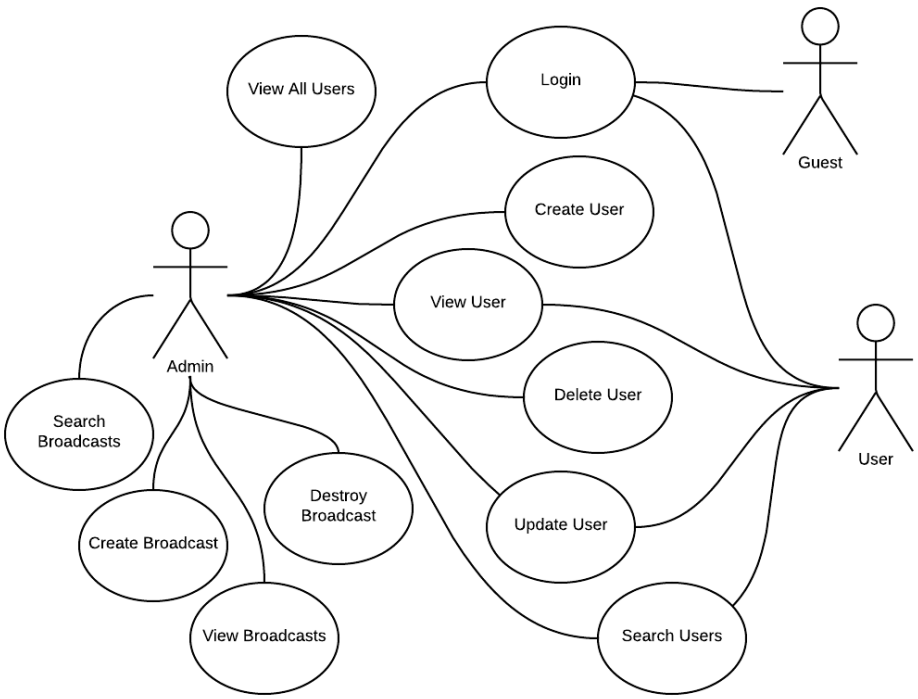

Figure 2.1: *Shows the use cases for the CS alumni site. Guests not registered in the with the site can only attempt to login. Users can login to the site and view and edit themselves. Admins can do all of these actions, but can also create users and view, edit and delete any user. Admins may also create and view broadcasts sent from the site, as well as destroy records of existing records.*

3 Design

This section outlines the design of both the existing server side application on which the client application communicates and the client application itself.

3.1 Server Architecture

The CS alumni website uses the Ruby on Rails web framework which uses a model-view-controller architecture to structure its programs. In the Rails framework the models are objects that follow the active record pattern and have a mapping to a table in the database. Controller objects provide definitions for the CRUD operations that can be performed on models with the site. Finally, the views in a Rails application are usually defined by embedded ruby files which build a dynamically created view of the application.

The CS alumni applications has multiple instances of models, views and controllers. The two main models are the Users object and the Broadcasts object. A User object defines a record for a registered site user. Every registered user also has an instance of a UserDetails model. This model is stored in a separate table in the database to the User model and handles a user's login and password details. Broadcasts are messages that can be send to users of the site by administrators. These messages as supposed to be sent out to variety of different media. Currently the only communication channels supported by Broadcasts are email and twitter.

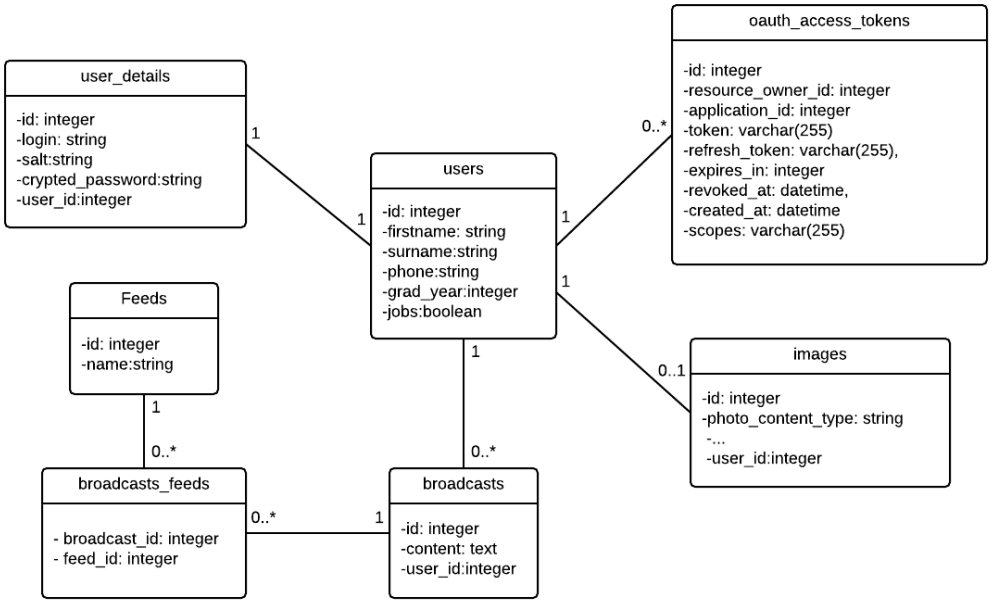

Figure 3.1: *The database design of the CS alumni site. This is based on the original design by Chris Loftus. The only addition here is an extra table for the oauth_access_tokens, linked to a specific user. This table was automatically generated by Doorkeeper. Additional tables were also auto generated, but are not currently shown.*

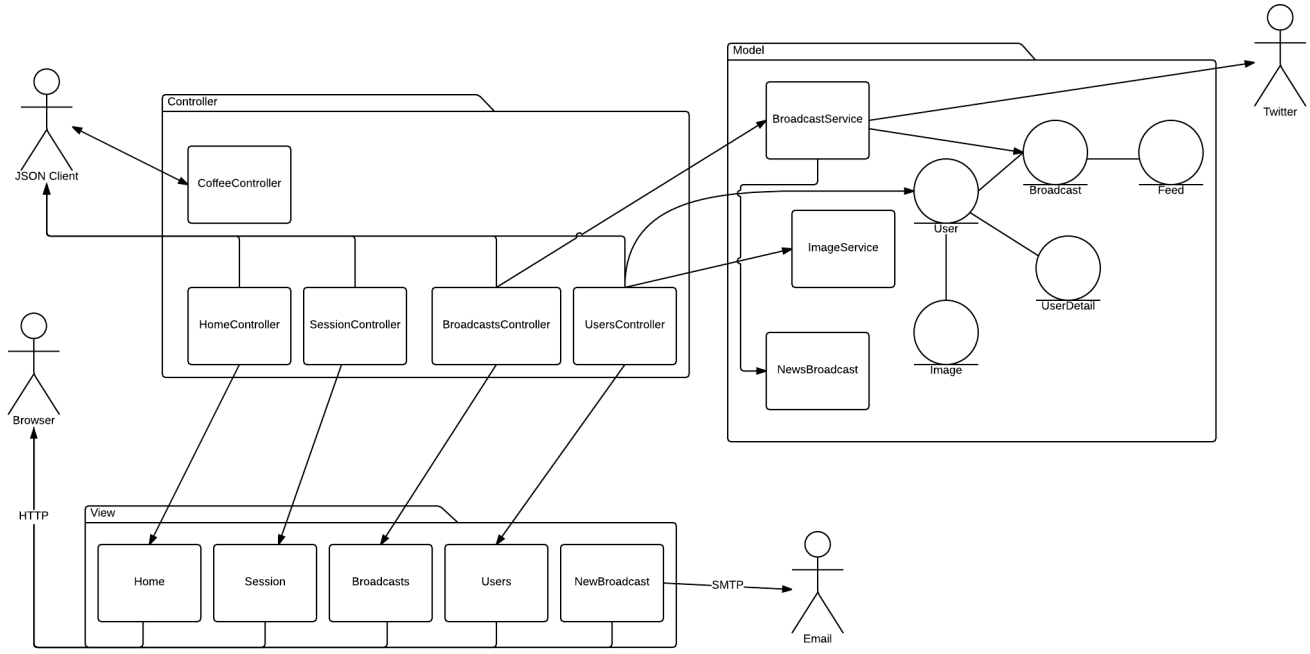

Figure 3.2: *Shows a class diagram overview of the system. This is based on the original design by Chris Loftus. The only addition to the original design is the CoffeeController for handling very basic HTCPCP requests.*

In Rails application the controllers handle the incoming HTTP requests from the web. In the CS alumni site six controller classes defined. The first is a generic application controller that extends the Rails version of an application contoller. The other controllers further extend this controller in turn. There is one controller each for the User and Broadcast objects which are responsible for performing CRUD operations on them. There is also a controller for handling the creation of a new user session when a user logs in from the "human" version of the site. There is also the (empty) home controller which currently

just shows the index page. I have also added a basic controller to add support for HTCPCP [6].

3.2 Client Architecture

The client side of the application is a lightweight wrapper to the requests library. The core class driving the design is the RequestsHandler. This handles setting up HTTP requests to one of the predefined end points of the client application. In order to add OAuth 2.0 resource owner support the RequestHandler is extended by an additional class called the OAuth2ResourceOwner. This provides additional functionality for automatically refreshing expired OAuth tokens. Additional "flows" from OAuth 2.0 could also be implemented through such a mechanism. The CsaAPI class has an instance of a RequestsHandler object and defines a collection of helper methods that make HTTP requests through the requests handler. Finally, the command line interface is defined as a collection of functions in command.py. This only utilises the CsaAPI and does not use the other classes directly.

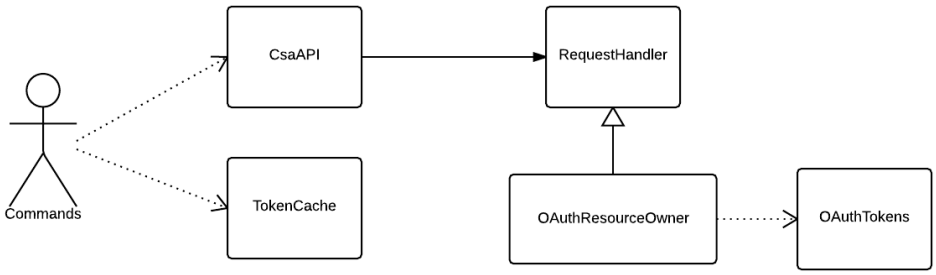

Figure 3.3: *Class diagram for the CS alumni RESTful client application. The commands actor represents the command line actions that use the CsaAPI and TokenCache classes. Commands is not represented as a class because it is just a collection of python functions.*

Running a command requires the user to be authenticated with the site. To do this they must first run the authenticate command and supply their user name and password. These details allow the server to generate an access token for the user which can be used to make further requests to secure content on the site.

After running a command the access token returned from the server is cached in a local file using the TokenCache. Internally this class uses the PickleDB library [7] to write out a JSON file. This allows the user to authenticate using their login details just once. The server also issues a refresh token with the access token which is also cached in the file. The access token expires every hour but the OAuthResourceOwner class can automatically handle refreshing the tokens provided that a refresh token is available. When a command other than authenticate is run the OAuth tokens are used. Figure 3.4 shows an example "flow" of authenticating a user then performing a command.

The CsaAPI class is designed to also function as a python package. This means that other applications or python scripts can be easily built on top of it. Because each command is executed independently of one another the tokens must be cached to file so that they are stored between runs. If the API was used a part of a different program (such as a GUI) the CsaAPI could be instantiated just once and then multiple calls could then be made without worrying about authentication and token refreshing.

The available commands for the command line application largely mimic the existing functionality and use cases of the original site, along with the same restrictions on operations imposed by the actors in the use cases.

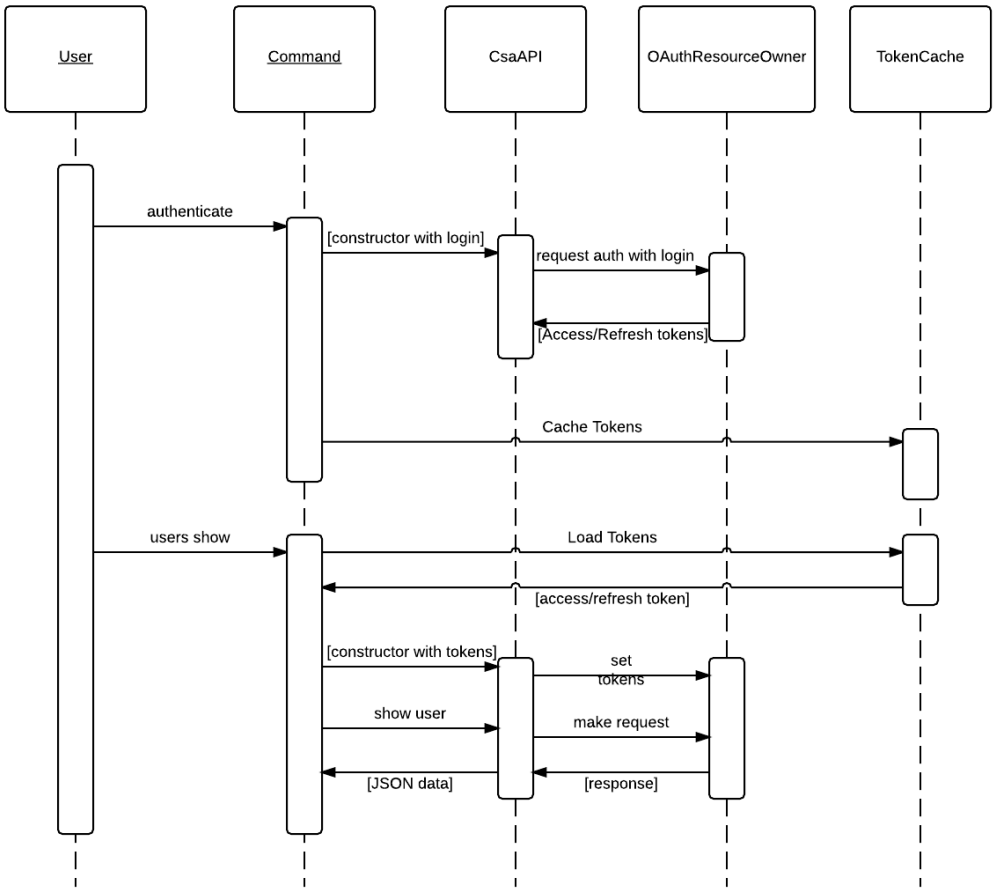

Figure 3.4: *Sequence diagram for one use case of the client. The client must first authenticate and their access tokens are cached. These tokens are then used to get access to secure data.*

4 Testing

4.1 Testing the Server

During the development of this application I attempted to follow a test driven development strategy discussed during the Agile Methodology module of the course. The first thing that I undertook during the development of this project was to bolster the test coverage of server application. I chose to do this first for threefold reasons:

1. Writing tests is a exploratory process. It enabled me to explore and understand the existing code base.
2. It provided a safe guard against breaking the server implementation in the future.
3. It ensured that there were no existing bugs in the system which could hinder future development.

I used the SimpleConv gem to evaluate the current test coverage and provide a target for the number of tests I needed to write. While coverage provides no indication of the quality of the tests running the code, it does show what is currently not being tested at all and is a good baseline of what needs to be done. Generally as a rule of thumb I tried to add both a positive and negative example for each use case, and to ensure that both human and machine versions of the site were tested.

4.2 Testing the Client

For the client application I also followed a test driven development approach to testing the application. Supplied alongside the python package I created a folder called tests which contains a test file for each class in the package. As I was starting from scratch with the client application the first thing I did was to write a set of failing tests for the CsaAPI class. As with the server side, I attempted to write both positive and negative examples for each use cases. I used the coverage plug-in for the nose test library with the python application to check the coverage level for the program and, as with the server side, use it as a baseline guide check I had enough tests. I also tried to write tests that incorporated the scope of permissions available for both admins and regular users. For example, checking that a user that tries to edit another use causes a 401 error.

In order to make it possible to run the tests in isolation from the server, I chose to use a HTTP requests mocking library called responses. This allowed me to define what the expected response from the server should be independently from the actual server. I used the concept of test "fixtures" that Rails uses in my client application by saving some JSON output from the server in local files, then loading them into test fixtures which be used as part of mocked HTTP responses. This worked well to ensure that HTTP requests were being correct made and that the responses were being processed correctly.

5 Discussion and Evaluation

As mentioned in previous sections I attempted to use some of what I learned in the Agile Methodologies module for this course as part of my process for tackling this assignment. Therefore attempted to use test driven development and refactoring instead of developing a concrete design up front. I first researched the libraries that would be most useful in the project (requests, responses, Doorkeeper, click etc.) to and made a rough plan of how each of these might be made to hang together. I then bumped up the testing of the server and wrong a tests class for the CsaAPI class of my client. I then iteratively added more functionality to my project though small pieces of development work.

The thing that worked the best during this project was thoroughly testing the existing server code base first. This meant I was not likely to break it accidentally, but also got me to read the code and understand how Rails works before blindly hacking away at the code base. It also meant that I got the boring part of development out of the way first and was free to enjoy the more creative aspects of creating the client.

While I often felt it hard to stick to at times, I found that the TDD and refactoring approach to development did work for me. Initially my client code started to suffer from the god class anti-pattern, with helper methods for accessing the API and request handing all tied up in one class. However when I started adding OAuth 2.0 support I first started by refactoring the existing code base and quickly found that something like the final design began to "fall out" of refactoring.

I also tried to adhere to the XP principle of YAGNI during development of the client side. I did consider writing the client to use a active record style pattern for data returned from the sever. However considering relative simplicity of the application I decided that this would most be overkill for the current implementation. If I was developing this further it might be relevant and useful to change the structure to follow such a pattern, but for now I resisted the temptation to use a pattern where it's not needed. Restructuring should be done as and when it's necessary.

One downside I found to following this process was the time it took to write the tests for the application. I feel that the testing I have provided in the client is not exhaustive, but that it still took me far longer to write half decent tests than it did to actually implement the system. I feel that this was mostly down to the effort of setting up appropriately mocked HTTP responses. Refactoring some of the more common request mocks into test helpers did alleviate this issue somewhat.

I also had some concerns about the JSON fixtures I was using to test the client becoming out of sync with the actual responses from the server. This was not too much of an issue as I only performed a very

small amount of development on the Rails side. However, if I had more changes I anticipate that I would have had many more problems. I think in this case I would of created some integration tests that actually make calls to a version of the server set up with a testing database and ensure that the whole process of running the integration tests is automated from start to finish.

Originally, I planned to use an external library called requests_oauthlib to handle OAuth integration with the CS alumni site on the client side. However, After spending an entire day exhausting making every possible effort to make the client and server talk to each other I eventually gave up using it. The requests_oauthlib library has only fairly recently added support for OAuth 2.0 and it appears to still be a working progress. This combined with a lack of documentation on their site meant I was unable to get it working properly. After failing to get it working I decided that the effort involved in getting the resource owner flow of OAuth 2.0 working would not be too difficult as so attempted to "roll my own" implementation by following the RFC. I still used the expandable authentication feature of the requests library to add support for my own implementation which made integrating the code the existing framework somewhat easier.

While on the subject of OAuth 2.0, in the RESTful client I have implemented a the resource owner flow which uses the users authentication details to be passed through the client to the site. This is not as secure as the standard, three legged implementation of OAuth because the client must pass their details directly to the client. However the only way to properly implement it is with the use of another server to store the application access tokens as they cannot be communicated to the user through the source code for obvious security reasons. If this were a real site I think it might be appropriate to add support for it, but as the client must install the program directly to their machine we can already assume some degree of trust, along with the fact that it was created by the CS alumni site maintainer. This also circumvented the extra effort of setting up three legged OAuth which is overkill for such a simple application.

Overall, I would award myself 70 percent for this assignment. I feel that I have demonstrated that I can implement a RESTful web service client and maintain an existing project. I feel that what I have done presents a decent effort at thoroughly testing an application which is not trivial to test. In order to aim for the higher marks I have added additional functionality in the form of very basic OAuth 2.0 support. I feel I have produced a simple and appropriate design that does not force patterns where they are not needed. However, The program is not without its flaws. I could of done something more with integration testing or implemented a three legged OAuth system. There is also no support for image uploads from the command line which would not of been to difficult to achieve if I had the time.

References

[1] A. Kenneth Reitz. Requests: HTTP for Humans, 2014. URL `http://docs.python-requests.org/en/latest/`.

[2] A. Ronacher. Click, 2014. URL `http://click.pocoo.org/3/`.

[3] Internet Engineering Task Force. OAuth 2.0, 2014. URL `https://tools.ietf.org/html/rfc6749#section-4.3`.

[4] F. E. Phillipp and T. Costa. Doorkeeper, 2014. URL `https://github.com/doorkeeper-gem`.

[5] Internet Engineering Task Force. OAuth 2.0 Resource Owner Password Credentials Grant, 2014. URL `https://tools.ietf.org/html/rfc6749#section-4.3`.

[6] Network Working Group. Hyper Text Coffee Pot Control Protocol, 2014. URL `http://tools.ietf.org/html/rfc2324`.

[7] Harrison Erd. PickleDB, 2014. URL `https://pythonhosted.org/pickleDB/`.