The Virtual Office Door | Team Conquistadoors

SOFTWARE DESIGN DOCUMENT

James Hauser, Mitchell Hewitt, Nicolas Melillo, David Snow, Tyler Tollefson

Mentor: Dr. Eck Doerry

Clients: Dr. Eck Doerry and Dr. Michael Leverington

Date: 2/5/17

Version: 2.0

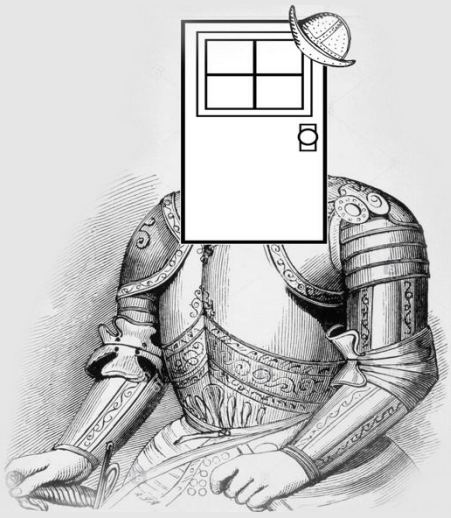

Table of Contents

1. Introduction

In many large organizations employees interact and reside in large physical office spaces that are often populated by cubicles and large areas where employees of the organization can meet and discuss ideas or issues. However, in most organizations cubicles and even office doors serve as a valuable tool for communication between workers and can help convey urgent information to coworkers and colleagues. This physical office door space is not only seen in the professional world though; it is heavily used academia as well. Office doors used by teachers at universities are often used to communicate with students about a variety of topics ranging from office hours, class announcements and even the occasional cartoon or comic strip.

This is where our client, Dr. Michael Leverington comes into play. A past teacher at the University of Nevada, Reno and a new professor at NAU. Dr. Leverington's main business is teaching classes, within that though resides a far more important business, communication with students. Through this business of communication Dr. Leverington and other teachers convey updates about students' classes, grades being posted, and other class related information. Academia is unlike most other businesses though, in 2015 NAU alone accepted 5,141 new freshman students[1]. With class numbers increasing on a yearly basis it becomes more difficult to communicate with larger groups of students, which can lead to a decline in student performance (missing due dates, class-wide messages, etc.). This is only one example of the multitude of problems that Dr. Leverington faces with the current methods of communication at his disposal.

Specifically, at NAU as well as Dr. Leverington's previous school, the major limiting factor in communicating with students just so happens to be the physical door space, or the professor's office door. A prime example of such a situation is represented in Figure 1, an office door that has a multitude of sticky notes attached to it. This kind of clutter can not only be confusing to students but also to the office door owner themselves. However, a cluttered office door is only one issue that occurs when relying on a physical space for teacher/student communications, some other issues are detailed below:

1. The door owner physically has to be at their door to post sticky notes, i.e. "Back in 15", "Office hours cancelled", etc. or the owner needs to send out a mass email to their audience to alert them.
2. There is limited space on an office door, and much like in figure 1, a door owner could have a multitude of different messages that they could need to convey to their auidence.

Figure 1: A cluttered office door

1 Source: http://news.nau.edu/nau-breaks-enrollment-records-welcomes-largest-freshman-class/#.WJtS67YrIp8

3. In order to view sticky notes, a office hours schedule, etc. it requires a person, who may be under a time crunch, to actually visit the door. This situation only worsens as soon as the person realizes the door owner is not in their office and they will have to make time to come back later.

1.1 Solution Overview

To combat the previously detailed problems our team, in conjunction with our sponsors have detailed a solution to combat communication issues in academia. Our solution is a web 2.0 application that functions as a virtual office door, which would allow the user to customize their door with widgets that can display different information to their auidence.

In addition to that our software enables working environments to communicate information that would conventionally be transmitted physically to communicate that information through the internet. This allows any person to receive the information put out by another person regardless of where the two are located on the planet.

We have also designed this application so that it complements other channels of communication rather than competes with them.
For example, our application provides basic messaging, social interactions as well as professional announcements in a way that focuses on content directed to specific known recipients and making our application's content visible to visitors of the office door whether they are known or unknown.

1.2 Requirements Overview

Once we had successfully outlined a solution we needed to generate base requirements for our application. The following list are a few of the user domain requirements for our project:

- Ease of Use
    - A door owner should be able to edit information on their door easily and in a timely manner. This also entails presenting information to a guest of the door in a clean and concise manner.
- Notification System
    - This component would take the form of a simple email communication sent to subscribed users of an office door. At the same time this same system would allow updates to be sent to users on a given time frame.
- Cloud based architecture
    - Not only should the system boast ease of use but it should also boast speed. That is why a constraint on the solution is that is is hosted in the cloud and thus can easily be deployed anywhere, and it can run at speeds not normally attainable when hosted on a school server.

This document however is not specifically about what the requirements of our solution are, but this document is here to serve the purpose of HOW we are implementing out solution. Coupled with high level architectural design sections as well as the structure of individual components, this document will convey how we are implementing our solution to the above problems. This document will also outline major project milestones and the timeline in which those milestones are expected to be complete.

2. Implementation Overview

When coming up with our implementation, our primary objective was to identify functions and packages that other people have already created for other people to use that would help streamline our entire development process. We knew that the more time we could save using publicly available packages and frameworks, the higher the likelihood that we could finish a quality website that works as desired by our clients.

The Django Web Framework will allow us to build a scalable, secure web service that will allow for faster development of the components of our application. Django includes an HTML templating system to allow multiple pages to inherit from a base HTML file, and allows conditional and iterative content display based on information passed through its Model-View-Controller system. There is also the ability to specify url patterns through Django, allowing us as the developers of our application to control what urls users can access to view specific functionalities. We will be using the Django-filter package to easily re-use filters on querysets in a controlled manner to minimize the data being passed around between different Django data queries and page renders. Using Markdown, we will be able to write HTML plaintext through python functions within our Django .py files and functions.

To speed up content updating and delivery, we will be utilizing Django REST framework to implement RESTfulness (REpresentational State Transfer) to communicate information regardless of the end-user's sate through API calls. Using such technology with Django will allow door owners to securely update their doors without reloading a web page, allowing for faster content management.

To utilize the API calls, we will be using jQuery to dynamically send calls to post and get data from the serializers. To display the content on a user's virtual door, we will be utilizing Webix, a JavaScript and styling tool, to create displays for individual door widget contents. To organize these widgets, we will be using Gridstack.js so that a door owner may organize their door widgets on a grid and the display will be consistent regardless of browser size.

To host this website, we will be using Amazon Web Services which allows a free tier of web hosting that we can deploy our website to. To add a strong layer of security, we will be

utilizing Google's login API to ensure that our website's user login is as secure as possible for our application.

These different technologies provide a significant level of abstraction to make our development of this application timely and efficient. Additionally, these tools will allow us to make the application both fast and secure so that content is delivered quickly, and the database is accessed and updated securely. Our goal is not to reinvent the wheel, but to create a good website that works properly, and the use of these different JavaScript libraries, python libraries, and online services will enable us to do so quickly.

3. Architectural Overview

To continue the process of detailing our applications architecture, we now need to go into a broad architectural overview that encompasses basically how our applications parts interface. The following diagram is the applications workflow:

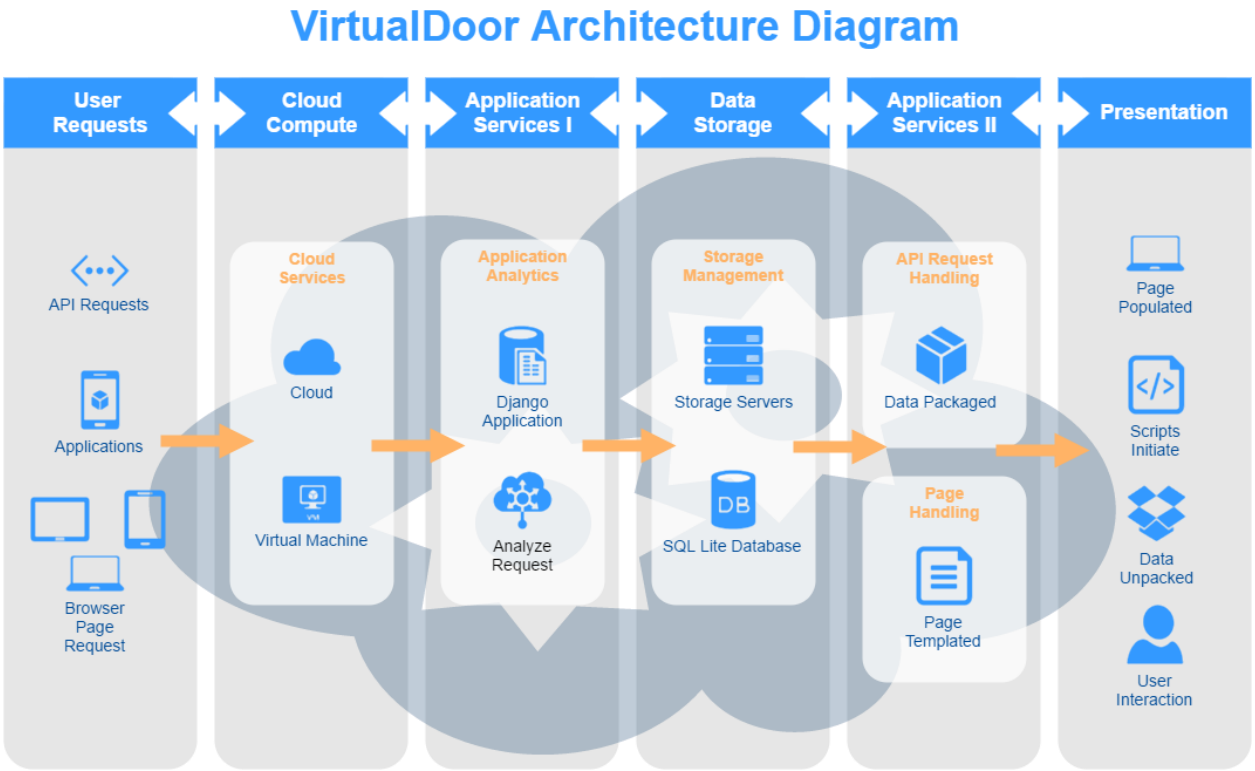

Figure 2: Architectural schematic detailing the flow of a request through the Virtual Door architecture.

Our system design closely relates to a multi-tiered client-server architecture. The flow of a request is as follows (Figure 2):

1. The user's system, via web browser, makes a request for either a webpage or a structured REST API data request [User Request].
2. Then the request is routed to the Amazon Web Services (AWS) Cloud platform where a virtual machine running our application/backend is being hosted [Cloud Compute layer].

3. The Django REST API analyzes the request and figures out what needs to be done to handle it appropriately [Application Services I].
4. The Django system then accesses data storage (for either reading or writing data based on the type of request) via a SQLite database file being stored on the virtual machine [Data Storage].
5. If the original request required the sending of specific data back to the client system, the Django system will package up this data and send it off; or if the original request was for one of the site's pages, Django will template the page with applicable information (based on user permissions or other factors) and send the page to the client [Application Services II].
6. The Presentation layer handles page population and structuring, script initiation (ex: scripts will handle AJAX calls from this layer to the application layer), data unpacking from returned server requests, and all user interaction on the client system [Presentation].

This system flows full circle, meaning after the user has requested a door page the presentation layer can then make API requests cycling the process over again. Essentially our architecture represents the conventional three-tier client-server relationship many systems use today. We decided to attempt a version of this architecture because of its well know success and modularity in modern page development.

4. Module and Interface Descriptions

From the six major components of our architecture, detailed in the previous section, we can identify three specific components of our application:

1. Presentation: jQuery, bootstrap, JavaScript, and Webix
2. Application: Django and Python
3. Data storage: SQLight

It should be noted that the Application and Data Storage are in the same section. This is because our database tables are extracted away by Django and are never directly interacted with, thus the two are grouped together. In the following sections the different components are detailed along with their functionality and setup.

4.1 Presentation

Our first major component of our architecture is the Presentation, or the customer facing application. This component has major significance in our system because it is what the user will see when visiting our webpage. The Presentation component is broken down into two sub-components, the Google Login API /User Profile System and the actual virtual office door itself, both of which are detailed in the following sections.

4.1.1 Google Login API and User Profile System

The responsibilities of the login-api is to allow door owners to login in with a secured account so they and only they can edit their own personal door. In addition to this the login api distinguishes the difference between door owners and door viewers by looking at the currently logged in account and comparing it to the owner of the door being viewed by the current account.

The login api will let users with Google/NAU accounts to create a user account on our website. Once logged into the website, the user will be asked questions such as what is their preferred name, the preferred url of their door, and similar pieces of information. Once this information is acquired from the users, the login api's secondary purpose is to then send this information to the back end Django database for storage and future reference.

The figure below shows the rough layout of how the profile aspect of the website will function. First a Django Python server will be run on a virtual machine. Whenever someone visits our site they will first arrive at the Django Python server as demonstrated in the figure. Upon logging into a google account, the Django Python server queries the Google+ API Cloud and returns with verification on whether or not the attempted logon is a valid Google account or not. If the Google+ API Cloud returns with positive verification, the python server then reroutes the user to a new HTML page to edit their profile information unique to our website. As shown by the figure, once this profile information that is unique to our website is submitted, it is sent to the Django Database backend.

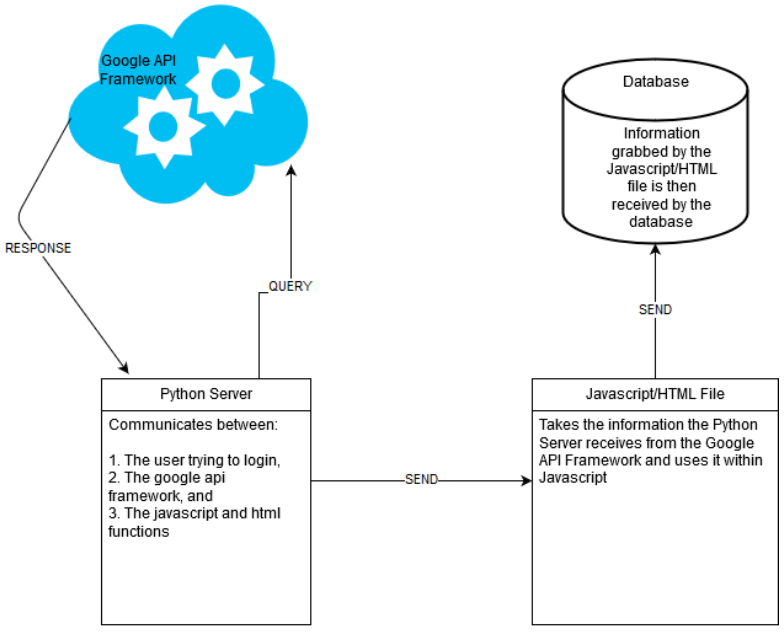

Figure 3: Login/User Profile design

From the perspective of the users of our website, the login Google+ API provides a literal window that the user can safely and securely login to a pre existing Google account. Designing around a preexisting login technology creates peace of mind for users of our website.

4.1.2 The Virtual Office Door

This specific component of our overall architectural design is the meat of the product. When the user logins in to their account they will be directed to their personal office door so they can customize it and add widgets to the page. Aside from that this component directly interfaces with the Application and Data Storage components the most, as it requires direct communication with both. The Virtual Office door is also where the complex UI elements, the widgets, reside. Each widget represents a sub-component of the Presentation component, with each having specific functionality and ways to interface with the Application component. Below is the UML diagram for this component:

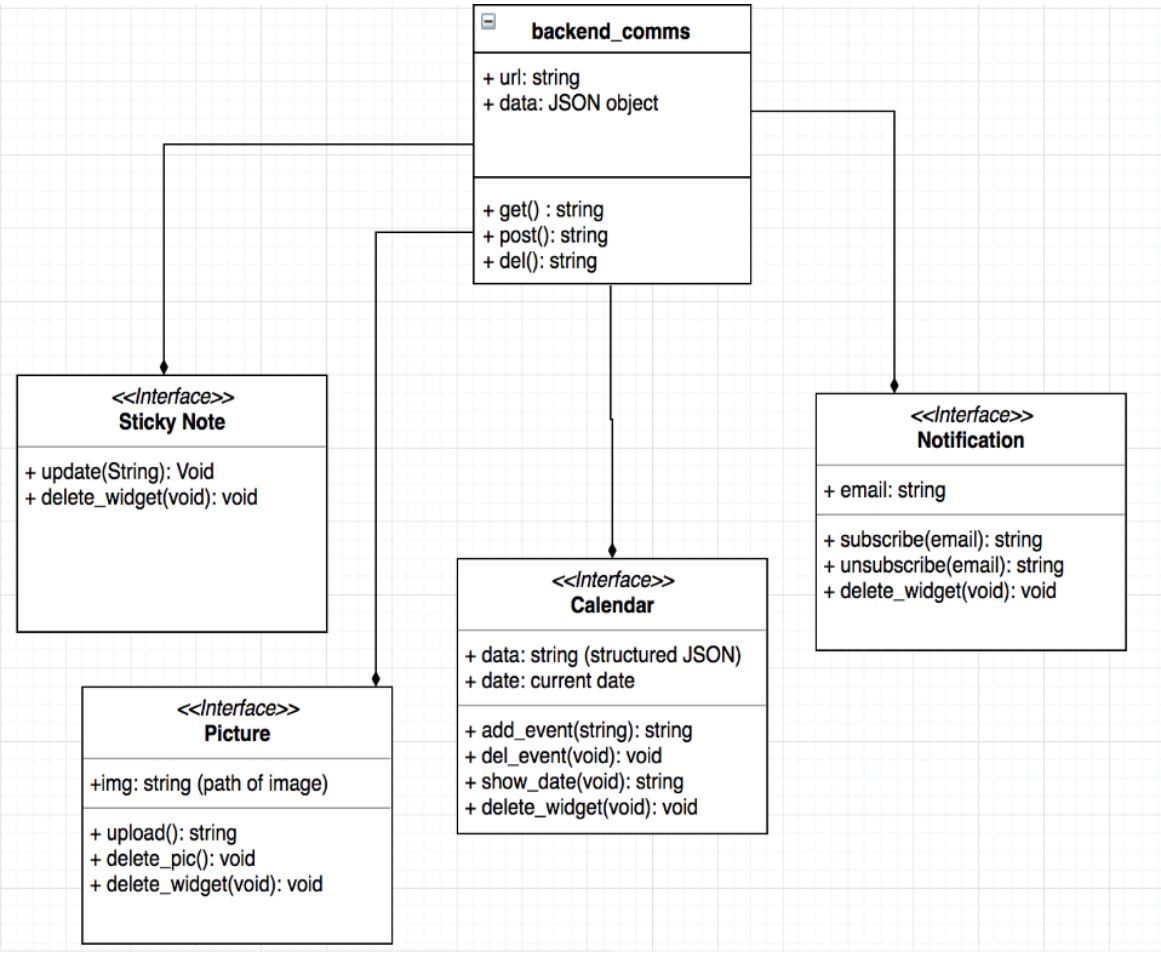

Figure 4: Virtual Door UML

For this specific component, the main public interfaces are outlined in the diagram above, to reiterate these are: Sticky Note, Picture, Calendar, and Notification. All four of these interfaces

represent what the user is going to see on their page as well as what visitors to the office door will see. We go into further detail about each specific widget below:

- Sticky Note
  Service Provided: convey short messages to visitors of the office door
  Workflow: User initializes the widget on their page, then click a button/edits a text field/etc. and the content in the read only text field is updated with the new message. The old message is replaced by the new text and cannot be recovered. Guests should only be able to see the widget contents.
- Calendar
  Service Provided: event planner and organizer
  Workflow: User initializes the widget on their page, once initialized events can be added to the widget by typing in an event and date. The events appear in a list that is easily stored in JSON format. When an event has passed, the user can remove the event.
- Picture
  Service Provided: display an uploaded picture to guests of an office door
  Workflow: User initializes the widget on their page, then the user can then upload a picture to the widget that fits within certain size constraints. The picture is displayed to guest users and does not allow guest users to upload photos.
- Notification
  Service Provided: communications between door owner and guest about door updates
  Workflow: After the user initializes the widget, it simply displays a form to the guest users that takes in an email, filters it, then sends the email to the database. Once in the database the email is used to send communications via email. Minimal setup is needed from the door owner.

The virtual door component of the Presentation is what utilizes most the technologies listed in the Presentation description. Each widget is a Webix widget that has underlying JavaScript functions implemented, this also applies to the structure of the actual door itself. The door utilizes bootstrap technologies as well as JavaScript and Gridstack (containers for the widgets) to organize whichever widgets the door owner wants. Aside from the door itself, the Google Login API and the User Profile editing utilizes JavaScript to communicate information through the Application component and into the Data Storage component. The next main component that we will discuss is the Application component, which serves at the communication between the Presentation and Data Storage component.

4.2 Application and Data Storage

In this section, we go into further detail about the structure of the Application and Data Storage components, both of which directly interface with each other. The main technologies used in these components are Django, Python and SQLite, all of which are used to facilitate storage of user data that is received from the front end. Application and Data Storage are further broken

down into 4 subsections that cover how the backend application and database work in conjunction, all of which are detailed below.

4.2.1 Models

The models are primarily a template of how the database stores data. Each defined model serves as a type of object stored in the database. These objects are necessary to interface with the database by different components of our Django application, and the models provide a means to extrapolate information from individual objects as necessary and on demand.

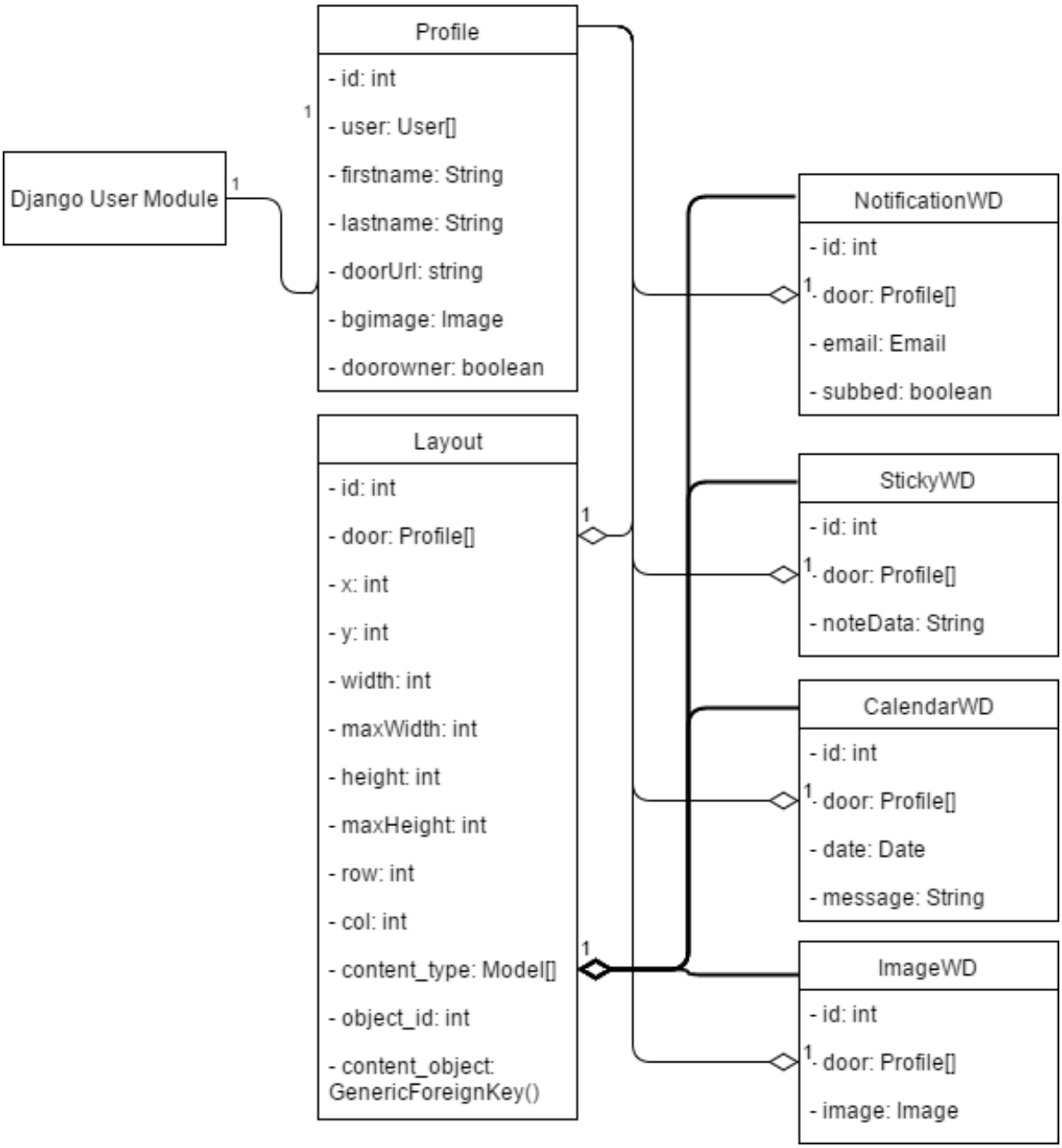

Figure 5: Model Layout

Each model in figure 5 is public, and functions inherited from the Django Models in each Model permit datasets of each model to be queried, filtered, and ordered by other modules used in the Django application. In figure 5:

- Layout model is where the information to display the location and type of corresponding widget denoted by the content type field. By using this multiple instances of this model, the location where a door owner has placed different kinds of widgets can be saved, edited, and retrieved.
- The NotificationWD model stores what door it belongs to, the email a user has entered, and a Boolean to determine whether they are subscribed for notifications on the door. This will permit sending email notifications to someone who signs up for them on a door, or not if they are unsubscribed.
- StickyWD is a basic widget that contains text specified by the door owner in the field noteData, and what door this information corresponds to: the foreign key to profile.
- ImageWD operates similarly, but instead of text, it stores the url of the image uploaded by the door owner to display on their virtual office door.
- CalendarWD is used by a door owner to store the date of an event in its own field, with a message field of what that event entails. There is also a link to the corresponding door that this record will be displayed on.
- The Profile model serves as a glue between the user's account and their door, and determines what is displayed on their door page that is not contained in a widget. It stores a reference to the profile's corresponding user, a firstname and lastname the door owner can choose, the url suffix that their door will be displayed at, a background image for their door that they can choose, and a Boolean that determines whether they own a door.

By using these models together, our implementation will offer users a large amount of control over what is displayed on their office door page.

4.2.2 Serializers

The serializers inherit from Django rest framework's model serializer, and base their keys for JSON key-value pairs off specified variables in a specified model.
Serializers specify how to format data associated with a model through specific fields of that model. Used in a corresponding view, serializers are public methods that tell a serializing view which fields to display and which fields to hide when turning a queryset into JSON or JSON into data in the database. In figure 6, the LayoutSerializer class specifies what model it is serializing (Layout), and the different fields of layout that it needs to serialize.

By using serializers, Getting and posting data is significantly trivialized to the point where creating an instance of a defined serializer and saving that reference or returning the instance's data makes the necessary selection or update in the database. The process is the same for

NotificationSerializer but for accessing the NotificationWD model, StickySerializer for StickyWD, CalendarSerializer for CalendarWD, and PictureSerializer for PictureWD.

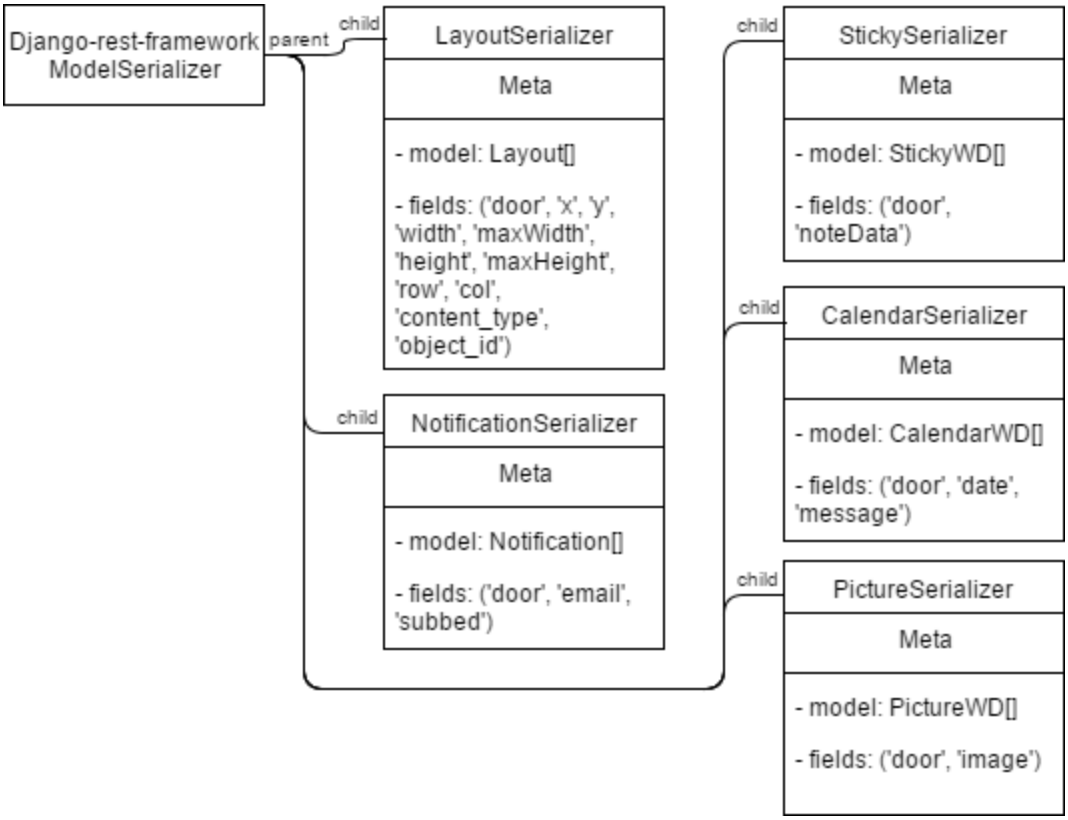

Figure 6: Serializer UML

4.2.3 Views

Views in Django are what handle specific requests made to urls. Specifically, they are the methods and classes that are called when a certain url is requested by a user. Views allow us as the developers to control what happens depending on the type of request a user makes (e.g. GET, POST) and return a response or call to render a page as we see fit.

The access of a specific url pattern specified in urls.py will call a view it is coupled with. For this reason, all views are public as they are used in the urls module of Django to display or return specific information. Our API views will handle serialization using Django REST framework's serializers, and the serializers outlined in the previous section. These API views include:

- The LayoutAPI inherits from Django REST framework's APIView, and handle specifically handles GET and POST requests. Upon a POST request, it is passed in JSON data, parses it with the LayoutSerializer, and saves the posted data into the Layout model. It returns either a success or failure depending on whether the data was valid and by the corresponding door's owner. For GET requests, it returns the Layout model entry of the page for the corresponding door owner, serialized as JSON through the LayoutSerializer.

- NotificationAPI also inherits from Django REST framework's APIView, and additionally only handles GET and POST requests. When this view receives a POST request, it takes the requester's POSTed email and checks to see if there is already an entry for them in the NotificationWD model entries for that specific door. If there is and the person is already subscribed for that door's notifications, it does nothing because nothing is necessary. If there isn't, an entry for that email data and the corresponding door is created in the NotificationWD model's entries. Upon a GET, all the NotificationWD entries for the door are returned, but only if the requester is the door's owner.
- PictureAPI inherits from the same class and handles the same request types as the above. When it receives a POST request by the door's owner, it updates the corresponding entry with the associated picture or creates a new one if none exists and returns a success or failure if the uploaded file was valid. When it receives a GET request, it returns the image data for the associated door in the associated door's PictureWD entry.
- StickyAPI behaves the same as the PictureAPI view, but passes around text instead of an image.
- CalendarAPI has the same inheritance and request types as all the above views. When a POST request is made to this view, a new entry for the associated door is made in the CalendarWD entries with the date and message passed in through the CalendarSerializer, but only if the associated door's owner made the request. When this view receives a GET request, it retrieves, serialized, and returns all CalendwarWD entries beyond the current date for the associated door.

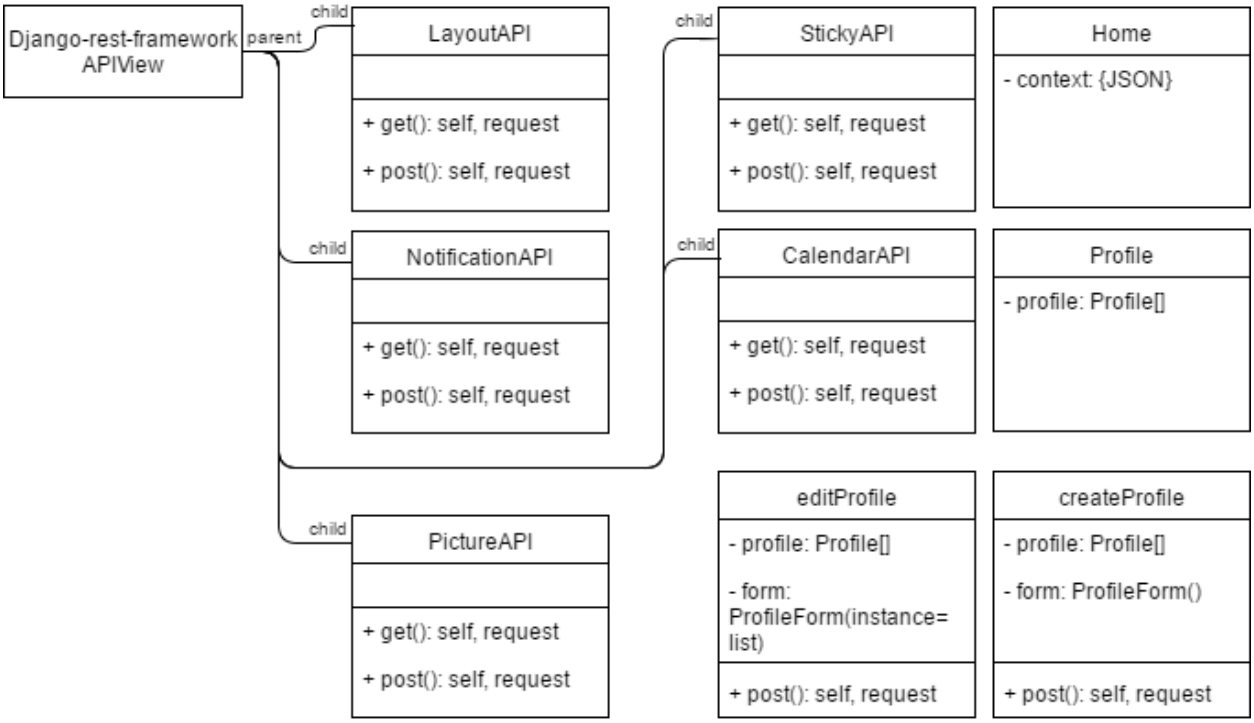

Figure 7: Django framework diagram

Additionally, there are more views to display the web pages' profile and home, as well as the createprofile and editprofile pages.

- The home view renders the home page of the website, which has a link to login. The context is JSON data that we pass into the page as it is being rendered.
- The profile view renders the profile html page of the website, and its relevant profile information for the user (that is logged in) is passed in through as JSON.
- The createprofile view renders the profile creation page, and is passed in a Django form that the same view handles if the request made to the view is a POST. If this request is a POST, it takes all the posted data and creates a profile entry in the database for the user with all of the input fields, then redirects the user to the profile view's page.
- The editprofile view renders similar content to the createprofile view, but the form passed in has the default values of the user's profile information. When this view is POSTed to, it updates the corresponding Profile model entry in the database.

These views provide the functionality of both our APIs for use in RESTfulness as well as displaying each page, allowing data to be passed into the views and the database to be updated.

4.2.4 Urls

We can specify what urls perform what functionality (through a class/method in views) by setting up an url pattern stored in a list in urls.py. An url pattern can either perform just that function, or it can send data (e.g. A primary key) through the url to the view class or method it is calling. This allows us to have significant control over what we show users where, and how we can control what we show them. In our application, this is done through these url patterns calling the corresponding views:

- "^$", views.home
- "^profile/$", views.profile
- "^profile/create$", views.createprofile
- "^profile/edit$", views.editprofile
- "^api/layout$", views.LayoutAPI
- "^api/notification$", views.NotificationAPI
- "^api/sticky$", views.StickyAPI
- "^api/calendar$", views.CalendarAPI
- "^api/picture$", views.PictureAPI
- "^soc/"– access to the social_django package which handles information routed between the django application and google's login API, allowing for Google authentication

5. Implementation Plan

After we had laid out the foundation of our architecture we began creating milestones that culminated to all the functionality of our application being complete roughly by mid-March. The following Gantt chart outlines our development cycle for this semester:

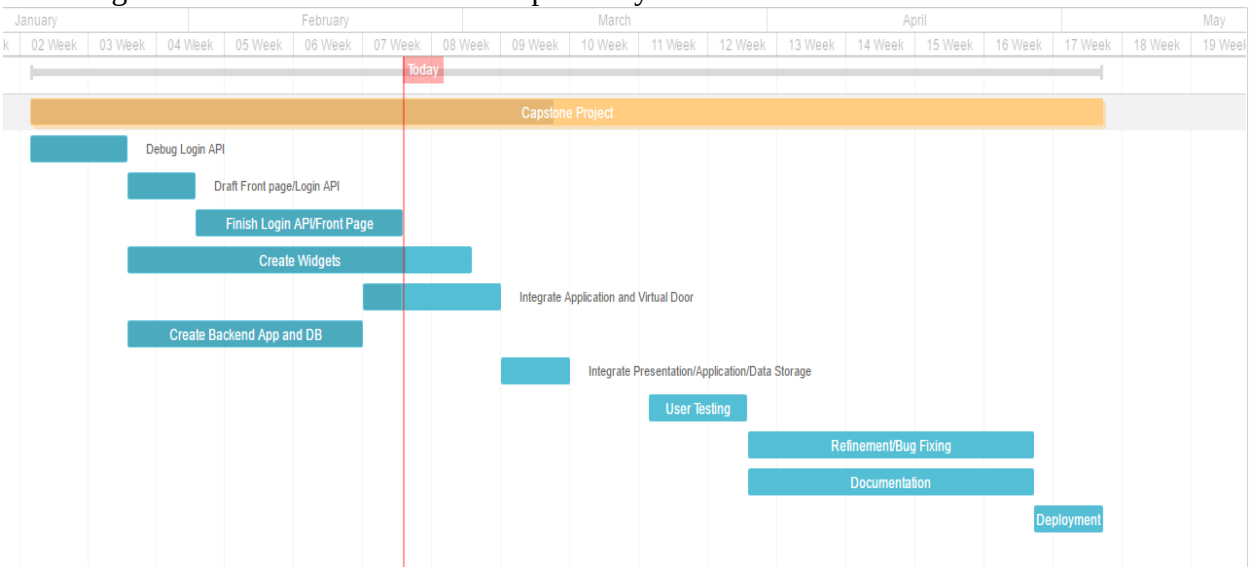

Figure 8: Current Dev Gantt

For our main development cycle this semester, which includes everything between the 2nd week and 10th week of development, we are on track and making constant progress. We have successfully finalized the main functionality of the login API and front page as well as creating a functional backend application and setting up a database.

As of today, the widgets are still undergoing development, while at the same time research into how the Login API/front page is being completed (not shown on Gantt). In addition to that we are in the process of integrating the Application component with the Virtual Door part of the Presentation component. This consists of making sure the virtual door is ready for templating with Django, and verifying that widgets can send and receive data from the Application/DB. Once we are certain that milestone is achievable we are then going to integrate all components of our architecture with each other to make a working demo by week 10 in March. After week 10 there is a gap because of spring break, but as soon as we return from that we plan to dive into user testing immediately. User testing will only be conducted for about a week, which after gaining feedback, we will do bug fixing and refinement. That section, overlapping with documentation, consists of taking user feedback and improving upon our current design, and implementing any lower priority features.

Then come the beginning of May we will be prepping for release of the application, which will be a stable version of our vision. The Gantt chart does also consider slippage during the development time, hence why bug fixing and refinement is a larger segment. Our main goal is to

have a stable and functional release come the 1st week of March; however, if need be we will still be on schedule if development is pushed back into week 10.

6. Conclusion

In academia, professors rely on their office door to communicate sometimes urgent messages to their students, such as “Back in 15 minutes”, “Office hours cancelled”, etc. These messages can sometimes be lost in translation or even go unnoticed for extended periods of time, which leads to a waste of time and loss of productivity. In our current virtual world, there is no application that serves as a medium of virtual communication between the owner of an office door and persons that wish to view the door. The main problem that our clients are specifically facing is: time is literally money. When someone posts urgent or critical messages on an office door these usually get overlooked, or are not seen at all. This leads to a lack of communication and organization which then leads to time being lost trying to find a fellow professional even if they are just away for 5 minutes. Our solution to this problem is to create a web application that serves as a virtual office door, where a user can post quick notes, their calendar, or whatever else they deem necessary.

To implement such a solution, we are utilizing a multitude of new technologies, ranging from Django and Python for the backend application to JavaScript and several libraries for the frontend UI. Taking those technologies, we are going to develop a web 2.0 application that contains the following features:

- A configurable virtual office door that boasts ease of use and speed
- Customizable widgets that allow for effective communication between door owners and guests
- Simple notifications, for guest users who opt in, that alert a guest on door updates
- A user friendly and clean UI that will promote effective communication
- A backend application that facilitates fast loading times and reliable data

Some challenges we might run in to when implementing such an application can range from time constraints to the limits of our current technologies. Specifically, if development is not completed by the end of March it will be significantly more challenging to deploy a ready to use application. In addition to that, if one of our current technologies turns out to have been the wrong choice, that can set our development back weeks to find a replacement. This also leads into the concern that our application might not even see maintenance and use after deployment; however, these are all concerns that we will address when and if they arise.

The Virtual Office door is on track with development and should have a functional application created by the beginning of March. Once the functional application is created we can focus our efforts on refining the application and implementing user suggestions. Team Conquistadoors is very optimistic that the project will be completed and ready for deployment by the beginning of May as scheduled.