The Virtual Office Door | Team Conquistadoors

FINAL (AS-BUILT) PROJECT REPORT

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

Mentor: Dr. Eck Doerry

Clients: Dr. Eck Doerry and Dr. Michael Leverington

Date: 5/9/17

Version: 1.0

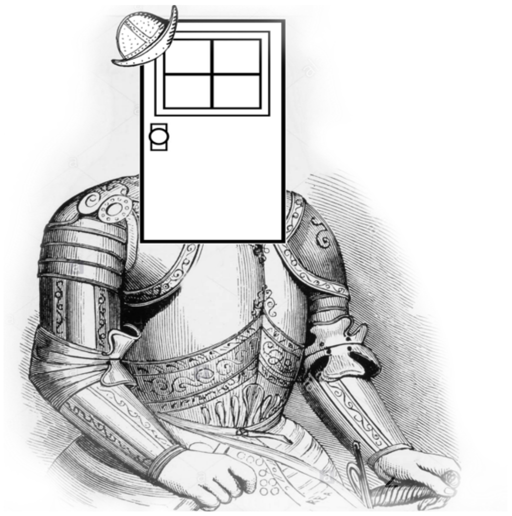


Table of Contents

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. Some other main 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.

[1]

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.

Our envisioned solution to tackle the outlined problems started off very simply, like in figure 1, and evolved into a full-fledged web application. Very simply, we wanted to build an application that not only boasted ease of use, but also boasted speed and efficiency. We started by envisioning an application that would be able to support multiple "widgets" that could be used to store door owner data and allowed for an easy to use application.

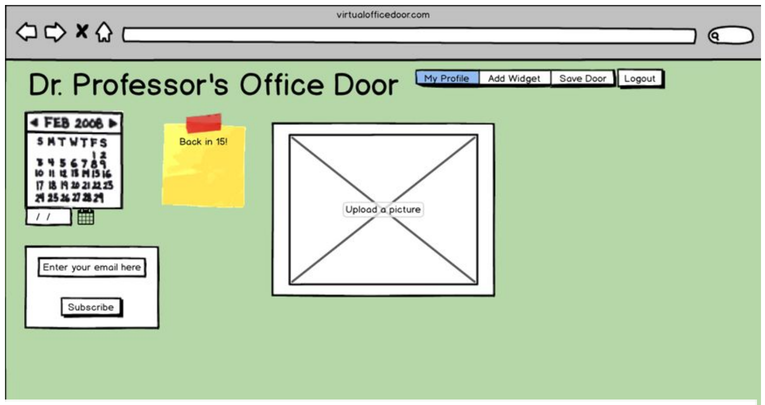


Figure : A virtual door mockup

This vision just so happened to take the form of a Web 2.0 application, hosted on some cloud platform. In addition to that our solution would utilize widgets to effectively communicate professor provided information to their wide range of students that span across all of their classes. Our Virtual Office door would serve as the new tool that teachers would use to communicate with their students in a timely and clean manner. What follows is the "story" of our application: our development process, the main requirements of the software, the architecture of the software, our testing process, and then the future of the project.

2. Process Overview

For our project we needed a simple way to organize our process as well as a sort of structure to fulfill our development goals, this fell to the typical software development lifecycle and a pseudo agile development process. Within our implementation of the SDLC we took on a majority of the major tasks that reside in it. The fall semester was dedicated to determining the main requirements for the project as well as the technologies we would be using. At the end of that semester we had to pull everything together to start the development process. In order to keep track of the team and the project we had to use several tools to assist us, and the team leader.

In order to keep everyone on track and aware of their assigned tasks we did this through a few avenues. First, for communication we utilized Discord messaging services, text and email messages. Text and email messages were not used as frequently as Discord, but they served as a medium of communication for urgent messages that needed to reach individuals immediately. Discord served as the main hub for all intra-team communications, it was also the one communication channel that we all happened to know the best and have the easiest access to.

To keep track of all of the team tasks we did not utilize any tools that required excessive management, so the main tool we used were weekly task reports done in Microsoft word. These task reports essentially served as the tool to keep track of the progress on the project as well as a tool to outline what needed to be completed each week so that we would not fall behind. Within these task reports we would outline what was completed in the previous week, what was slated for completion in the current week, and what was designated to be completed in the future. In addition to that we kept a specific Gantt chart for the status of the project, which changed several times during the semester. It was used as a sort of time piece for the project to remind us when deadlines needed to be completed. In regards to managing our codebase we utilized a bitbucket repository that one of our team members created. Through this we maintained several different versions of our software, we had two working branches (this eventually merged into one), then a master branch which was only used to deploy the application to our cloud services. This system worked perfectly as it allowed the developers a place to back up the codebase as well as a tool to track who was contributing to the project the most.

As far as our team itself goes, there wasn't any specifically defined roles. This worked out perfectly because we felt that, if we did define certain roles, aside from team lead, that some might feel that they were being cheated out of a certain aspect of the project. Which is what lead to us really just being a team of developers that floated around to different parts of the project when needed. For consistency's sake there was at least one dedicated front end and back end developer, but for a majority of the project everyone pitched in on both sides of the project at any given time. This system seemed to work the best for us, which really made the project flow and contributed to how we were able to come up with the requirements and design the entire web application.

3. Requirements

During the very beginning part of the Virtual Office Door project, we didn't have the slightest idea as to what the client really wanted the application to be. At first we all had different solutions envisioned, which led to several meetings between our team, Dr. Leverington (our client) and Dr. Doerry (our mentor/client). These meetings consisted of presenting the requirements that we drafted up for the project, having them either accepted or rejected, and then iteratively refining them. Below are the main User Domain level requirements that we had drafted up:

UD1. Website login must be secure and reliable.
UD2. Allows for notifications from the owner of an "office door" to users who opt in
UD3. User needs to be able to create a customizable "office door".
UD4. Support multiple "apps" that are editable by the user.
UD5. A touchscreen door display that can be placed on an office door that shows the specific user's virtual door.
UD6. Utilize a cloud based server to store user and office door data.

Each of the above requirements represents a major component of our envisioned solution, and serve as the baseline for what our completed project needs to have. From these User Domain level requirements, we were tasked with drafting up several functional requirements for each.

These functional requirements had to consist of the lower level, but not implementation level, details of how the software would basically function. We also had to draft these in a way that any random engineer, who might possibly want to take up the project, could come in and view the requirements then know what would need to be completed. In order to refine these, we went through the same process as we did with the User Domain requirements, iterative refinement. Below are some of our main functional requirements, they follow a numbering scheme which attributes them to their respective User Domain level requirement.

UD2.FR1.	A guest visiting a user's office door should be able to opt in for notifications
UD4.FR1.	Widgets need to be created so the user can put content on their door
UD6.FR1.	Frontend will communicate with a database hosted on an Amazon cloud server

These functional requirements were then thought about in a performance context, such as how fast should requests take to process with the backend? Because we were not working with copious amounts of data the performance requirements were not the main concern in the requirements process, which leads into the environmental requirements. Thankfully our clients were very flexible with the environment that we were working in and didn't constrain us to using a specific language to code our project, and really only had one constraint, the project had to be easily deployable on a cloud platform. From all of our requirements we were then tasked with actually developing our software, this included coming up with an architecture that could easily be modified and was loosely coupled, which is how our architecture was designed.

4. Architecture and Implementation

To create a fully operational web application we needed to draft out an architecture that matched perfectly with our envisioned solution. This took the form of a Django REST framework utilizing AJAX requests made by the front end, which to convey this we have the following architecture diagram.

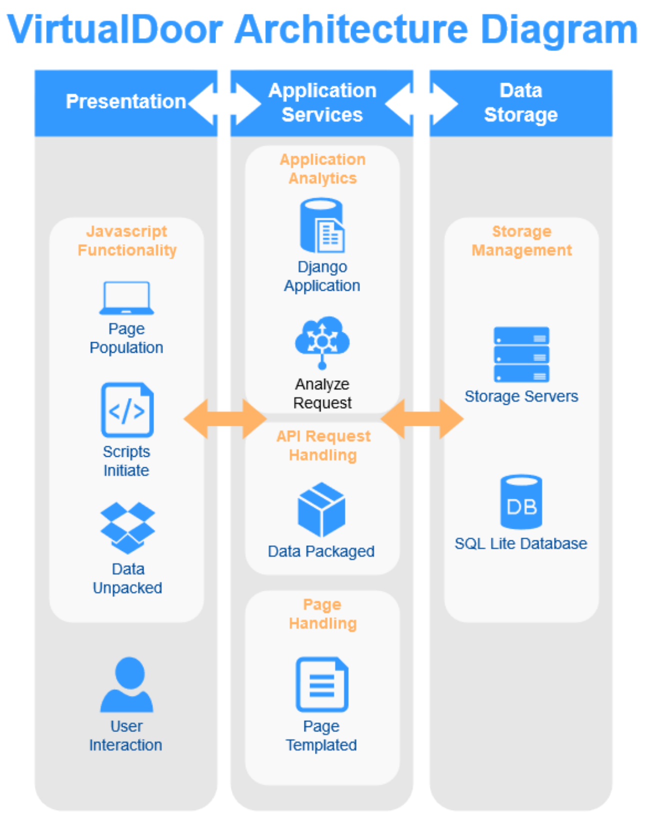


Figure 2: Virtual Office Door Architecture Diagram

In our architecture we have 3 main components, the Presentation component, Application Services, and the Data Storage component. The first main component, the presentation, is the main user facing interface. Coded in Javascript and utilizing Gridstack, Bootstrap and Webix, this is where all of a person's data is populated, where scripts are run to display the web pages and their formatting, and where the user physically interacts with the system. Out of all of the minor components that live in the presentation layer the main one is the Office Door itself, which we will cover in more detail later on. This layer also makes the requests to Application Services for all manner of data related to an office door, which is then taken from the Data Storage layer.

In the Application Services layer resides all of the logic for the entire Virtual Office Door application. Coded in Python/Django, this entails processing the different POST and GET requests sent by the presentation layer, as well as posting or getting data from the Data Storage layer. Within the application layer are also the models, serializers, views and URLs, all of which need to function together to effectively keep the Virtual Office Door running, these will be covered in more detail later.

The last layer, the Data Storage layer is nothing more than that, the part of our application where all of the data is stored. Because it is heavily managed by the Application Services layer, the data storage layer is a fairly simple SQLite instance that maintains all of the user data from start to finish. During a typical use case though, all of the components have to work together perfectly to maintain the system.

As an example, we can look at the simplest use case, adding a widget to a page and populating it with data. To start off with the user has to log into their door, which is done through the Google Login API. If they are a new user, they are then directed to a page to create their profile otherwise they are directed to their home profile page. Form there they can submit their new profile, or any changes made, which is a POST request made by the presentation layer, which is parsed by the application layer and then stored in the database in the User model. Once the user is directed to their door they are free to add widgets to it, which is only done in the presentation layer for now. Once a widget is on the page the user can either save their layout or wait for the system's automatic layout save, which is again simple post requests parsed by application services, then stored in the data storage layer. After the widget has been added to the page the user can insert data into the widget, say a sticky note, which is then packaged as a POST request as soon as the user updates the widget. This same program flow is done for every widget and any kind of layout, a completely cyclical flow of data with Application Services at the center.

To understand the intricacies of each layer, the following sections will detail how each component functions.

4.1 The Presentation Layer

In order to maintain ease of use in our system, we began by designing the Presentation layer in such a way that it would function on a high level, but still be easily accessible to any user in any domain. This is evidenced by our use of the Google Login API, which is heavily used at our client's university as well as in multiple other business domains. In regards to the software, our application utilizes Google's built in authentication system which provides us with an added layer of security for the application. This data is then used to populate the user's door by default or they can edit their information to what they would want displayed on their office door. From this the user is then directed to their profile page where they can edit their profile information as they please, each time a change is made the user must submit their profile to update it in the database.

From the login and profile pages, there is the actual office door itself. The office door functions as the main user interface and is thus fairly separate from the functionality of the rest of the presentation layer. The first major subcomponent of the office door is the layout, which is implemented by Gridstack, a library used to align objects in a grid pattern that also offers built in responsiveness as well as touch screen support. This layout is updated every 30 seconds when the door owner is editing their door as to retain consistency of information and lower the overhead on the door owner. Within the layout are the four main subcomponents of the office door, the widgets, all of which are Webix widgets

The first widget is the sticky note widget, which is fairly simple. It functions as a normal body of text, and is updated like one as well. When a user updates the content in the widget a JSON post request is sent to application services where the data from the sticky note, the sticky note text, is parsed and then stored in the database. The widget content is then updated in real time for the user and if they navigate away from the page, the sticky note content will still be there. The sticky note also does have the capacity to support HTML tags, this can easily be changed by adding in a regex that is already in the code, if the owner's specific domain doesn't allow it. In sum, the functionality of the sticky note widget is as follows:

- Allow the user to display simple text to guests
- Easily update the text in the sticky note via a popup attached to the widget
- No need for the door owner to force an update of the sticky note content

Secondly, there is the notifications widget which operates more so in the Application layer than the other widgets. This is done through posting data to the database, specifically any email that is put into its text field. The email is then sent to the application layer which parses it and then places it in the correct model, after which the registered email is sent a confirmation email. When the link in the confirmation email is clicked the guest is directed to the confirmation page where they can manage their email preferences (subscribe/unsubscribe). After that it requires no more interaction from a guest user, the owner of the office door can choose when to send out notifications by means of one of the icons in the toolbar of the widget. In sum the functionality of the notifications widget is as follows:

- Low overhead on both the guest and the door owner
- Guest can opt out of notifications at any time
- Door owner controls when notifications are sent
- Emails are stored securely in the database

The next widget is the calendar widget, the most complex (from an implementation standpoint) widgets. For the user the calendar widget is easy to operate, if they want to add an event they click a plus icon, highlighting an event and clicking the minus icon deletes an event. If the user made a mistake when inputting an event, they can highlight the event and then modify which ever field they need to in the event. Every time an even is created, deleted, or edited, a post request is done to the database to store the name, date, and time of the event, all of which is parsed by the application before any data is actually stored. In sum the functionality of the calendar widget is as follows:

- Easily add, delete, edit events
- Events can have a description, date and time associated with them
- Events cannot be added if they are before the current date

Lastly is the picture widget, which is fairly self-explanatory. For the picture widget, the user utilizes the popup for the widget where they can then choose a picture to upload and then click the refresh button in the widget to get the picture to display immediately. The picture widget can then be resized to the desired size for the image so that the image isn't squished in any way. This same process can be repeated for any picture the user wants, as when a picture is uploaded is simply overwrites the one that is current in stored in the database. In sum the functionality of the picture widget is as follows:

- Quickly add a picture into the widget
- Can resize the widget and the image will adjust size accordingly

All of the widgets do inherit some of the same functionality, this ranges from the static use of get and posts done through the application layer, to being able to move the widgets to the desired spot on an office door. Each widget also has the ability to become a responsive widget when the office door is viewed on a mobile device. In regards to that, widget content is also able to be updated on a mobile device, but widget layout and sizes are not.

One of the other key functions of the presentation layer is that all of the pages, except for the office door, utilize Django templating. This basically means that each page inherits a base design (specified in a base.html file) to promote consistency among all of the pages. There are certain components in pages that can be specifically altered; however, if the entire design of a page needs to be changed it either needs to be done in the base file or done specifically on the page. The templating referred to relates directly to the next section of this document, the Application Layer.

4.2 Application Layer 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.

4.2.2 Log-in

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 Django application 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.

The figure below shows each step of how the Django-social-auth pipeline works. As shown by the figure, once the user is logged in through Google, this package guides them and their account information automatically through this process in order to transform a Google login into a Django login for the Virtual Office Door website.

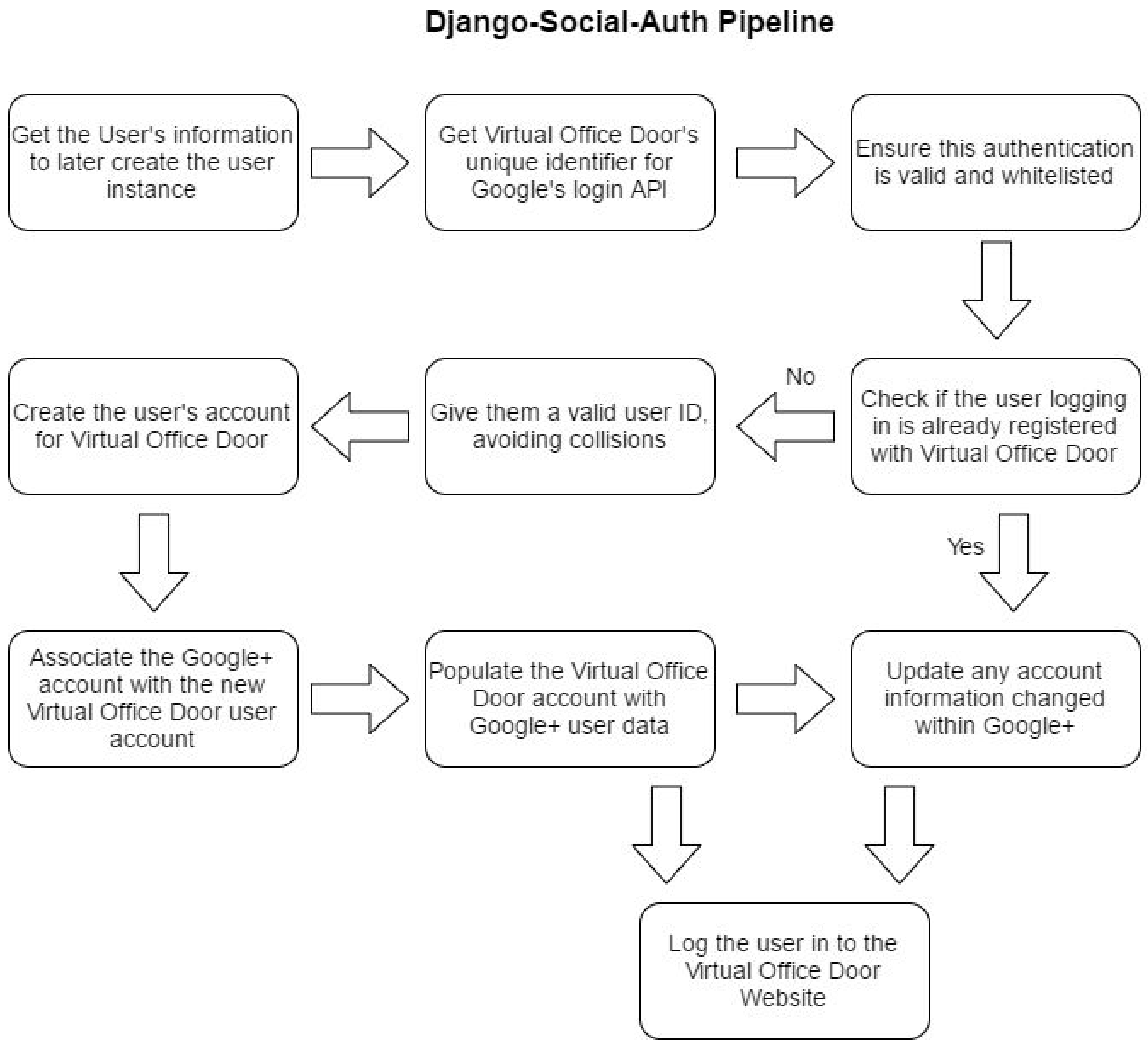

Django-Social-Auth Pipeline
Get the User's information to later create the user instance
Get Virtual Office Door's unique identifier for Google's login API
Ensure this authentication is valid and whitelisted
Check if the user logging in is already registered with Virtual Office Door
No
Give them a valid user ID, avoiding collisions
Create the user's account for Virtual Office Door
Yes
Associate the Google+ account with the new Virtual Office Door user account
Populate the Virtual Office Door account with Google+ user data
Update any account information changed within Google+
Log the user in to the Virtual Office Door Website

Figure 3: Django-social authentication pipeline

From the perspective of the users of our website, the Google+ login API provides an ability for the user to safely and securely login to a preexisting Google account. Designing around a preexisting login technology creates peace of mind for users of our website.

4.2.1 Models

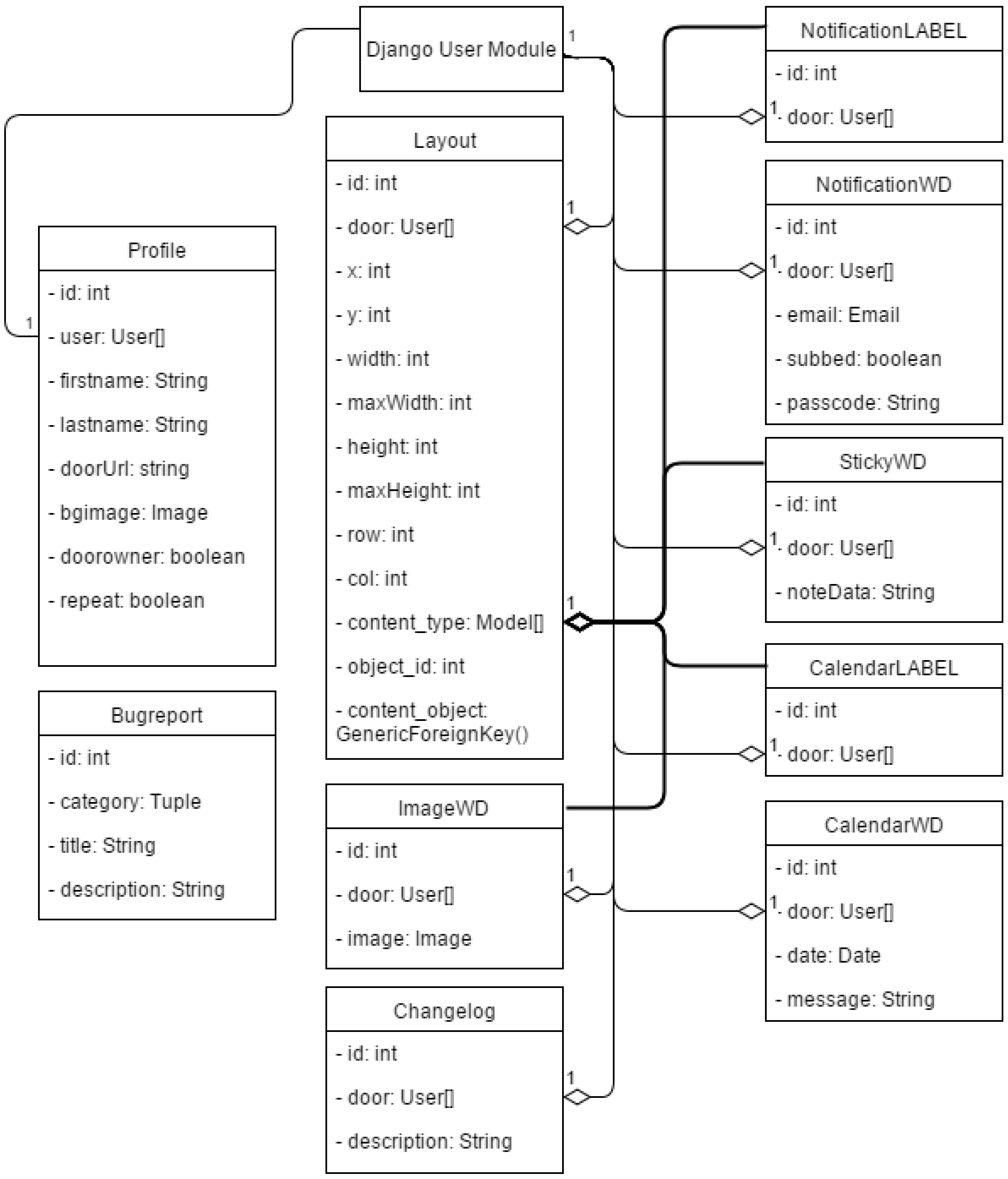


Figure 4: Virtual Office Door's models

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.

Each model 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 our application:

- 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.
- The NotificationLABEL and CalendarLABEL models serve as an anchor to tie calendar and notification widget data to a door. Since NotificationWD and CalendarWD are each zero-to-many record widgets, there must be some fixed, permanent way to bind the notification and calendar widgets to an office door without risk of losing the widgets.

By using these models together, our implementation offers 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.

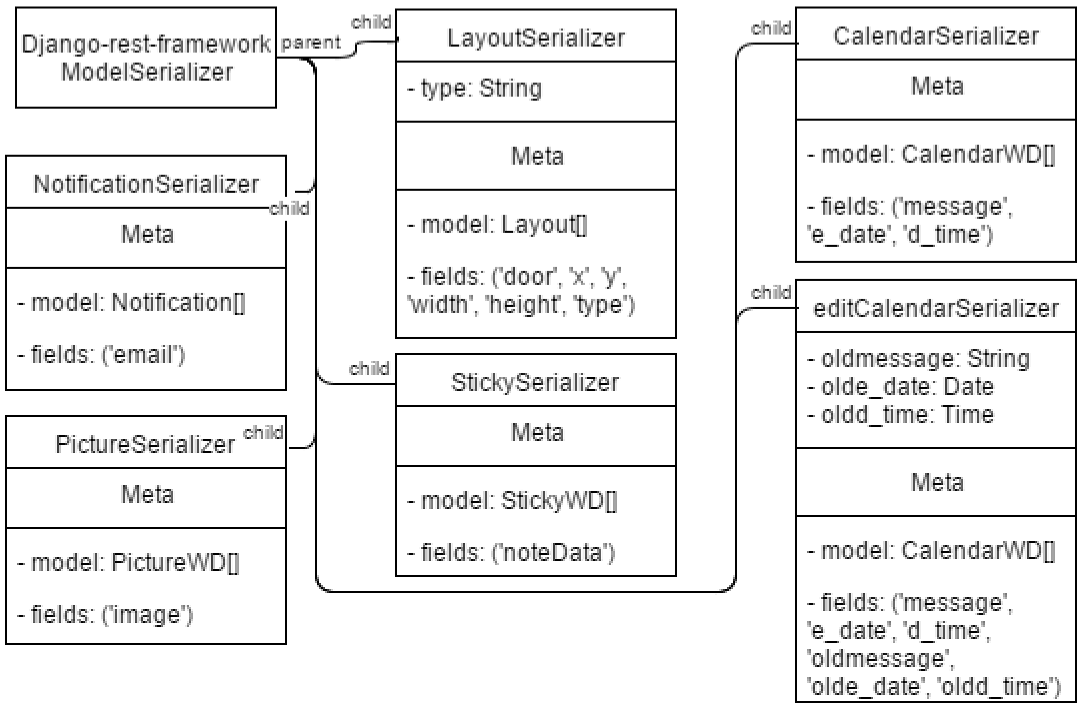


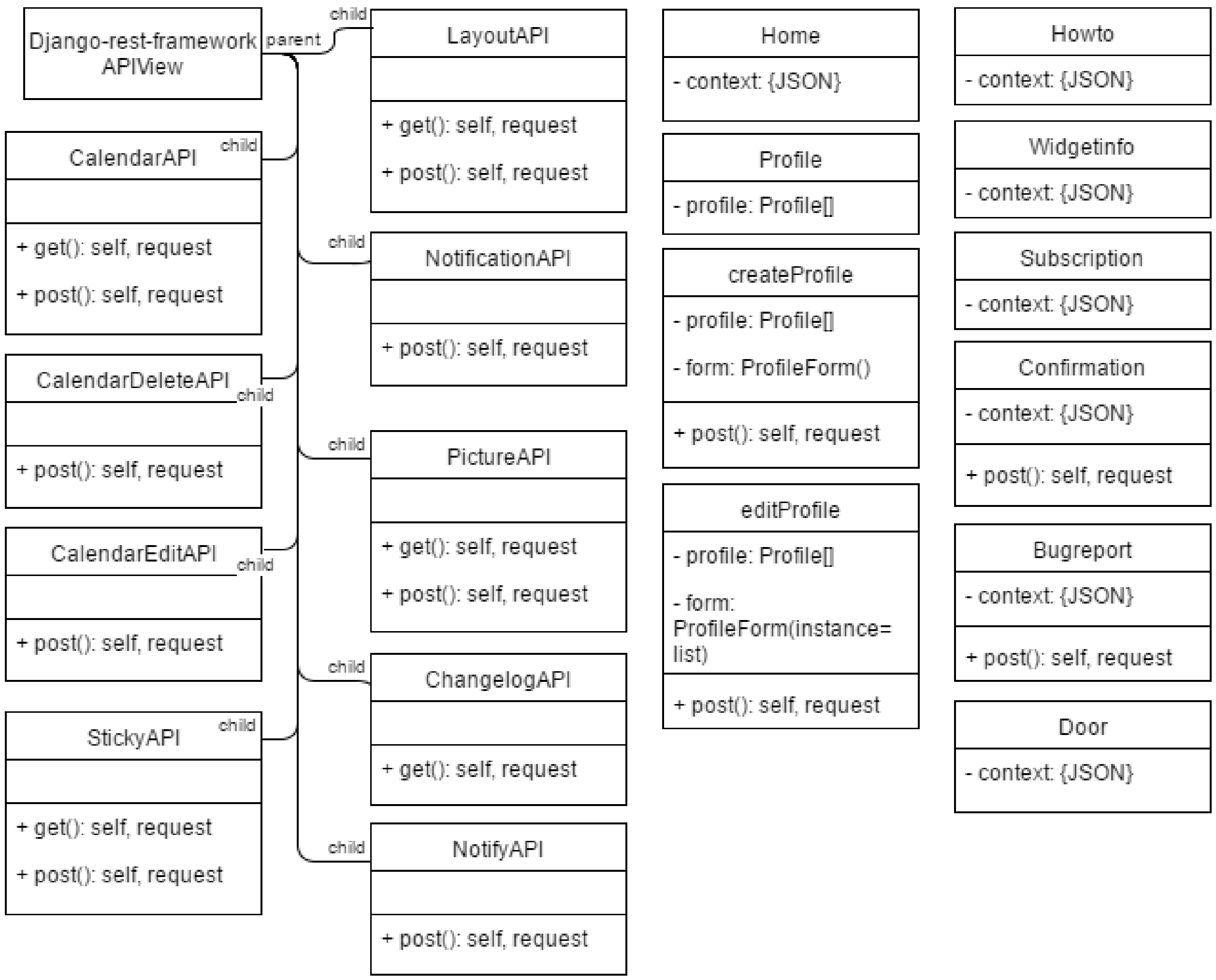
Figure 5: Virtual Office Door's serializers

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. 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. EditCalendarSerializer is a special case where it is overloaded with double the amount of information based on CalendarWD fields, the exact purpose of this is detailed in its corresponding API view below.

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.

Figure 6: Virtual Office Door's views

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 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. Depending on whether or not this entry exists, a different email is sent to the subscriber. If no entry existed, an email is sent with a link that will subscribe them to the door. If an entry existed, an email is setn with links that will subscribe or unsubscribe them from the door.
- PictureAPI inherits from the same class and handles the same request types as LayoutAPI. 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
- CalendarEditAPI takes a POST request containing data of a calendar event to modify for the requester's door. If this event is found, it is changed to the new calendar event information that was also sent in the POST request. If no calendar event was found or this change would result in a duplicate calendar event, it returns an error.
- CalendarDeleteAPI takes a POST request containing a calendar event's data to remove from the database. If the requester's door has no such event, it returns an error. If no event was found, it returns an error.
- ChangelogAPI returns all changes logged with the associated door accessed as JSON.

This view is currently unused.

- NotifyAPI is usable by any door owner, and upon a GET request it sends out a notification email to all subscribers of the request sender's door. Within this email is all door content changes since the last notification email, as well as a link to unsubscribe from the door's notifications.

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.
- The door view renders the virtual office door page of a user whose profile doorurl variable matches with the actual URL of the page. If none exists, the user is presented with a "door not found" page.
- The subscription view allows a user to subscribe or unsubscribe from a door associated with the same NotificationWD entry that contains their email based on an url pattern of "s" for subscribe or "u" for unsubscribe. Also in this URL pattern is the passcode for the NotificationWD entry mentioned earlier, and any ill-fitting URL pattern or one with no data to be found will produce an error message.
- The confirmation view allows a user to perform the actions noted above, but through a form that checks to see if the input email and passcode match a notification entry instead of just a passcode in order to change a user's subscription status to a door. No matching entry for the input information will generate an error message.
- The howto view renders the page containing instructional information on how to use the Virtual Office Door's main page: the door page.

- The widgetinfo view renders a page containing information about the different widgets available on the door page.

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:

- "^$", views.home
- “^howto/$”, views.howto
- “^widgetinfo/$”, views.widgetinfo
- “^confirmation/$”, views.confirmation
- “^subscription/(?P<status>\w+)/(?P<passcode>\w+)$”, views.subscription
- “^bugreport/$”, views.bugreport
- “^door/(?P<doorurl>\w+)$”, views.door
- "^profile/$", views.profile
- "^profile/create$", views.createprofile
- "^profile/edit$", views.editprofile

The following URL patterns are API views, utilized asynchronously via JQuery requests.

- "^api/notify/$", views.NotifyAPI.as_view()
- "^api/changelog/(?P<pk>\w+)/$", views.ChangelogAPI.as_View()
- "^api/sticky/(?P<pk>\w+)/$", views.StickyAPI.as_View()
- "^api/picture/(?P<pk>\w+)/$", views.PictureAPI.as_View()
- "^api/notification/(?P<pk>\w+)/$", views.NotificationAPI.as_View()
- "^api/calendar/(?P<pk>\w+)/$", views.CalendarAPI.as_View()
- "^api/calendardelete/(?P<pk>\w+)/$", views.CalendarDeleteAPI.as_View()
- "^api/calendaredit/(?P<pk>\w+)/$", views.CalendarEditAPI .as_View()
- "^api/layout/(?P<pk>\w+)/$", views.LayoutAPI.as_View()

- "^soc/" – access to the social_django package which handles information routed between the django application and google's login API, allowing for Google authentication.

The (?P<variablename>\w+) is parameter passing through the URL patterns. These parameters can be passed into the corresponding view functions, allowing for different page-rendering or data-returning actions to be performed for different data.

4.3 Architecture Wrap-up

Throughout the process of actually building our application our architecture actually changed multiple times. At the very start of our development process nothing was actually integrated together so that cause a huge issue to start, and we didn't have a set framework we were going to use for how our system interacted with itself. Most of the decisions we made on framework were originally driven on what could be implemented in the shortest amount of time possible, due to the time constraints on the project. We did end up settling with this REST framework which just so happened to fit the system that we envisioned perfectly so there isn't any difference between the plan and how it was actually built. The presentation layer is a completely different story though.

For the presentation layer we had gone through several iterations of what we wanted it to look like and how it was going to function. At first we had planned on using JQuery UI to build our different widgets, but that rapidly changed as we discovered that the learning curve and lack of some functions in JQuery UI would end up hindering us. This is how we found Webix, our current library being used, which streamlined the widget building process and it also boasts a less steep learning curve which fit our schedule better. Originally we were also going to go with a freely moving widgets, such as a widget could be placed anywhere on the page and have no grid formatting. This plan rapidly changed as we discovered the multitude of issues that could arise from possibly having widgets overlap, or having widgets placed outside of the view of the browser. Gridstack, our currently implemented grid technology saved us from having to determine if two widgets collided and what to do if they did.

In addition to that, we had originally not planned on making all of the pages (except the office door), template pages that heavily interfaced with Django. The original idea was for the django application to only handle the get/post/delete/etc. requests and handle parsing and storing data in the database. This changed as we discovered the ease of use and cleanliness that templating provided, which lead to the development of most of the major pages to follow a templated design. Also through using Django, we were able to incorporate the Django-social-auth package that severely streamlined the process of Google login we had originally planned to use. This package made interfacing with Google's login API much easier, and saved us the trouble of running a Google authentication endpoint locally and tying it into our application manually.

A multitude of additional API views, page views, Serializers, and URL patterns were developed as needed as development progressed. We quickly realized our base-design was insufficient in accommodating all of our requirements, so we made new modules as needed until our application was able to satisfy the intended functionality. With the exception of serializers, the URLs, the APIs, the Models, and the views each nearly doubled in number. The Models specifically were shaken up a bit in terms of foreign key relationships. Having each foreign key that pointed towards a Profile now point towards a User made dereferencing much faster and allowed information associated with the same user be accessed much more easily.

Once the architecture fell into place, we then had to address testing our new software that we had just built.

5. Testing

The development of this project heavily focused on a design, test, and revise process. During initial stages of development, our main form of testing and validation was to deploy the software as a local server to permit testing in a browser. This allowed us to get instantaneous results when checking if any recent modifications made to the codebase were passable. While this is not the only testing strategy we incorporated, this was the most heavily used. Another testing method we took advantage of was unit tests, where we tested to see if the Django backend was successfully handling requests correctly. Initially, we spotted that there were some issues with requests being handled incorrectly and this caused some complications on the front end. However, since our testing results indicated it was due to data serialization, the fix was swift and easy to implement. Lastly, the most rigorous and fruitful test we ran was a live demo in a CS class. This test was by far the most extensive as it allowed us to verify a multitude of use cases and stress tests on the Django backend such as load balance testing, request handling, and user login/validation testing. The results of this test were the most helpful in revising our system as the log captured every request, failure, and error that the system encountered during use. From this log, we could document the notable bugs/issues reported during the demo into our bug report spreadsheet. From there, we could modify the codebase to rectify these issues and ensure that any errors experienced during the demo were taken care of.

Even though testing didn’t come along until further in our development process, we did manage to stay on track with all of our milestones and remain on schedule.

6. Project Timeline

For both the fall and spring semester the amount of work that needed to be completed was fairly large. In the fall semester we got to experience what the whole planning phase in a piece of software consisted of. A few of the major milestones in the fall were:

1. Acquire Requirements from the client
2. Prove the technologies we chose are feasible
3. Create and get our requirements document approved by the client

4. Create a prototype that will be utilized in development

Out of all of the milestones listed above, and even the ones that were not listed, we completed them all on time and our client and mentor were satisfied. We then had to begin the actual development work which started in the spring and is represented by the Gantt chart below.

All of the segments in the chart represent major development related milestones, where the milestones that were left out were the different document deliverables.

As the Gantt char shows, we have been on track with our development the entire half of this year. The only major set back was during the middle of March when the group had spring break. It should also be noted that Testing, Refinement/Bug Fixing and Documentation are all overlapping. This mainly has to do with that fact that all three of those tasks were being worked on at the same time by different members of the group. We did this to stay as productive as possible and to resolve as many issues as possible before we had to demo our project at the UGRADS symposium at the end of April.

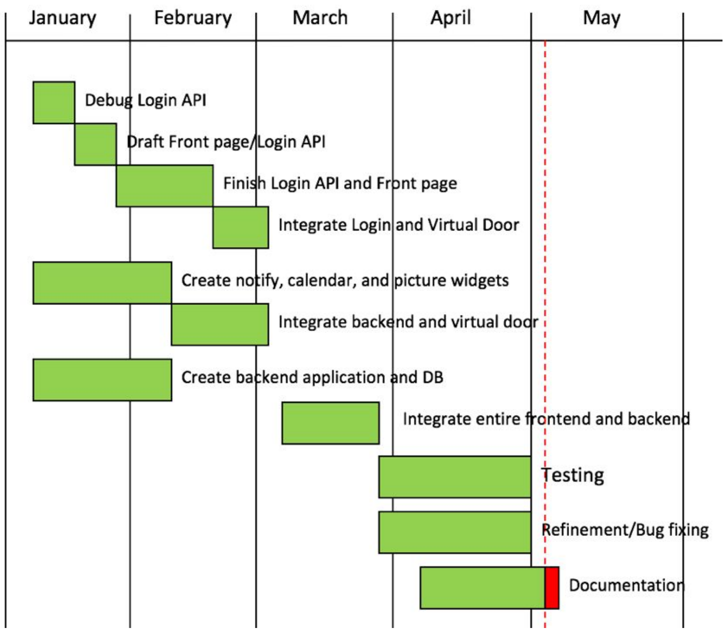


Figure 7: The project's schedule

In regards to the different development blocks themselves, many of them overlap too. This was because we had different team members working on the project at the same time. The Debug Login API, Draft Front page/Login API, and Finish Login API and Front page were being worked on by two of our group members, David and James. At the same time Mitchell and Nick were tasked with creating the application and the database, while Tyler was assigned with creating the different widgets. All of the different sub-teams came together for the integrating of each of the pieces of the software, so David/James and Nick/Mitchell worked on integrating the Django application with the template ready front page (and the other pages in the site), and then Tyler worked with Nick/Mitchell to integrate the office door and the backend application.

In the latter half of the semester, James and David took the lead on the Testing portion, which fed directly into Tyler and Mitchell leading in the refinement and Bug Fixing. During that same time Nick took the initiative and lead the documentation phase. In the current timeline we are still in the documentation phase as the user manual still needs to be completed along with the actual product delivery to our clients. However, this Gantt chart no where near covers the potential work that could be done to make this application even more useful than it already is.

7. Future Work

For the Virtual Office Door, the future is full of nothing but improvements, most of which we either just did not have the time to implement or brainstormed at the completion of the project. Each idea in itself represents new and challenging ways that the technologies we implemented could be morphed into something that could easily replace most technologies currently utilized. Below is the list of ideas:

1. Duplicate Widget support
   In the current implementation of the office door, only one instance of a widget is able to exist on the page at a time. This unfortunately introduces the constraint of only having one calendar/sticky note/picture widget for a professor that might have multiple classes. If this feature were to be implemented it could expand the functionality of the office door to cover all of the needs of a professor with a multitude of classes. Because our development team has already put time and effort into researching this idea, implementing it wouldn't take much time at all and improve the application overall.

2. Google Calendar integration
   This idea was originally not posed by our team; it was mentioned to us during the Undergraduate symposium. For future implementations there could be an option to integrate google calendars with the calendar widget that is on a user's office door, which could effectively eliminate the process of adding an entire calendar's worth of events by hand.

3. Customizable Notification preferences
   Currently the notifications widget sends out a preformatted email that is not created by the door owner themselves, an admin has to format it. This could easily be changed by allowing door owners to customize their email that is sent out as well as what content is sent out in the email.

4. Different login methods (twitter, facebook, etc.)
   For the users of the office door that might not have a google account, future developers could implement a Facebook login, which could work as a substitute for the Google login. If a development team wanted to, they could even implement their own login system from scratch and eliminate the requirement of having a Google or Facebook account.

8. Conclusion

The idea for the Virtual Office Door was originally brought about by our client Dr. Michael Leverington, who after several years of having a cluttered office door, decided that it was time to change. The inefficient and unreliable method of posting time sensitive and important communications on an office door needed to change, which is where our solution comes into play. With the Virtual Office Door, the application can essentially cut out the need for students or fellow colleagues to physically be at a person's office door just to find that they will be "Back in 15 minutes." This not only saves students and colleagues time but it also saves the door owner from having to either communicate they will not be in their office via email or by putting a sticky note on their door. Not only that but our Virtual Office Door provides the following features:

- A clean and easy to use user interface
- Multiple configuration options for door owners
- Different widget types to communicate vast types of data
- A notification system to alert guests to the door of changes that were made
- Easily deployable on any cloud platform

All of these features will not only make it easier for our client to communicate with students, but he could also use it to help organize his personal schedule. Our application could also be used in a variety of domains, meaning it isn’t strictly limited to academia. The Virtual Office Door could easily become the Virtual Cubicle Wall or another implementation in that likeness.

Overall the Virtual Office Door was an amazing project to be tasked with completing and it effectively represented the culmination of the Computer Science program. Our team enjoyed working on each component of the project and found it challenging and rewarding at the same time. Despite there being some setbacks during the development process, the entirety of the Capstone class and our semester of development work stayed on track and we were able to complete all milestones on time and to the best of our ability. The Virtual Office Door is a functional application that works as intended right now but could be improved upon and made into an application that could serve multiple domains in a clean and professional manner.