# Project Report

**Michael Zekoff**

**seeking**

**Master of Software Engineering**

**under the guidance of**

**Dr. David Umphress**

## Abstract

Applicants to the graduate school at Auburn University are currently served through the Graduate Web Application and Admission Process system, or GWAAP. This system was originally designed for handling operations at the Graduate School, not for processing applications at the departmental level. The goal of this project was to apply software engineering practices in the creation an expanded and updated system as a replacement for internal use by the CSSE department. The new system is a web application using the Django framework to serve both applicants and faculty members. Applicants may use the system to submit, manage, and receive feedback on their application, including a recommendation system to replace letters of reference. Faculty members may use the system to view, comment on, and vote on the suitability of applicants, and enjoy a higher degree of automation for common tasks related to applicant evaluation.

# Chapter I

Applicants to the Graduate School at Auburn University are currently served through the Graduate Web Application and Admission Process system, or GWAAP. As a result of being several years old, its interface has aged and does not benefit from modern web design practices. Furthermore, it was originally designed for handling operations at the Graduate School and not for processing applications at the departmental level. CSSE faculty members evaluating applicants do so primarily through a physical paper trail that is collated and passed around for each applicant. The GWAAP system for faculty is a thin GUI around the GWAAP applicant database, offering little to ease the several pain points of evaluation. There is a need for a more robust solution for both groups.

Applicants will benefit from a new system in several ways. By collecting and managing data in one place, there is no need for the multiple usernames, PINs, passwords, and reference numbers that currently exist in the GWAAP system. Having a single point of reference also helps applicants by streamlining the assembly of their application and providing clearer, more immediately feedback. Applying a modern layer of polish to an online system also helps present a more desirable front for the university, which is especially important considering the CSSE department in particular.

The benefits of a new system for faculty members are greater still. One of the primary benefits is the elimination of a great deal of physical documents being passed through the department. It is more difficult for all involved parties to manage multiple paper copies of transcripts, resumes, letters of intent, and standardized test scores when the same documents can be both uploaded and used in purely digital forms. A new system can also allow a digital forum for faculty discussion on applicants. By opening up applications to easy, digital commentary and voting, more eyes are available to evaluate applicants and they can do so faster. Increased automation is another benefit that can be leveraged in a new system. Much of the boilerplate communication between the CSSE department and potential students is still done by hand. By turning control of these important, but mindless, tasks over to the system, the rate of error and the delay in communication can both be decreased. Finally, a streamlined application process for applicants means less time spent by faculty members responding to confused emails by potential students.

From this problem, a set of use cases was derived that a theoretical replacement system should be able to fulfill. These use cases are included as Appendix A. Details on the specific use cases and how they were implemented in this system are included in Chapter 3.

# Chapter II

The requirements for a new GWAAP system necessitated a web-based solution. Because the system deals with applicants' private data, that data must be able to be secured. Additionally, there is a need for a definitive version of the material that makes up each application. The easiest way to ensure that an authoritative version of the data exists is to have control over it. Furthermore, this data may need to be accessed simultaneously by multiple users. These things point to a RDBMS as the method of data storage for the system. Next, some method of interacting with the database must be available to users. Providing SQL access directly to the data is a poor choice for reasons too numerous to list here. A desktop application that communicates with the database is an option, but is undesirable for at least two reasons. One, it would require that users download and install a desktop client prior to submitting their application. Two, it would ignore an extremely robust infrastructure that already exists for client-server applications: specifically, web browsers and the Internet. Web browsers provide very well-defined ways of interacting with a server running an authoritative copy of information at a remote location. Furthermore, the ability to use HTML and CSS markup for the application greatly speeds development time. It effectively provides a huge, stable, well-supported library of GUI widgets for building the frontend of the application.

Although a web application is the clear choice, there are roughly three levels of granularity in implementation that can be targeted. The first is a raw programming language, like PHP. The second is a Content Management System, or CMS, like Wordpress. The third is a web application framework, like ASP.NET. This is a spectrum of course, and the distinctions between these levels can be fuzzy. It could be argued that one system could be moved up or down the spectrum given certain circumstances, but the general idea is that different implementation methods provide different basic levels of functionality. There are additional concerns beyond programming granularity as well. One is the need to interact with a legacy system, i.e. the existing GWAAP implementation in use at the Graduate School. A second concern is the need for a secure system, meaning resistant to unauthorized access (accidental or malicious). A third is stability and control over the final system, since it will serve not only as the primary repository for official documents, but also as a platform for higher-level activities—most notably, faculty collaboration and evaluation of applicants.

Of the three implementation levels, a raw programming language provides the most flexibility. PHP is offered as an example because it is historically targeted at programming for web applications, with an extensive standard library for web development as well as thorough documentation. Other common choices include Python, Java, and Ruby, although nearly any language can be used as most applications using the Internet backbone boil down to an ordered series of string inputs and outputs (at the application level, at least). The downsides of a system programmed in this way from the ground up are centered on the need to reinvent the wheel for a great deal of functionality that isn't specific to the system itself. For instance, user authentication in web applications is a complex and problematic area that has a very high degree of importance. However, it is also a problem that is very similar no matter what system is being implemented. By virtue of being a stateless protocol, HTTP has no inherent way of setting up a user session if the identity of the user is needed for interaction with the system. For instance, applicants and faculty members need to interact with the system in different ways, but it is infeasible for the server to open, authenticate, and maintain a socket connection with every user that connects to the system. Instead, HTTP requests are usually handled by the principle of REpresentational State Transfer, or REST. In a RESTful system, well-defined HTTP verbs are sent as requests against URIs that the server exposes to all users. This means that every user, at all times, is technically making each request anonymously. If some content on the server needs to be hidden behind authentication, those authentication details must be sent along with every request so that the server can authenticate the requester and determine if they have valid access permission to the resource exposed through the URI. This presents a number of problems. First, different URIs may require different access credentials. For example, the URI that Google exposes for checking webmail requires a different set of credentials (e.g. username and password) from the ones an online banking website requires. It is undesirable to send all available credentials for a user to each server that the client makes a request of. At the same time, users expect a session-based interface even with web applications. Just as entering a username and password once for a terminal session authenticates a user until they explicitly log out, so do users expect a single login to authenticate them to a web service until they are finished making requests. (The alternative would be requiring a username and password field on every request made to the server, which is both unsafe and obnoxious.)

To bridge the gap between user expectations of sessions with web applications and the server's necessity for stateless transactions, cookies are used. A modern session cookie usually includes nothing but a hashed or random string either derived from login information or issued by the server upon first authentication of the user. This cookie is used as user credentials on subsequent requests rather than the username and password strings.  In addition to improving security, the server can use information about the resources being requested along with the requests themselves to provide the illusion of statefulness to the client.

This comes full circle to the problem of implementing web applications in “pure” programming languages.  Every bit of the above functionality must be implemented along with the rest of the system.  Not only is this time-consuming, error-prone, and difficult, it is a problem that has been solved already.  In the context of developing web applications, using a pure language is comparable to programming using assembly language.  While there is certainly still a need for it in certain cases, the power of already-existing systems is such there would have to be an extreme need for such a raw level of control to make the headache worthwhile.

In contrast, Content Management Systems form the other end of this spectrum.  The WordPress CMS (related to but not the same as the WordPress blogging service) is arguably the best-known CMS, and implements not only every bit of authentication functionality but much more as well.  WordPress, along with other CMS offerings such as Joomla and Drupal, abstract away as much of the implementation details as possible and attempt to leave users with fully-functional systems with the absolute minimum of knowledge or setup required.  (How little knowledge?  Many web hosts now offer “One-Click Wordpress Install” to users who want to get the system running using all default settings.)  There are benefits to this approach.  First, third-party developers often offer plugins or themes that can change the look or functionality of a CMS install with little mess for the end user.  Second, CMS providers tend to be very good at the things they do best—it would be difficult to design a blogging service from scratch that would operate as well as WordPress does.

Unfortunately, while a CMS is extremely well-suited for non-technical users or users who desire common functionality, they can be poor choices for niche use. One of the drawbacks is their extreme overhead—by accounting for all possible use cases, the system can swell to a size unmanageable by a single developer. (For rough comparison: a basic Wordpress install is in the neighborhood of 125 MB, while the entire GWAAP system developed for this project is around 600 *kB*.) The size of the system on disk is hardly a concern, but the number of files to go bug-hunting through is very much a concern for a developer attempting to add or especially change existing functionality. The size and sprawling focus of a typical CMS is the biggest drawback to them for a project such as this. Although it may be very good at what it does, it is highly likely to be awful at things it doesn't do, if it can be coerced to do them at all.

The middle ground is the use of a web framework, like Django, Ruby on Rails, or the Microsoft ASP.NET stack. A web framework takes a more balanced (though not necessarily better) approach to built-in functionality. Frameworks will typically solve many of the foundational problems in web application development (like user authentication) but do little more than that. Ruby on Rails is a popular framework because of its "convention over configuration" approach, in which the development model is generally taking the base use case of the Ruby on Rails system and defining any ways it should behave *differently* from its standard configuration. The ASP.NET framework is also popular, as are frameworks such as CakePHP and Catalyst (a Perl-based framework). Django is a Python-based web application framework. Like any other approach, it has many benefits. An experienced Django developer can prototype rapidly with it. It includes a very mature testing suite that builds on top of Python's already-robust unit testing framework. It does a fine job of providing basic functionality in its custom settings, is well integrated with Eclipse, and provides good command-line tools for interacting with an application under development. Best of all is its ability to abstract (but not hide) details of database implementation. Database tables are defined in Python using inheritance from some base classes provided by Django, and an external tool is used to generate and then run appropriate SQL calls to the database. It is quite possible to develop an entire database-backed application without ever making a manual SQL query.

Django has at least two major drawbacks, however. First, the learning curve for Django is significant. It is a very large, very powerful, well-established system under active development. It is not as simple as, say, learning a new library for Python. Rather, it is more closely comparable to learning an entirely new language based on Python syntax. Second, Django is very rigid in its approach to certain aspects of web application architecture. While its Model-View-Controller separation is very clear and easy to understand, there are fairly extreme levels of complexity within each layer and Django is very strict about the way it wants things done, and the order in which it wants them done in. This is especially problematic when designing a system using software engineering standpoint, which is further documented in Chapter 3.

Ultimately, a system built in Django is able to satisfy all non-functional requirements of the proposed system. It can be backed by a RDBMS, it can be written to interoperate with a legacy system, it puts a high level of control in the hands of the CSSE department, and it can offer thoroughly-tested security.

An additional non-functional requirement unique to this project was to apply software engineering principles to the construction of a web application.  To help achieve this goal, construction of the system followed PCSE, a personal software process under development at Auburn University.  PCSE is a full-lifecycle process that addressed every section of the system, although some practical modifications were made during the course of construction.  The PCSE process, its application to this system, and modifications made to it are discussed in Chapters 3 and 5.

# Chapter III

## Analysis

The first stage of this project was describing the needs that a new system would have to address. PCSE refers to this part of the lifecycle as "Analysis", and for this project it took the form of a set of use cases derived by Dr. David Umphress (see Appendix A). The use-case document included a diagram of actors and how they might interact with various components of the system and each other, as well as 14 use cases describing specific functionality desired in the final product. Each use case included information like objective, entry and exit criteria, basic and alternate paths, and notes or diagrams expanding or explaining the desired functionality. The use cases document was purposely non-specific regarding implementation details, describing a black-box approach of inputs and outputs that a user would expect from the system.

This document was supplemented by occasional stakeholder meetings to refine the ideas presented in it, including a lengthy initial meeting to fully describe the problems facing the existing GWAAP system and how the use cases were to address those problems. Taken as a whole, this information represented all necessary requirements to complete PCSE's information-gathering Analysis phase. Because the use cases matched existing pain points, and because they specified some parts of the system's complete functionality, the Analysis phase more closely matched a real-world scenario in which a solution to a known problem has to be constructed to match loose stakeholder requirements. This is in contrast to a typical (undergrad) academic scenario where a toy system is created to exacting specifications, or a system is created to address a theoretical problem.  The flexibility of PCSE was a benefit here, as the process could accommodate this type of Analysis phase as opposed to a system where actual components or function points had already been specified.

## Architecture

The next phase of construction was a process of mapping the system requirements to an overall architecture as well as individual components.

From a design pattern standpoint, Django effectively mandates a Model-View-Controller architecture.  Because this pattern is exceptionally well-suited for web applications, it makes sense to base the architecture of the system around it.  Django deviates from the standard terminology for this pattern, however.  Django Models are equivalent to the M of MVC, although Django does allow models to define their own methods for convenience or clarity.  Django introduces confusing use of the term "View", however.  In Django, a View is a Python method that contains the logic for processing requests and rendering HTTP responses to the client.  This is analogous to the Controller in standard MVC parlance.  The display layer, normally known as the "View" of MVC, is referred to as the Template layer by Django.  In short, Model-View-Controller architecture is better described as "Model-Template-View" for the sake of Django.  The concepts in each layer map more or less as expected, so it is primarily just a difference in terminology.  Because of the well-defined nature of MVC architecture, a good deal of the process was mapping use cases to their necessary components among the three layers.  Models were derived from logical needs in the use cases.  For instance, the use case specifying the need for commenting on applications led to the Comment model containing a foreign key for the application it points to, an additional foreign key for the user who made the comment, and a character field for the contents of the comment. The foreign key to application is required (since it doesn't make sense to leave comments in the database for applicants who don't exist or have been deleted), but the foreign key to the faculty member was made optional from a database perspective to account for admin-level comments not associated with a faculty member, as well as to prevent deletion of users from breaking the comments on an application. During this phase, all known additional models were derived in this way. The need for a few extra models was determined during construction and is outlined below.

However, the details of Django implementation caused certain architectural issues to arise after the fact. During this phase, one of the most difficult problems of applying standard software engineering process to a Django application reared its head. As part of Architecture, the inheritance hierarchy for users was developed. Django offers a User base class outfitted with authentication functionality. The GWAAP system, however, has two major classes of user – faculty members using the system to evaluate applicants, and the applicants themselves. The boilerplate authentication needs for both (permissions, sessions, etc.) are similar, but each class of user has access to an entirely different set of use cases and expect a different path through the system. From a purely architectural standpoint, a natural choice would be to have children that inherit from Django's User base class. One, representing a GWAAP User (i.e. faculty member, administrative staff, GPO, etc.) needs high levels of access, while the other, representing each Applicant, needs low-level access tailored to their personal application. This fell out into two components to be built in the first iteration.

The problem in this situation stemmed from the fact that, as mentioned before, Django has very strong notions of how things “should” be done. Due to a lack of implementation experience with Django, it was unclear during the Architecture phase that subclassing Django User presents a number of problems. (Due to namespace conflicts in the prose description of these classes, the base Django user class will be referred to as DjangoUser, and the GWAAP user class will be referred to as GwaapUser.) First, all classes that inherit from DjangoUser are accessible by querying the DjangoUser object manager. Although this can be accounted for in custom views and test cases, the Django framework itself does not differentiate between GwaapUser and Applicant when pulling them from the database. The primary issue here is that the authentication/session middleware of Django passes Views an object containing the currently logged-in user as a field. No matter what the actual class of that user is, Django can only recognize it as a DjangoUser. This forces awkward casting within each view to gain access to specific attributes or convenience methods of the models.

This is the second major problem of subclassing from DjangoUser: because all users are the same at the framework level, it is possible to give certain classes permissions they would not otherwise be entitled to. Since these permissions are the *only* way of preventing access to unauthorized user types at the framework level, abnormal user types with non-default permissions can progress to Views they would normally be rejected from. Within Views that attempt to cast users to classes they don’t actually inherit from, database errors will be thrown. This is not in and of itself a security hole, because non-default permissions can only be set by system administrators (who have the power to cause other, more serious security vulnerabilities). Nonetheless, it is troublesome because it is conceptually confusing and because it can result in bizzare HTTP 404 and 500 errors.  Notably, superusers automatically get all permissions by default, meaning to prevent these unexpected behaviors they need to deactivate their own "is_gwaap_user" and "is_gwaap_applicant" permissions.

Permissions were ultimately used as the solution to this architectural problem—within Views that need to access specific fields or methods of users, requests are first filtered by permission (redirecting unauthenticated or unauthorized users to the appropriate login page), then pulling the appropriate class of user from the database with manual calls to their object managers.  For normal operations, this means that the system *can* be designed to match the architectural expectations of child classes off of DjangoUser.  One of the benefits of this is that Applicant models can have database fields or related rows in other tables that are exclusive to them.  GwaapUser models do not need entries in the table of Application models or Profile models, and this way those tables are not filled with rows upon rows of null values.

Attaching extra information to Applicant models is accomplished through a hybrid of two Django-specific techniques:  UserProfiles and signals.  UserProfiles are the canonical Django way of adding extra information to DjangoUser objects.  In recent versions of Django, support for true subclassing and explicit addition of extra fields is improving, but documentation on these features is less robust and at times overshadowed by old but popular workarounds for previous versions of Django.  Signals are messages sent by the framework upon certain events in the lifecycle of an HTTP request.  Methods can register themselves to receive signals that interest them, take whatever action they need to respond to the message, then return control to the Django dispatcher.

In this system, UserProfiles are used to add additional information to Applicant models only. The Django framework provides a get_profile() method to all DjangoUser objects which returns an instance of the custom profile class defined in the settings file and hooked to a custom model. Arbitrary extra information for User models can be attached to their UserProfile and retrieved by Views during the processing of HTTP requests.

Signals are used at creation time for these models. Django offers (among other types) a "post_save" signal, in which a specified method receives notification any time a model is saved to the database. Methods that wish to receive this signal will get at least three parameters: a class name expressing what kind of object is being saved, a Python object representing the instance of the model object, and a boolean value expressing whether this object already exists in the database or is being added for the first time. This signal framework is used to handle appropriate instantiation for Applicant objects. One particular method listens for any time a new Applicant object is being added to the database. After being added (and thus after it has been issued a primary key) the method instantiates a new user profile for the applicant and saves it to the database as well (thus ensuring that *it* now has a primary key). An Application and GwaapProfile object are then created as well with foreign keys pointing to the user profile. This process (which is admittedly one of the more convoluted Django idioms) makes possible at least two things. First, it means that GwaapUser objects do not have Applications associated with them, which would not only take up space in the database but also makes no sense. Second, it guarantees that if an Applicant exists in the database, it also has a matching Application object to hold information associated with it. This is very useful because it means that Views using Applicant models do not have to manually check each time for whether the

Applicant actually has an Application – they can (correctly) assume that Applicants always have Applications and make calls on it with that expectation.

Two additional questions arose during Architecture. The first was related to a single applicant having multiple applications. The Applicant model logically matches the human who is applying – the model contains information on username and password along with email address, none of which *must* change even if the person applies a second or third time. (This is in contrast to the existing GWAAP system, where applicants must begin the process completely from scratch including a new account.) One option would be to allow applicants to attach an arbitrary number of accounts via foreign key relationships. This would handle the problem of applicants who make more than one application. It would also allow old information to be neatly reused in a new application. For example, an applicant wishing to reuse an old Reference could simply change the foreign key in the existing Reference to point at the new Application object. The problem with this setup is not an engineering one, but a user interface one. For most users (who will only apply once to the department), it adds an additional layer of difficulty in understanding how the parts relate to one another for no benefit. Applicants must set up their applications separately from their account, which for most will be a meaningless or even confusing differentiation. By setting up a one-to-one relationship between Applicant and Application, there does not have to be a differentiation for the most common case. In the rare cases where an applicant does want to reuse their account information with a new Application, this can be achieved manually by an administrator. For this system, it was determined that this one-to-one relationship was the most sensible approach.

A second additional concern was the way Reference models would be handled. One of the major use cases of the system specifies that external users are able to interact with the system and register their comments and rankings as a reference for a given applicant. References are given an online form that is meant to take the place of a paper recommendation. Not only does this eliminate a large amount of additional paperwork for all parties, but it standardizes to some degree the applicant qualities addressed by references. By offering a defined ranking system and categories for references to fill out, faculty evaluators can not only compare apples to apples but also have data on qualities that are important for acceptance decisions. One of the biggest questions in this system was how references would authenticate themselves to the system. One solution would be to have them create accounts with the system just like applicants. A benefit of this approach is that Views handling references could be written in the same ways as those handling applicants and faculty users; i.e. checking for appropriate permissions and having built-in access to the User representation of the person accessing the system. A second benefit is the additional level of security and confidence for both references and evaluators. By ensuring that all persons filling out recommendations are authenticated to the system, involved parties have an additional measure of confidence that references are who they say they are, and that contact information can be retrieved if it is suspected that they aren't.

Here again the problem lies in the user experience with the recommendation process. Asking users who will probably never again interact with the system to create an account with it is non-trivial overhead both for the database and for the people serving as references. Since all applicants are required to have three references, that category of users would quickly swell to much larger than faculty users and applicants combined, yet most of them would never need or use their account with the system again.  Second, the whole online reference process is intended to both lessen the burden of recommendation on those third parties as well as increase the value of the recommendations to faculty members evaluating applicants.  Adding an account creation and login overhead to the process runs the risk of irritating those users which might jeopardize the validity of their data.  It was determined that for this system, references should not need to log in or create accounts with the system to use it.  The implementation details of that were purposely deferred to a later iteration.

## Iterative Design and Construction

Having completed Analysis and Architecture, PCSE becomes an iterative process in which the Planning, Construction, and Refactoring steps are completed in cycles, each cycle moving closer to a complete system.  PCSE does not mandate the order in which components are built.  Therefore, there can be a number of approaches to dividing the implementation into iterations.  One approach is to build a minimum viable product in the first iteration, with each subsequent iteration adding additional functionality, polish, or other value.  Another approach is to take one or more functions of the system at a time, implement them to production quality, then repeat.  Yet another approach is to build components in order of their projected difficulty from hardest to easiest or vice-versa.  For this project, it was determined that the implementation of Model components would be the focus of the first several iterations, followed by View components, and finally by Template components, with one or more iterations for polish near the end after the basic functionality of the system was complete. There were several motivations for this.  First, the model components were foreseen to be the easiest to implement. Although this turned out to be incorrect, the thought was that practice on the easier components would be desirable as a method of becoming familiar with the complex Django framework. Second, the nature of the framework is such that the Models must be in place for the Views to have anything meaningful to interact with. Third, Models and Views form the core of the system, at least from a business logic standpoint. No viable product can exist apart from them so it seemed backwards to focus on writing a Template layer for a product that did not exist. Finally, as the Template layer represents mostly just the user interface/experience, and this project focused on software engineering principles, a functional system and test suite was deemed of more value than a slick interface, at least early on.

## Iterations 1 through 4

Iteration 1 tackled the core authentication issues of GwaapUser and Applicant, how to subclass them from DjangoUser, and how to attach Applications to Applicants. In the course of implementing these classes, the problems described above of mapping architecture to implementation arose. Due to external circumstances, a long break was taken from implementation, and upon return to the system Iteration 1 was arbitrarily ended and Iteration 2 began with mostly the same goals. The Django User Profile system was discovered in the course of Iteration 2, which led in turn to the discovery and implementation of signals (specifically the post_save signal) and the architectural revisions to account for the idiosyncrasies causes by subclassing DjangoUser. Test-Driven Development was *invaluable* at this stage. Because Model objects were the first to be constructed, there was no other method of validating their correctness besides test cases generated as part of TDD.

Unfortunately, the practice of TDD for this system (especially for early iterations) bore little resemblance to normal experience with TDD. The process of writing a failing test case was fairly straightforward after becoming familiar with the Django-specific abilities like new asserts, but the time between red-light test cases and green-light test cases was on the order of hours instead of minutes. As repeatedly mentioned, Django has very specific ways it needs things done, and very specific orders it wants them done in. While the Django community site offers a well-written tutorial and extensive documentation, it is still a *very large* framework. It was extremely difficult to take the "magic" methods and procedures from the tutorials and generalize them to application for this project. Failing that, it was necessary to learn how things worked before they could be made to work. This involved assimilating tremendous amounts of information to accomplish tasks that looked simple in tutorials.

The power of the Django Model layer, and its abstracting of the database, actually made this process more difficult. With the benefit of hindsight, it is clear that Django Models are very direct analogues to pure SQL—they can be best thought of as Python syntax to make SQL calls. Unfortunately, that very important fact was not clear from just the tutorials and high-level documentation. This resulted in attempts to mesh knowledge of what should work in Python with reality of what wasn't working in Django. Additionally, Django's highly-specific ideas of what order things must be done were troublesome here as well. Notably, definitions of primary and foreign keys must be done not only in logical order for the sake of the database, but also at specific moments in the Python object lifecycle, which is compounded by the fact that relevant details of that lifecycle are "conveniently" abstracted by the Django object managers. Once a conceptual design is formed about exactly what the framework is doing, it is much easier to deal with. Forming that mental model for the first time is a major hurdle, however.

This lengthy process of trial and error lasted throughout iterations 1, 2, and 3, as well as most of 4. One of the side effects of such warped TDD is that time technically spent in construction and labeled as such was really sandbox time. Although the work was done in production and test code, a great deal of it was figuring out how to use the framework effectively. This skewed construction times too high at first. A task recorded as taking one hour might have only taken an experienced Django developer ten minutes or less. An additional side effect was the practical reality of doing code experimentation within the production code. Ideally, sandbox work would be done in a completely separate file. The problem is recreated in the sandbox, a solution is developed, and context switches back to production code and the sandbox is deleted. In what quickly became a large system, however, it because increasingly difficult to recreate production problems from scratch after every context switch to the sandbox. In response to this, a more formal approach to the sandbox was developed, which will be described later in the report.

The end of Iteration 2 saw the creation of the first basic Views for the system, namely the login and logout views. By differentiating users at a framework level via Django Permissions, custom Views could make use of decorators provided by Django that allow filtering via permissions. Valid users are allowed to continue in the View, while invalid (unauthenticated or unauthorized) users are redirected to a login page. Basic use of these decorators Iteration 2 was followed by more advanced use in Iteration 3, which fleshed out additional Views including an early Reference model. At this point in construction, Views dealt with hand-made HttpResponse objects containing simple strings to validate that the views were functional.

Iteration 3 also saw the setup of the built-in Django admin system. This collection of modules, which is an optional part of the framework (referred to as one of many packages of “middleware”), offers a web interface to the database, allowing a user to interact with Models through a GUI instead of solely on the command line. The Django admin system is flexible and powerful but also requires a non-trivial amount of setup to make it appropriate for systems with complex behavior (like this one), so it did not entirely supplant command-line tools. Making it usable (and useful) required an additional module in the GWAAP system as well as overriding a few key magic methods, notably the __unicode__() method for several of the Models that needed to be accessed through the admin interface.

Iteration 4 was focused on the Comment and Vote models as well as the views to interact with them, plus two additional views for GwaapUsers to get a list of all applicants in the system as well as details on individual applicants.

At this point in construction, the system had most of its major functionality available from the command line or admin interface. Most Models were defined and working properly, and basic raw-text Views were available for several critical system functions. The functionality of the system was well-tested (around 100 test cases at that point) and separate paths for GwaapUsers and Applicants existed. Following iterations would add the Template layer and focus on several aspects of the user experience.

## Iterations 5 and following

Iteration 5 was the first iteration to include the development of components for the Template layer.  Using previous experience as a guide, effort estimation was handled differently from the previous iterations.  15 components were identified for construction in iteration, which seems like an overly large number except for the fact that the Django template layer includes an inheritance API.  Base templates can be defined once and child templates can simply inherit from one or more base templates and only override their specific area of functionality.  This was accounted for in iteration planning by putting a large percentage of the expected effort on the first (base) template, then assuming that child templates would take less time.  This approach yielded fairly accurate effort estimation, especially considering that this was the first iteration for an entirely new type of component.

The architecture for the template layer makes use of three levels of inheritance hierarchy.  The first level consists solely of the base template.  All other templates are expected to include this template in their hierarchy, so it was a natural place to put things common to every page in the web application.  This includes the expected HTML skeleton common to any page, as well as the linking of stylesheets and JavaScript.  To speed development, a collection of CSS and JS boilerplate files called Bootstrap was used.  Bootstrap defines several CSS and JS classes that elements of the DOM can be marked up with to enhance their presentation (and in some cases their functionality).  It is a popular and actively-maintained package with good compatibility among its own classes and between different browsers.  Because the development of a new stylesheet is not only time-consuming and error-prone but also outside of the direct scope of the project, it was determined that the use of Bootstrap was preferable to an entirely hand-made stylesheet. The project does include a set of custom styles to extend and modify some aspects of Bootstrap, but these are just that – extensions, not a standalone style.

The Bootstrap stylesheet and JavaScript files are linked to by the base template, giving all child templates direct access to their markup. The base template also outlines the basic structure of the HTML, including <head>, <body>, message, and comment blocks (the purpose of the latter two is described later). By handling this all on the base template, most of the non-page-specific markup only had to be written and debugged once. After this component was declared stable, all child templates could assume that their inherited markup was stable, leaving those components to focus only on their own unique functionality.

One of the major ways this technique proved valuable was in the use of the Django message middleware. Views are able to call methods of an automatically-provided Message object to add short messages intended to be displayed to the user in response to important actions or conditions. Messages passed in this way are added to a queue of messages available to the template layer the next time a template is rendered. The message queue can then be iterated over (which also clears it) and the messages can be output to the final HTTP response. To make this process as painless as possible while coding, some additional message-handling code was added to the base template. This block of code is run every time the base template is rendered (which is every time that *any* template is rendered in the system). Inside, the template iterates over the message queue, detects what type of message has been passed (“success”, “info”, or “error”), creates an HTML <div> element with appropriate coloring, and renders each such div in a standard format and location on the screen along with a small “X” to dismiss the message. This technique, called “toast” windows in many systems, is a convenient way to keep the user apprised of their progress through the system and the results of interacting with various parts of it. More importantly, it provides a way to give user feedback without having to design extra, explicit views for simple messages.

For example, consider the case of an applicant uploading a document like their letter of intent. There are at least two exit states: success, wherein the document was successfully uploaded and now resides on the server, and failure, wherein *something* went wrong and the document has not been uploaded. Furthermore, there are several ways to exit with failure. The user could have attempted to upload a document in an unsupported file format, could have submitted a form with no document attached at all, or could have attempted to upload a document to an application flagged by an administrator. In any case (success or failure), the user needs some way of knowing what happened. Without feedback, they simply get redirected to some page with no certainly of whether their action had the intended effect or not. One way many systems handle this is a simple success or failure page with a message and a link to the next reasonable page (the account page on success, the upload page on failure, etc.). This is reasonably effective, but has at least two drawbacks: it involves an extra click for the user, and in the specific case of this framework, requires extra View and Template code to be written for each and every failure or success case throughout the entire web application. This would mean an explosion in test cases and View methods even if a single template was developed using variables to populate its contents. The toast window system was a natural answer to this, and tying the toast messages to the base template means they are always available to any view at any time.

In short, the base template gives a consistent look to the application, as well as consistent availability of CSS classes and template blocks for structuring page content. However, a second level of the template hierarchy was devised to differentiate between the faculty portion of the application and the application portion. Even with a consistent UI, not all parts need exactly the same functionality. For instance, clicking on the "Home" button in the navigation bar should take faculty users to a different page from applicants. Similarly, the contents of the navbar should be different for the two classes of user. This behavior could have been implemented on the lowest level of templates, but it would have required a great deal of code duplication which would be more susceptible to mistakes as well as more painful to change if site-wide changes were required.

With all of the above functionality in place, the third and lowest level of the template hierarchy is occupied by the child templates defining page-specific UI elements. This includes everything else from form fields to the display of applicant details. At this level, the Django Template layer is powerful, but no so powerful that application logic can be put in it. This is one of its strengths. The main control structures in the Template layer are an if/else block and a for block, along with a brace notation that tells the template parser where to look for and insert variables. Variables are passed to the template layer as a Python dictionary. Whenever the template parser encounters a variable, it checks to see if the dictionary contains that string as a key, and populates the variable block with those contents if so.

The parser is capable of following limited dot-notation relationships as well. This is used in the templates for this system when an object (usually representing some Model passed from the view) is placed directly in the dictionary, as is typically the case with collections that need to be iterated over by the template. For example, in the “display all applicants” View that faculty evaluators have access to, a collection of all applicants currently active in the system is placed in the dictionary as a QuerySet (which is essentially a Python list wrapped in some extra Django functionality). The template iterates over each entry in the QuerySet to get one applicant at a time, then uses dot-notation lookups to populate specific holes in the template for each row in the table.

The other notable bit of advanced template usage for this project is the idea of template includes. These are somewhat like template inheritance but from a different perspective and usually with a smaller scope. When include blocks are encountered during template parsing, the parser loads them from the filesystem and parses them in the same context as the calling template, optionally with a reduced or renamed set of variables. Whereas inheritance is used when the framing aspects of a page need to be the same, includes are used when small parts need to be consistent within pages or sections that are generally different from one another. In this project, includes were used to display the different fields of recommendations in a consistent manner. Recommendations for applicants include five different fields asking the recommender to rank the applicant. Although the names for each field are different, the ranking levels are the same for all fields, and the display for them should be consistent. To achieve this, the five fields were placed in a table in the page template, with holes in the template for the contents of the fields. Populating each hole required a 6-level if-statement to account for the various possible responses that could be pulled from the database. Because that logic was consistent among all five of the fields (and verbose), it was extracted and placed into an include. For each row in the HTML table, the relevant variable was passed to the included code which in turn filled the hole with the appropriate response. The benefits of this were that the logic only had to be defined once (and therefore could be changed in a single place if necessary), and that the page template was not cluttered by five large, identical if-else blocks.

The most significant component pair of Iteration 5 was the View and Template for the "View Application" use case. This use case represents applicants' ability to check on the completion of the various parts of the application document and to take action to complete it as necessary. The template for this page consists of an HTML table with 7 rows in addition to its header. Four of the rows are direct representations of the status of their database fields. "Complete" fields yield a green cell with a check mark, "Not Applicable" fields yield a grey background with the text "N/A", and all other database contents yield a red cell with an "X" icon. If the database field is set to "Contact Administrator", this page will also yield an error message with that text. The bottom two rows of the HTML table represent the status of the applicant's resume and letter of intent uploads. These cells are considered complete if a valid document has been uploaded, and incomplete if not.

The other row represents the status of Recommendations. Because an applicant must have three references fill out these recommendations, the conditions for determining this status are more complex. If the applicant has not yet issued three recommendations, the status cell is filled with the incomplete contents. In addition, a third “Actions” column will have one button that yields details on why the cell is incomplete, and a second button that allows applicants to specify another reference. Once three references have been specified, the status column changes to a special “pending” cell with an orange background and the icon of a clock. The details button yields information to the effect that while the applicant can take no further actions, the cell cannot be marked complete until all references have filled out their recommendations. The applicant cannot specify additional references at this point, although in the event that a reference refuses to fill out a recommendation, an administrator can delete one of the incomplete references thus allowing the applicant to specify another. Once all three references have completed their form, the cell changes to green and the row is marked complete.

This page in particular necessitated more complex interaction between the View and Template layer than for most other pages. Specifically, dynamically determining and populating not only the colored status cells but the action buttons required a significant amount of HTML markup in the Python code for the Views, which then had to be placed correctly in the final HTTP response so that the DOM could render the contents properly. Two techniques were developed to help cope with these problems. First, the *highly* error prone generation of dynamic HTML elements was placed in a factory method accessible by the View. The factory method takes the custom strings needed to generate and differentiate certain elements, then plugs it in to a long string amounting to a complete HTML element. This allowed for a single place to assemble and test the HTML code (made more problematic by having to constantly open and close three different Python quote types) and then allowing one well-tested factory to generate all subsequent HTML elements of that type.  Furthermore, the return value of the factory method could be stored as a Python variable, making it much easier to deal with in code than a large, multi-line string literal.  The second technique was developed to handle the arbitrary JavaScript that was necessary for some of the components to operate properly. Because the JS on the page needs to be rendered in an area reserved for the base template, and because the exact content of that JS could not be abstracted for all pages, a special block was created on the base template.   The block, titled "extra_onready_js" could be filled by any view that needs a place to put its own specific JS code.  In the base template, this block was then placed inside a <script> tag, and specifically inside a pre-defined method guaranteed to be run on each page load.  If this block is empty (as it is for most pages), it does not slow down the page because it is effectively a blank line inside a minimal function.  On the other hand, pages that need to define their own JavaScript can do so straight from Python.  This technique was used on the View Application page to enable the mouseover behavior for the Detail buttons, among other places.

Iteration 6 added some additional polish to the system while tackling two new design problems.  The first of these was the implementation of the complete recommendation/reference system.  As mentioned, it was determined that this system should not require users to make account with the system.  Nonetheless, it is necessary that users still be authenticated (in the non-Django sense) and that the system offers an effective user experience.  For instance, references need to see the name of the person they are recommending on the page to prevent confusion especially in the event that they are asked to serve as a reference for more than one applicant through the GWAAP system.  At the same time, there needs to be a high degree of certainty that users cannot somehow find their own recommendations in the system and fill them out. It is also desirable that users cannot find out who else is applying and who their references are by searching through URIs in a systematic manner. Exposing URIs comprised of the primary keys of applicants and the primary key of the Reference objects in the database are acceptable if there is some sort of password protection or other validation between users and the display of the recommendation form, but not acceptable if the reference form is immediately available upon visiting the correct link.

The solution used in this system is to create a unique identifier string for each Reference object in the database at the time it is first saved. Next, a base URI is exposed by the Django dispatcher to catch attempts to access Reference objects. A regular expression extracts the unique identifier string, pulls the matching Reference object from the database, then populates a form template and displays it to the recommender. This method achieves all goals necessary for the system. First, it allows recommenders to interact securely with the system without creating an account with it. Second, if the key space of the unique identifier string is large enough, it prevents unauthorized users from accessing Reference objects that they shouldn't.

Using a hashing method to generate the unique identifiers is one possibility, but the chance exists (however small) that two different Reference objects may hash to the same value, thus creating a conflict when the system tries to pull exactly one matching object from the database and receives two. To prevent such collisions, new additions to the table of Reference objects would need to check to make sure that no other Reference objects existing in the database already share that unique identifier. This means the process of inserting a new

Reference object will have to include a linear search using this method. Since that is the case, a simpler method of generating a random string of ten upper- and lower-case alphanumeric characters is used, checking to make sure that string does not already exist in the database, and then using it as the unique identifier. This is still a linear search but has the benefit of increasing the key space for the same number of characters while not having to depend on a hashing algorithm.

Finally, this unique identifier is sent in the email to references already included in a link to access the reference forms. This means that potential references have only to click a single link in their email and get direct, secure access to the recommendation form. Because the key space for this string is 10^62 (ten characters randomly selected from all numbers and upper and lowercase letters), the chance of brute force access to even a single random reference in the database is vanishingly small; the chance of brute forcing access to a *specific* reference, smaller still.

The second, albeit smaller, design concern in this iteration was the final page of the applicant area of the system – the "Home" page, or Applicant Status page. This primary feature of this page is a graphic indicating how far along the applicant is in the application process. The applicant will always be at one of six linear milestones, which are held in the database as small integers. The human-readable milestones these integers represent are declared as a tuple of values in Python, each value being itself a tuple pairing the integer value with a human-readable explanation of the milestone. Using this method, the Django admin interface is able to populate itself with the string values for convenience, while the database only has to hold the numeric representations.  This fact is used on the Applicant Home page to determine the size of the progress bar graphic that represents which milestone the applicant is at.  Instead of pulling the string from the tuple, the progress bar pulls the integer value, multiples it by a percentage, and then uses the resulting value directly in the CSS definition of the progress bar width.  Additionally, the template includes six holes to indicate whether the corresponding help text should be bolded or not.  Since the template parser renders nonexistent variables as blanks, the variable named for the integer value in the database includes the markup to make the text bold, while the other variables render nothing and therefore do not have their corresponding text bolded.

## Sandboxing complex system functionality

Although it is not part of the linear flow of any iteration, PCSE describes an additional implementation block called the "sandbox."  In this block of time, context is switched a blank, unconnected module in order to test a new or poorly understood technique without compromising the integrity of production code.  Sandbox time can occur at any point in any iteration, although it usually occurs during construction to address problems encountered while coding.  When the source of confusion is a syntax error or language issue, the blank-slate approach of the sandbox is highly desirable.  However, there were many times during the implementation of this system where problems arose precisely from the implementation of the system itself, or from the interaction of various parts of the Django framework.  In these cases, a typical sandbox approach was infeasible after about halfway through Iteration 2—basically, once the system had grown too large to accommodate reconstructing relevant parts for the sake of the sandbox.

To address this concern, a source control solution was devised. Whenever there was a need to break the code out into a sandbox, the system was briefly refactored into a stable state. This state was committed to a Git repository. Next, a completely separate instance of Eclipse was opened (to prevent namespace conflicts that arose from having two Eclipse projects with the same name in a workspace). Finally, the stable version of production code was imported into the sandbox instance of Eclipse, and sandboxing could be performed on the full system without violating production code. Lessons learned could be immediately applied back to the production code and another commit could be made, then the sandbox could be entirely wiped out without loss.

# Chapter IV

## Validation

To determine that the system solved the problem as outlined, two major methods were used.

### Deriving system components from use cases

First, the system was compared at every stage to the use cases. Each proposed component was derived (either directly or indirectly) by linguistic analysis on the use cases. Given the line from the “Submit Comment” use case, “The faculty member submits a comment,” for instance, three components could be derived—Faculty Member and Comment as Models, and Submit Comment as a View/Template. Other components were derived from non-functional requirements. For example, the base template component was derived from the non-functional requirement that the interface be unified among the several pages of the application.

In addition to deriving components from the use cases, a second pass through the use cases was made to determine that every action described in the use cases could be accomplished through at least one portion of the system. This was a formalization of deciding what use cases were available through the built-in Django admin functionality, and which were only possible through custom Views. For example, the use cases specify that both applicants and administrators must be able to update applicant profile information. Because administrators have access to the Django admin interface, this use case did not require a custom solution. However, because applicants do *not* have access to any portion of the admin interface, custom View and Template components were derived to satisfy this requirement.

One use case described visually the relationships between actors and their access to the system. To use the example above, both Administrator and Applicant types have access to the "Update Applicant Profile" use case and this is shown by a line connecting their respective stick figures to a circle containing the name of the use case. Validating that these requirements were met was a matter of ensuring that each user type could be created from some mix of permissions granted through the admin interface or by signals in the system at object creation time. For example, Faculty Members can be generated by creating a base User object and adding Comment permission, while Committee Members are generated by doing the same plus adding a Vote permission. Administrator-level users can be created by adding DjangoUser objects to the system then having a superuser give them some additional system-specific permissions.

The final method of validating the system was by comparing expected outputs expressed in the use cases to the actual outputs of the system. In some cases, this was as simple as validating that the text displayed in the use cases matched text displayed on screen. For use cases including sample graphics though, this was a less-rigid process of evaluating the visual output of the system and ensuring that it matched (at least on a conceptual level) the sample graphics from the use cases.

### Regular stakeholder meetings

The second method of system validation was by regular meetings with primary stakeholder for the system. This allowed for quick feedback loops early in the design, as well as requests for design changes as necessary. As the implementation of the system matured, the tight feedback loop and ability to combine or modify use cases as appropriate became increasingly more valuable. After approximately Iteration 4, meetings also included system demonstrations to ensure that the system was progressing according to stakeholder expectations (both functional and non-functional).

## Verification

By virtue of constructing using Test-Driven Design, a full suite of test cases was developed for the system. Approximately 180 test cases were devised to test all three layers as thoroughly as possible.

Model testing was fairly straightforward. Because Django abstracts the database tables and rows into Python objects, these objects were directly available for test case asserts. The intermediary of the framework meant that standard Python idioms could be applied to data that was ultimately backed by a database. For example, test cases had access to Python built-in methods like len() for checking the number of rows returned for particular queries, etc. As the Model layer is primarily a data definition layer in Django, most of the model-related test cases checked for the existence of appropriate fields, correct foreign key relationships, correct data storage, and similar issues. It is impossible to overstate the value of the Django framework in converting database contents to testable Python objects.

View testing was slightly more complex, since ultimately the View layer takes HttpRequest objects as inputs and returns HttpResponse objects as outputs – essentially, glorified string values.  Again, the Django framework provided several facilities to ease this process, though.  First, HTTP header fields and status codes are accessible through dot-notation on the relevant Django objects.  This eliminates a great deal of string parsing to get at key information.  For instance, some test cases were designed to check for the existence of expected status codes (i.e. a test case purposely calling for a View that doesn't exist should get a 404 status code, etc.).  Django offers these status codes as a status_code attribute off of the returned HttpResponse object.  Even more important is Django's provision of a Client object to interact with views as if testing against a (very abstracted) web browser.  Using the Client object, test cases are able to make requests of the system as if they are querying a server.  This is useful for ensuring that proper URIs exist for the relevant Views and that the Django dispatcher is serving them as expected.  Additionally, it allows for both GET and POST requests on Views, the latter of which taking a Python dictionary as variables for the POST request (for instance, form contents for testing that the Views handle form data correctly).  The Client object also allows for simulated login, which was invaluable for testing that the correct types of users could access the correct areas, or more properly that the wrong types of users could not.  Finally, the Django testing suite offers a few extra asserts (compared to the standard PyUnit asserts) including the assertContains() method.  This is a relatively simple method that searches through the text content of an HttpResponse object for a specified string.  Though it is conceptually simple, it is powerful in practice, especially in conjunction with the testing method devised for the template layer.

## Unit testing for the template layer

One of the methods used to validate the template layer was the comments block. In the base template, an additional block called “comments” was included inside an HTML comment on the template. Because the Django parser ignores HTML comments (Django templates have their own notation for comments), the parser can populate an HTML comment just like any other block in the template. In this way, the actual display of the page to the user can be decoupled from the need for certain test strings to appear on the page. For instance, one test case tested for the presence of the string “View Applicants” on the ViewApplicants template. This way the test case could validate that the correct template is loading. Then, that string was inserted inside the comments block on the appropriate page to ensure that it always appears in the template, even if the display of the page is changed such that “View Applicants” no longer appears as such to the user. Furthermore, through the use of Django template variables within the comments block, arbitrary comments can be added at runtime by the View layer itself. In another template, an “extra_comments” tag was added to look for a string by that key in the dictionary. The View can then add comments as necessary to verify that the template is correctly pulling data from models, etc. For example, a test case was devised to test that the template layer correctly populated an HTML table with the proper number of rows pulled from the database. In the view, a simple counter was incremented each time another model was placed into a row in the table. Then, the counter was inserted into the extra_comments block along with an identifying string. Finally, the test case can check for the presence of the correct counter value after the identifying string, thus testing that the table is populated with the correct data.

Obviously, the technique is unsuitable for sensitive data that should never appear in the template layer at all, because it is still visible to users who view the source of the page. Still, it works well for digesting information that is already available on the page, but that would be extremely difficult to extract by simply analyzing the text content.

## Note on system functionality tested externally

Four aspects of the system were assumed to be tested elsewhere. First is the Django framework itself. In fact, Django comes with about 300 test cases included that can be run as a standalone test suite or as part of an application's individual test suite. All test cases pass on the development system. Second is the CSS and JavaScript components that make up Bootstrap. Because Bootstrap is actively in use and regularly updated, this component was assumed to work properly given correct HTML markup. Third is the web browser on which the application will be run. Because the system takes advantage of no browser-specific functionality (limiting its markup to a subset of cross-browser compatible HTML), it is assumed that the system will work in all major web browsers. (Note that testing *was* done to ensure that HTML tags were syntactically correct, that anchors pointed to the correct links, etc.). The fourth and final externally tested component is the database. It is entirely beyond the scope of this project to validate that the RDBMS backend is working properly, but it is also very safe to assume that it is.

# Chapter V

## Lessons learned in Django

The process of learning Django while developing this system made it somewhat unique compared to implementing it in a pure subset of Python. Thanks to the exceptional documentation and community of the framework, most of the system exhibits best-practices for major functionality. However, the User authentication system could be improved. There is no single standard way in Django 1.3 (the version used for this project) to handle additional information tied to users, especially subclasses of User. The most idiomatic way according to the Django community site is the UserProfile functionality, but this is designed specifically for applications that directly instantiate DjangoUser to represent users on the site. Because of the creation of the two child classes, several other workaround had to be devised to handle the idiosyncrasies that arose. No single workaround for these problems exists or even seems to be favored (for Django 1.3 at least). There was a fair amount of trial and error involved, along with implementing the system around existing design decisions for the User classes. If there is one area of the system that could be improved by starting over, this would be it.

One of the biggest early obstacles was interacting with the database. Django does an exceptional job of handling database interactions for the developer – so exceptional, in fact, that it is easy to make poor design decisions by applying object-oriented reasoning to what is in reality a database schema. The takeaway from this is to enjoy the database convenience that the framework offers, but to always remember that it is in fact a database just under the surface, and to think in terms of SQL, not object orientation, when defining models.

Testing FileFields for models was very challenging to handle in a non-trivial way. For this project, files are tested by actually creating dummy files and passing them as POST data to Views during testing, then asserting that the presence of the file exists in the correct place after the View runs. The problem with this method is that it actually creates files on the filesystem. In practice, this meant that once those test cases were determined to run correctly, they were commented out until the file system could be cleaned up. Every time they are run they require manual cleanup afterwards. Some sort of file mockup package would have been helpful for this testing. At least one such package appears to exist for Python, although time did not permit its use in this project.

Finally, it was interesting (although not necessarily Django-specific) to have to interact with a legacy system during this project. For instance, Applicant Profiles have a field called "ENTER_QTR", referring to the "quarter" they begin their studies. Auburn has not been on a quarter system for some time now, but the name persists in the old GWAAP database. To ensure that the new system is interoperable, the field in the new system shares the same (outdated) name. It was interesting to have to make a known "bad" decision in the new system in order to coexist with the roots in the new system.

## Applying PCSE to web development

One process problem arising from the use of the Django framework is the issue of how to track time spent in the sandbox versus time tracked as construction. Typically, any time spent writing test code or production code would be counted as part of the construction phase. However, a significant amount of time in this project was spent trying to perform simple tasks in production code using Django idioms.  As described earlier, this could cause extremely long periods between failing and passing test cases.

Since construction *technically* consisted of writing production and test code during that time, the choice was made during iteration 1 to document this time as construction time. Within the scope of this project, that decision worked for estimating effort in subsequent iterations.  However, a problem would arise if data from this project were used to estimate effort in a second Django project.  Because PCSE construction data tracking was not granular enough to distinguish “learning-Django-construction” from “normal-TDD-construction”, using this data again would produce skewed effort estimation since the learning curve for Django would be lessened.  This would likely hold true for any project heavily based on a framework that was being used for the first time (or even a project in a new language).  For this reason, if the project were started again today, time spent learning Django (even time spent technically writing production code or tests) would be recorded as sandbox time.  That would enable accurate time estimation for subsequent iterations by combining sandbox and construction time.  In a subsequent project time effort estimation could be based only on construction time.

Testing of the template layer could also be expanded for the development of other web applications.  Although the content of the template layer can be validated through PyUnit using the comment block technique described above, there are also in-browser testing frameworks designed to pick up where testing in this system leaves off.  Specifically, these are used for testing actual DOM elements and even CSS markup.  For a higher confidence level in the user experience with the templates, one of these tools could be used.

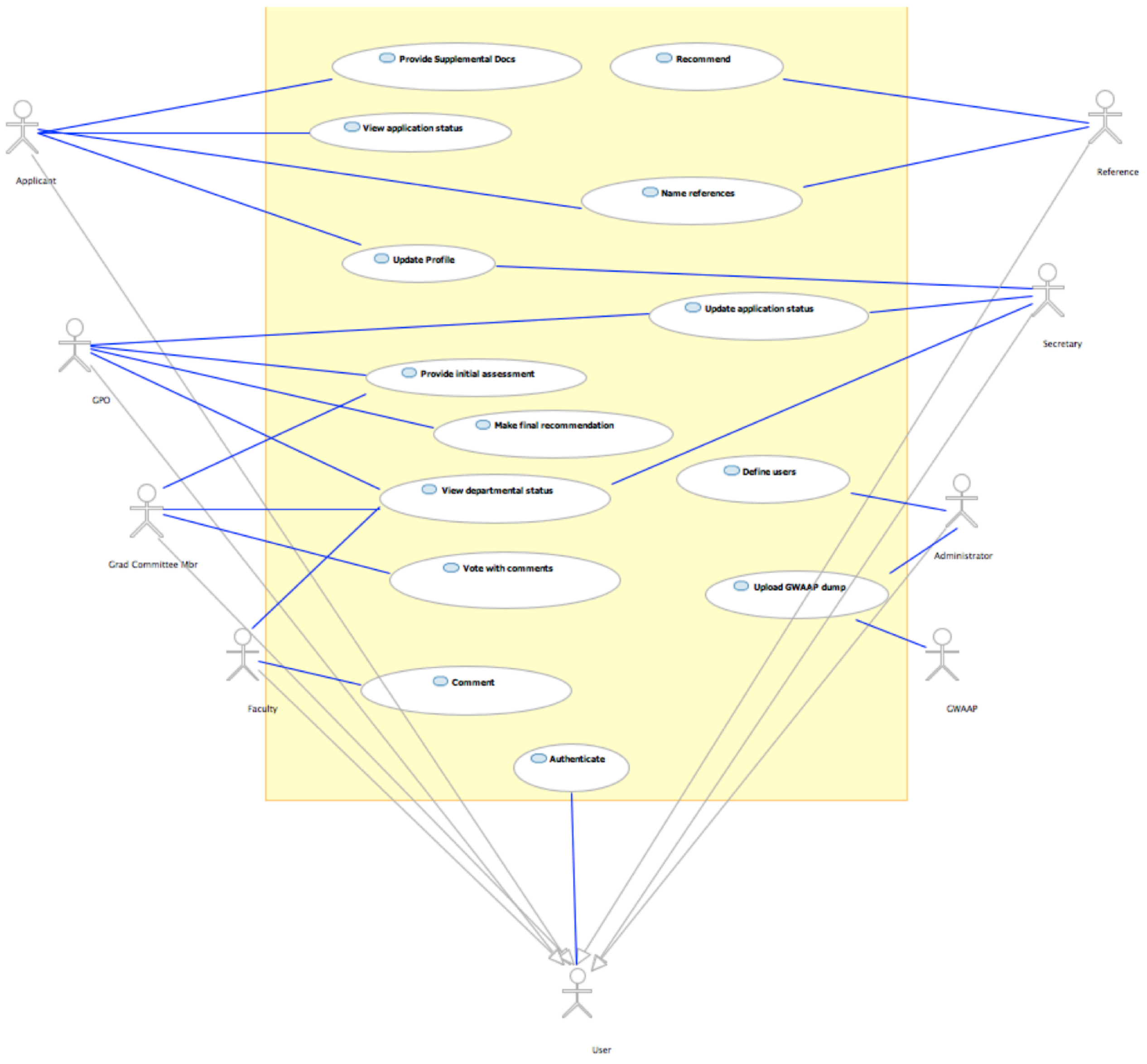
## Future work

At least three expansions to the current system could be made to improve functionality. First, the current file server is intended for debugging only and should be replaced with a server (or at least a View) with permissions tailored more properly to its use. In its current configuration, files uploaded to the system (including all resumes and letter of intent) are publicly available if their correct URI is retrieved. Second, some additional verbosity in detail buttons and error messages would be welcome. Although the system does its best to cover the most common errors, it is still possible to give it such malformed input that it will throw an "ugly" error, like an HTTP 404 or 500 response. Most of these errors could be eliminated through the power of the Django dispatcher if desired. Third, an automated method for the creation of applicants through the GWAAP dump files is desirable. Ideally, the system could expose a URI that accepts POST data consisting of a CSV file from the Graduate School and automatically creates new Applicant objects in the system and emails their new owners with setup details. It would also be easy to create a basic command-line tool for this in Python.

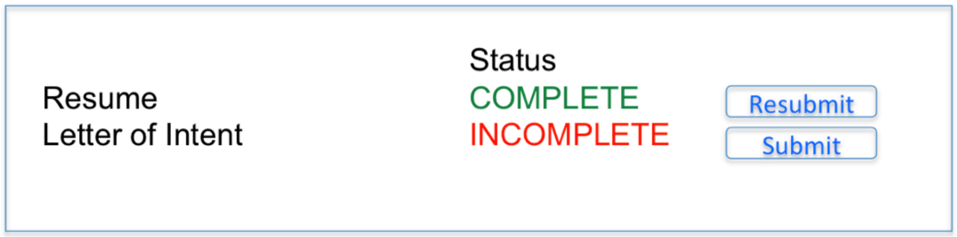
Finally, a major expansion of the system would be to modularize or abstract its CSSE-specific details so that it could be used by other departments. The Django framework already makes use of a settings file where deployment-specific details of the application can be stored. With some major refactoring, the system could be rewritten to take advantage of similar functionality and be expanded such that migrating it to a new department would require only a custom settings file. This would mean the main aspects of the system could be held on a single authoritative server, enabling all departments using the system to automatically benefit from updates to the system while still maintaining their department-specific behavior.

# Appendix A

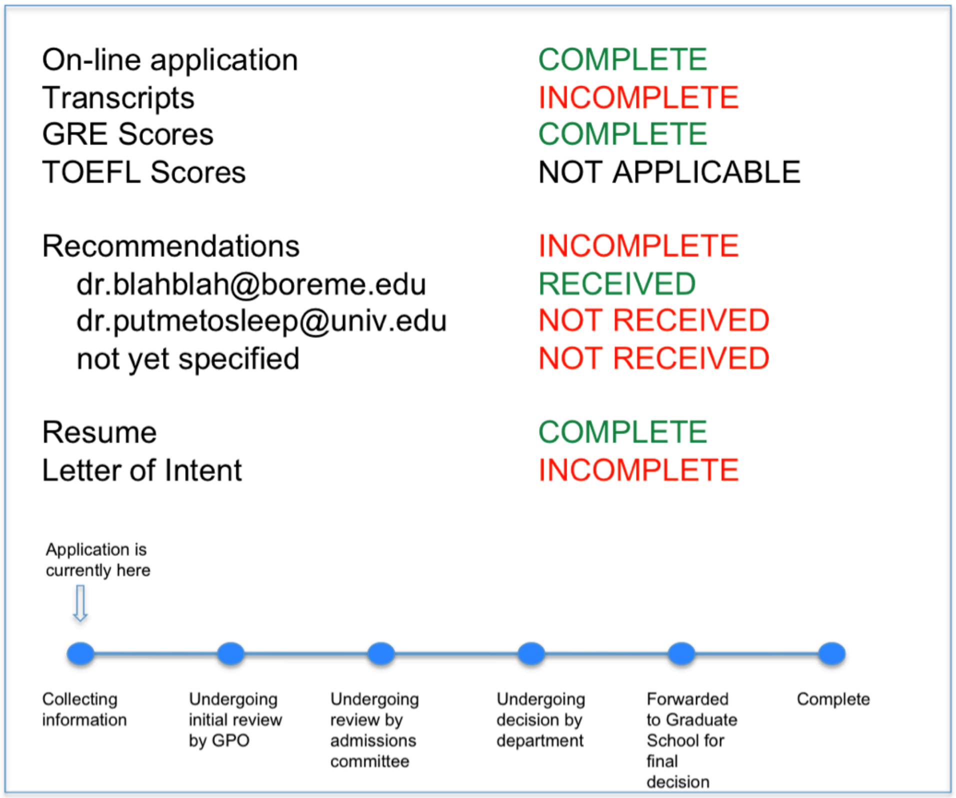
## Original Use Cases


Provide Supplemental Docs
Recommend
View application status
Name references
Update Profile
Update application status
Provide initial assessment
Make final recommendation
View departmental status
Define users
Vote with comments
Upload GWAAP dump
Comment
Authenticate
Applicant
Reference
Secretary
GPO
Grad Committee Mbr
Administrator
Faculty
GWAAP
User

1

| | |
|---|---|
| Use case name: | Provide supplemental documents |
| Primary actor: | Applicant |
| Supporting actors: | |
| Use case objective: | To allow the applicant to upload department-specific application information |
| Entry criteria: | The applicant is authenticated |
| Trigger: | The applicant starts the "Upload Supplemental Information" function |
| Basic Flow: | The applicant is informed of what department-specific information has been received by the department, and can submit (or resubmit, if already submitted) a resume or letter of intent. See notes below. |
| Success exit criteria: | Submitted documents are saved. The status of each submitted document is considered "complete." |
| Alternate Flows: | The applicant may abandon this function at any time. |
| Failure exit criteria: | No documents are saved. |
| Priority: | Ugrad: optional; Grad: optional |
| Notes: | The GPO sketched the item below during a client-developer interview. It was intended to convey content, not necessarily a specific layout or color scheme.   Status<br>Resume<br>COMPLETE<br>Resubmit<br>Letter of Intent<br>INCOMPLETE<br>Submit |

2

| | |
|---|---|
| Use case name: | View application status |
| Primary actor: | Applicant |
| Supporting actors: | |
| Use case objective: | To allow the applicant to see where his/her application is in the admission process |
| Entry criteria: | The applicant is authenticated |
| Trigger: | The applicant indicates he/she wants to view his/her status |
| Basic Flow: | The applicant is presented with 1) what application information has been received by the department, and 2) where the application is in the admission process.  See notes below. |
| Success exit criteria: | None |
| Alternate Flows: | None |
| Failure exit criteria: | None |
| Priority: | Ugrad:  mandatory; Grad:  mandatory |
| Notes: | The GPO sketched the item below during a client-developer interview.  It was intended to convey content, not necessarily a specific layout or color scheme.  |

| | |
|---|---|
| On-line application | COMPLETE |
| Transcripts | INCOMPLETE |
| GRE Scores | COMPLETE |
| TOEFL Scores | NOT APPLICABLE |
| | |
| Recommendations | INCOMPLETE |
| dr.blahblah@boreme.edu | RECEIVED |
| dr.putmetosleep@univ.edu | NOT RECEIVED |
| not yet specified | NOT RECEIVED |
| | |
| Resume | COMPLETE |
| Letter of Intent | INCOMPLETE |

Application is currently here
Collecting information
Undergoing initial review by GPO
Undergoing review by admissions committee
Undergoing decision by department
Forwarded to Graduate School for final decision
Complete

3

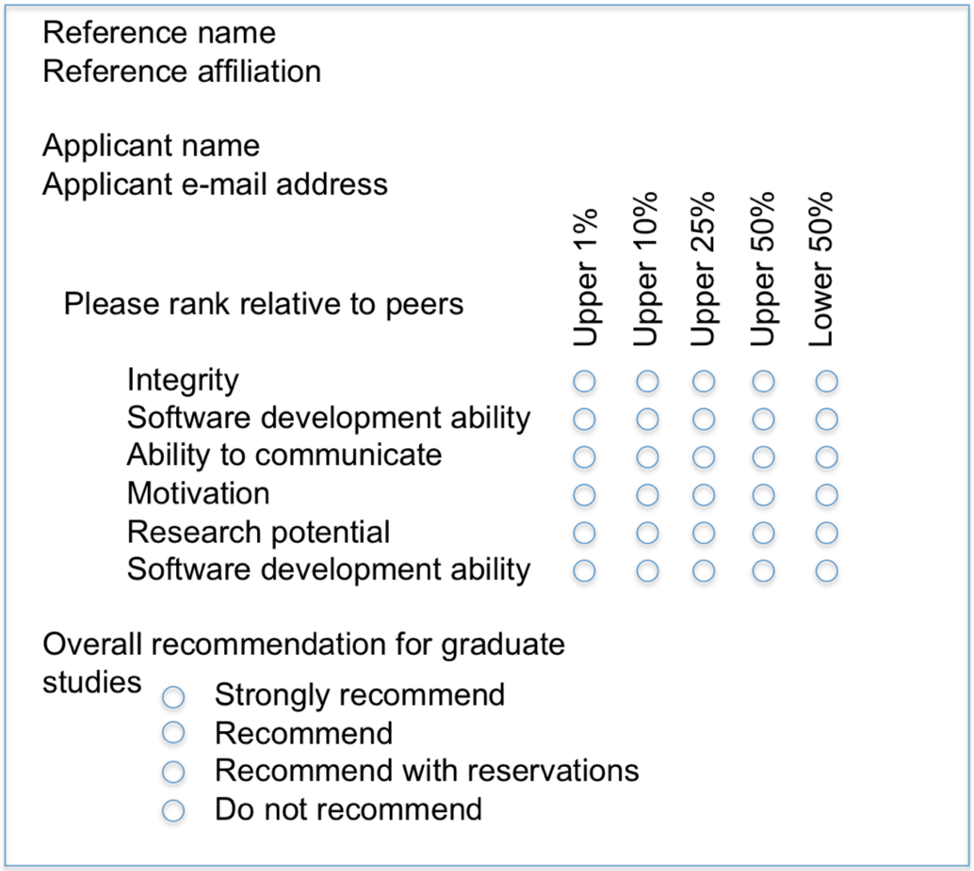
| | |
|---|---|
| Use case name: | Name references |
| Primary actor: | Applicant |
| Supporting actors: | References |
| Use case objective: | To allow the applicant to specify contact information for three references |
| Entry criteria: | Applicant has successfully logged in |
| Trigger: | Applicant starts "Name References" function |
| Basic Flow: | The Applicant specifies the name, affiliation, and e-mail address for three references. An e-mail is sent to each reference requesting that he/she provide a recommendation for the applicant. The e-mail contains 1) the applicant's name and 2) instructions on how the reference should log into the system to complete the recommendation form. |
| Success exit criteria: | 1) The applicant is informed that the reference has been added.<br>2) An e-mail has been sent to each reference |
| Alternate Flows: | Alternate 1 -- Incomplete reference information:<br>If the applicant omits a reference's name, affiliation, or e-mail, the system will inform the applicant of the missing information.<br>Alternate 2 – Illegitimate e-mail address:<br>The applicant is informed that no recommendations will be accepted from references whose e-mail address is an e-mail service such as gmail, yahoo, hotmail, etc. |
| Failure exit criteria: | Alternate 1: Incomplete reference information is saved for later editing, but no e-mail is sent to that reference. The applicant's status for providing reference information is considered "incomplete."<br>Alternate 2: An illegitimate e-mail address is saved for later editing, but no e-mail is sent to that reference. The applicant's status for providing reference information is considered "incomplete." |
| Priority: | Ugrad: mandatory; Grad: mandatory |
| Notes: | |

4

| | |
|---|---|
| Use case name: | Update profile. |
| Primary actor: | Applicant |
| Supporting actors: | Secretary |
| Use case objective: | To allow the applicant to update personal application information |
| Entry criteria: | The applicant is authenticated |
| Trigger: | The applicant indicates he/she wants to update his/her profile |
| Basic Flow: | The web app displays the applicant's data (see note below). Changes to editable information are saved. The secretary is sent an e-mail indicating an update has been made. |
| Success exit criteria: | Applicant data is current. |
| Alternate Flows: | Alternate: With the exception of EMAIL, no editable field is validated. EMAIL is invalid if it is empty, contains spaces, has more than one "@", or has no periods after the "@". |
| Failure exit criteria: | The applicant is informed that his/her e-mail address is invalid or blank. He/she is requested to correct it before being able to save updates. |
| Priority: | Ugrad: mandatory; Grad: mandatory |
| Notes: | The GWAAP fields below may be shown the applicant. A hypen ("-") indicates the field is displayed but cannot be edited by the applicant; a plus ("+") indicates that the contents can be changed by the applicant. |

| | |
|---|---|
| -FIRST<br>-LAST<br>-MIDDLE<br><br>+STREET1<br>+STREET2<br>+CITY<br>+PROVINCE<br>+STATE<br>+COUNTRY<br>+ZIP<br><br>+EMAIL<br>+TELEPHONE<br>-BIRTHDAY<br>-GENDER<br>-COUNTRY_BIRTH<br>-CITIZENSHIP | -REF_NUMBER<br>-DATE_APPLY<br><br>-ENTER_QTR<br>-ENTER_YEAR<br>-DEGREE<br>-MAJOR<br><br>-GRE_TAKEN<br>-O_GRE_V<br>-O_GRE_Q<br>-O_GRE_A<br>-O_Gre_W<br>-TOEFL_TAKEN<br>-O_TOEFL_SCORE |

5

| | |
|---|---|
| Use case name: | Recommend |
| Primary actor: | Reference |
| Supporting actors: | |
| Use case objective: | To allow a reference to comment on the suitability of the applicant for graduate education |
| Entry criteria: | Reference has been authenticated |
| Trigger: | A recommendation is started automatically upon entry. |
| Basic Flow: | The Reference fills out a recommendation form (see note 1) and submits it. |
| Success exit criteria: | 1. An e-mail is sent to Reference indicating the recommendation has been completed.<br>2. An e-mail is sent to Applicant indicating Reference has completed a recommendation.<br>3. The recommendation is saved and the application status is updated. |
| Alternate Flows: | Reference abandons the form without saving. |
| Failure exit criteria: | No information is saved. The applicant's status with regard to recommendations remains unchanged. |
| Priority: | Ugrad: optional; Grad: mandatory |
| Notes: | The GPO sketched the item below during a client-developer interview. It was intended to convey content, not necessarily a specific layout or color scheme.  |

Reference name
Reference affiliation

Applicant name
Applicant e-mail address

| Please rank relative to peers | Upper 1% | Upper 10% | Upper 25% | Upper 50% | Lower 50% |
|---|---|---|---|---|---|
| Integrity | ○ | ○ | ○ | ○ | ○ |
| Software development ability | ○ | ○ | ○ | ○ | ○ |
| Ability to communicate | ○ | ○ | ○ | ○ | ○ |
| Motivation | ○ | ○ | ○ | ○ | ○ |
| Research potential | ○ | ○ | ○ | ○ | ○ |
| Software development ability | ○ | ○ | ○ | ○ | ○ |

Overall recommendation for graduate studies
- ○ Strongly recommend
- ○ Recommend
- ○ Recommend with reservations
- ○ Do not recommend

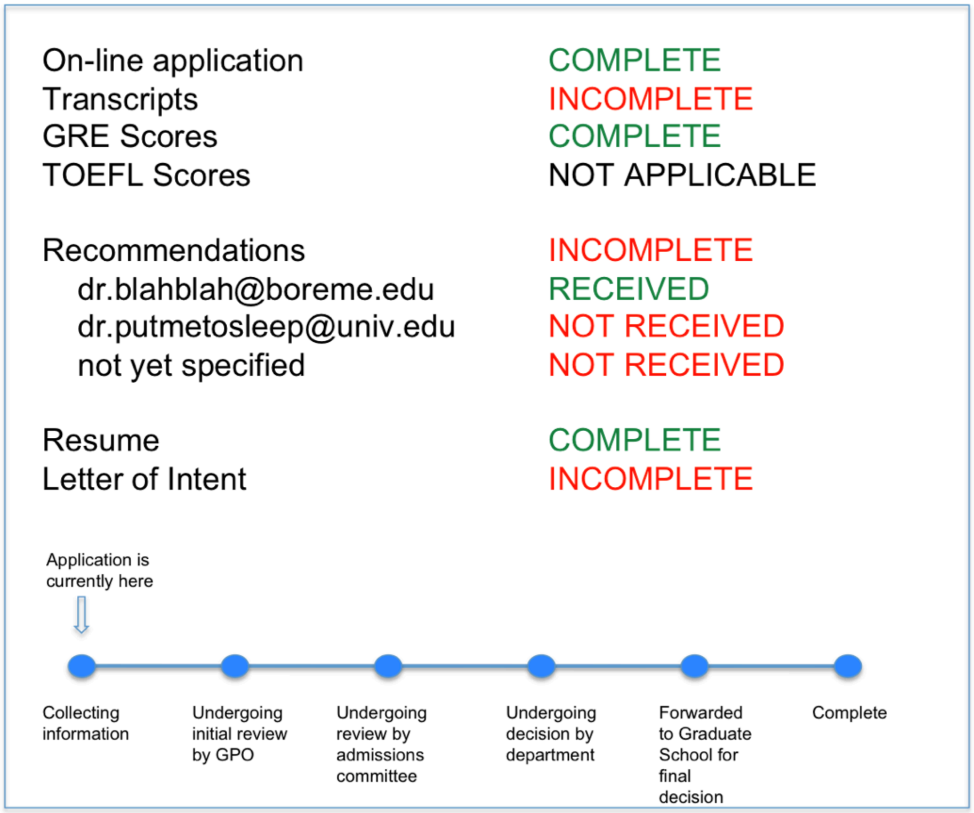
6

| | |
|---|---|
| Use case name: | Provide initial assessment |
| Primary actor: | GPO |
| Supporting actors: | Graduate Committee Members |
| Use case objective: | To allow the GPO to cast the first vote on the applicant |
| Entry criteria: | GPO has been authenticated |
| Trigger: | GPO starts "Provide Initial Assessment" function |
| Basic Flow: | GPO is shown the following information on the applicant: application information, recommendations, and faculty comments. GPO adds his/her own comments. GPO submits a vote on the applicant (see notes below). |
| Success exit criteria: | GPO comments and vote are saved. Graduate Committee Members are notified via e-mail that the applicant is available for their vote. The applicant's status is advanced to the next stage in the admission process. |
| Alternate Flows: | GPO may abandon the initial comments before voting. |
| Failure exit criteria: | No information is saved; no e-mail is sent. |
| Priority: | Ugrad: mandatory; Grad: mandatory |
| Notes: | Possible votes:<br>Accept<br>Weak Accept<br>Weak Reject<br>Reject |

7

| | |
|---|---|
| Use case name: | Make final recommendation |
| Primary actor: | GPO |
| Supporting actors: | |
| Use case objective: | To allow the GPO to document the final departmental decision on the applicant. |
| Entry criteria: | Graduate Committee Member has been authenticated |
| Trigger: | Graduate Committee Member starts the "Make Final Recommendation" function |
| Basic Flow: | The GPO is shown his/her initial vote and comments, as well as the vote and comments submitted by the Graduate Committee Members. The GPO casts a final vote (see notes below) with comments. |
| Success exit criteria: | The GPO final vote and comments are saved. The applicant's status is advanced to the next stage in the admission process. |
| Alternate Flows: | Graduate Committee Member may abandon the initial comments before voting. |
| Failure exit criteria: | No information is saved. |
| Priority: | Ugrad: mandatory; Grad: mandatory |
| Notes: | Possible votes:<br>Accept<br>Reject |

8

| | |
|---|---|
| Use case name: | View departmental status |
| Primary actor: | GPO, Secretary, Faculty Member, Graduate Committee Member |
| Supporting actors: | |
| Use case objective: | To allow department members to see where an applicant is in the admission process |
| Entry criteria: | The user is authenticated |
| Trigger: | The user indicates he/she wants to view status the status of an applicant |
| Basic Flow: | The user selects which applicant he/she wants to view. The user is shown 1) what application information has been received by the department, and 2) where the application is in the admission process. See notes below. |
| Success exit criteria: | None |
| Alternate Flows: | None |
| Failure exit criteria: | None |
| Priority: | Ugrad: mandatory; Grad: mandatory |
| Notes: | The GPO sketched the item below during a client-developer interview. It was intended to convey content, not necessarily a specific layout or color scheme.  |

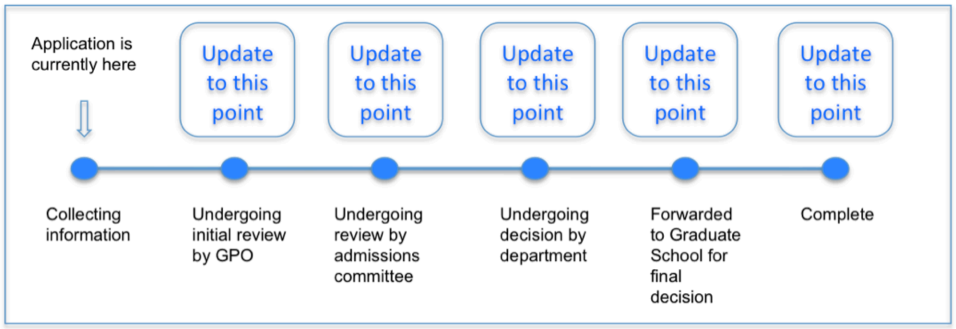
| | |
|---|---|
| On-line application | COMPLETE |
| Transcripts | INCOMPLETE |
| GRE Scores | COMPLETE |
| TOEFL Scores | NOT APPLICABLE |
| Recommendations | INCOMPLETE |
| dr.blahblah@boreme.edu | RECEIVED |
| dr.putmetosleep@univ.edu | NOT RECEIVED |
| not yet specified | NOT RECEIVED |
| Resume | COMPLETE |
| Letter of Intent | INCOMPLETE |

Application is currently here
Collecting information
Undergoing initial review by GPO
Undergoing review by admissions committee
Undergoing decision by department
Forwarded to Graduate School for final decision
Complete

9

| | |
|---|---|
| Use case name: | Vote with comments |
| Primary actor: | Graduate Committee Member |
| Supporting actors: | |
| Use case objective: | To allow the Graduate Committee Member to voice his/her opinion on the acceptability of an applicant |
| Entry criteria: | Graduate Committee Member has been authenticated |
| Trigger: | Graduate Committee Member starts the "Vote" function |
| Basic Flow: | Graduate Committee Member is shown the following information on the applicant: application information, recommendations, GPO comments, comments of other Graduate Committee Members, and faculty comments. Graduate Committee Member adds his/her own comments. Graduate Committee Member submits a vote on the applicant (see notes below). |
| Success exit criteria: | Graduate Committee Member comments and vote are saved. When at least two Graduate Committee Member votes have been cast, the GPO is sent a notifying e-mail and the applicant's status is advanced to the next stage in the admission process. |
| Alternate Flows: | Graduate Committee Member may abandon the initial comments before voting. |
| Failure exit criteria: | No information is saved; no e-mail is sent. |
| Priority: | Ugrad: mandatory; Grad: mandatory |
| Notes: | Possible votes:<br>Accept<br>Weak Accept<br>Weak Reject<br>Reject |

10

| | |
|---|---|
| Use case name: | Update application status |
| Primary actor: | GPO, Secretary |
| Supporting actors: | |
| Use case objective: | To manually change the status of an application. |
| Entry criteria: | The user is authenticated |
| Trigger: | The user indicates he/she wants to update an applicant's status |
| Basic Flow: | The user selects which applicant he/she wants to update. The current point in the admissions process is displayed. The user has the option to move the application to another point. |
| Success exit criteria: | The status of the application is changed. |
| Alternate Flows: | The user may abandon a change without saving it. |
| Failure exit criteria: | The applicant status is not updated. |
| Priority: | Ugrad: mandatory; Grad: mandatory |
| Notes: | The GPO sketched the item below during a client-developer interview. It was intended to convey content, not necessarily a specific layout or color scheme.   Application is currently here<br>Update to this point<br>Update to this point<br>Update to this point<br>Update to this point<br>Update to this point<br>Collecting information<br>Undergoing initial review by GPO<br>Undergoing review by admissions committee<br>Undergoing decision by department<br>Forwarded to Graduate School for final decision<br>Complete |

11

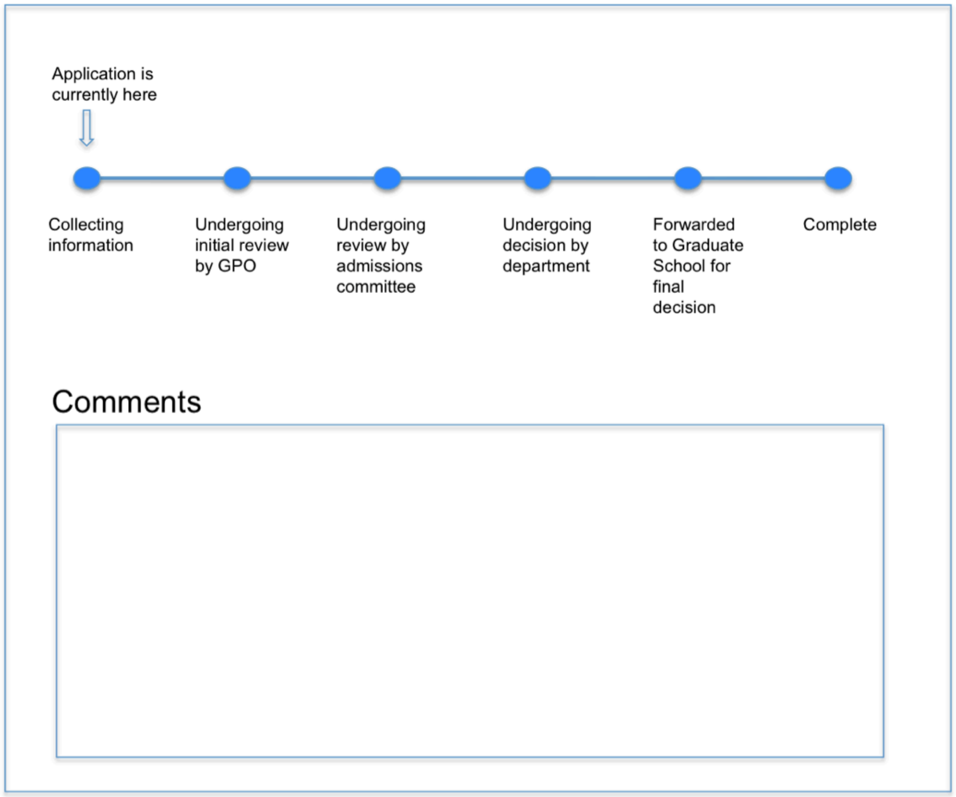
| | |
|---|---|
| Use case name: | Upload GWAAP Dump |
| Primary actor: | Administrator |
| Supporting actors: | GWAAP |
| Use case objective: | To upload application data from the official Graduate School system. |
| Entry criteria: | A GWAAP data file is available; Administrator has been authenticated. |
| Trigger: | Administrator starts "Upload GWAAP Dump" function |
| Basic Flow: | The administrator specifies the name of the GWAAP file. Each applicant's information in the file is uploaded into the web app. Applicant information that has been previously uploaded is overwritten by the data in the dump file. Each applicant that was not previously loaded is sent an e-mail with the URL of the web app, a user id, and a password. |
| Success exit criteria: | 1. The administrator is informed that the GWAAP data has been successfully uploaded.<br>2. New applicants have been informed via e-mail of the web app. |
| Alternate Flows: | Alternate 1: A problem occurs while uploading the GWAAP data. |
| Failure exit criteria: | Alternate 1: Information uploaded up to the point of the problem is considered valid. Applicant Information from the point of the problem to the end of the file is not uploaded. No data from the application with the problem is uploaded. |
| Priority: | Ugrad: optional; Grad: mandatory |
| Notes: | Application information uploaded from the GWAAP file includes the fields listed below. Other GWAAP fields may be safely ignored.<br><br>-FIRST<br>-LAST<br>-MIDDLE<br><br>+STREET1<br>+STREET2<br>+CITY<br>+PROVINCE<br>+STATE<br>+COUNTRY<br>+ZIP<br><br>+COUNTRY_BIRTH<br>+CITIZENSHIP<br><br>EMAIL<br>+TELEPHONE<br>-BIRTHDAY<br>-GENDER<br><br>-ENTER_QTR<br>-ENTER_YEAR<br>-DEGREE<br>-MAJOR<br><br>-GRE_TAKEN<br>-O_GRE_V<br>-O_GRE_Q<br>-O_GRE_A<br>-O_Gre_W<br>-TOEFL_TAKEN<br>-O_TOEFL_SCORE<br>-REF_NUMBER<br>-DATE_APPLY |

12

| | |
|---|---|
| Use case name: | Define users |
| Primary actor: | Administrator |
| Supporting actors: | None |
| Use case objective: | To define the users of the web app. |
| Entry criteria: | Administrator has been authenticated |
| Trigger: | Administrator starts "Define Users" function |
| Basic Flow: | Administrator is shown a list of names, user names, and e-mails of users of the web app organized by category (see note below). Administrator can add, delete, and edit users by category. Administrator can also reset passwords. |
| Success exit criteria: | Users are updated. |
| Alternate Flows: | None |
| Failure exit criteria: | None |
| Priority: | Ugrad: optional; Grad: optional |
| Notes: | Data access permissions:<br>1. Can only view applicant data<br>2. Can specify references<br>3. Can view and update applicant data<br>4. Can submit recommendation<br>5. Can view recommendation<br>6. Can submit comments on applicant<br>7. Can view others' comments on applicant<br>8. Can vote on applicant<br>9. Can make recommendation to Grad School<br>10. Can update admissions status of applicant<br>11. Can manage users |

| | |
|---|---|
| Administrator | 11 |
| GPO | 3, 5, 6, 7, 8, 9, 10 |
| Applicant | 2, 3 |
| Secretary | 3, 10 |
| Reference | 4 |
| Faculty | 1, 6 |
| Grad Committee Member | 1, 5, 6, 7, 8 |

13

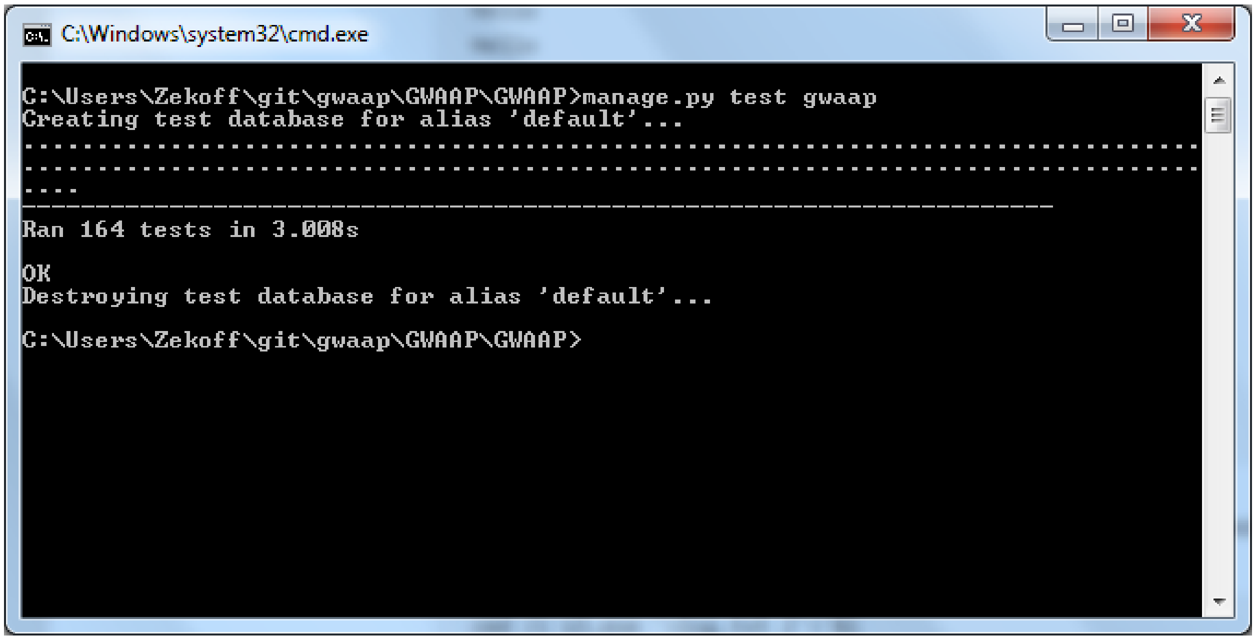
| | |
|---|---|
| Use case name: | Comment |
| Primary actor: | Faculty member |
| Supporting actors: | |
| Use case objective: | To allow non-voting faculty members to comment on applicants |
| Entry criteria: | The faculty member is authenticated |
| Trigger: | The faculty member selects the "comment" function |
| Basic Flow: | The faculty member selects which applicant he/she wants to view and is shown 1) what application information has been received by the department, and 2) where the application is in the admission process (see notes below). The faculty member submits a comment. |
| Success exit criteria: | The faculty member's comment is saved. |
| Alternate Flows: | The faculty member may abandon the comments before submitting them. |
| Failure exit criteria: | No information is saved. |
| Priority: | Ugrad: optional; Grad: optional |
| Notes: | The GPO sketched the item below during a client-developer interview. It was intended to convey content, not necessarily a specific layout or color scheme.   Application is currently here<br>Collecting information<br>Undergoing initial review by GPO<br>Undergoing review by admissions committee<br>Undergoing decision by department<br>Forwarded to Graduate School for final decision<br>Complete<br>Comments |

14

| | |
|---|---|
| Use case name: | Authenticate |
| Primary actor: | User |
| Supporting actors: | |
| Use case objective: | To determine if a user is authorized to use the system and, if so, what functionality is allowed. |
| Entry criteria: | Web app is started |
| Trigger: | User starts web app |
| Basic Flow: | The web app requests the user to identify him/herself (see note below). A previously-registered user does so. The web app provides a selection of operations the user is authorized to perform. |
| Success exit criteria: | User identity is verified |
| Alternate Flows: | Alternate 1: The user is given three opportunities to authenticate him/herself. Failure on the third attempt results in the web app displaying 1) a message indicating that the user has failed to provide legitimate identifying information, and 2) the e-mail address of the system administrator to contact for assistance. The user is given no further log in attempts without restarting the web app. |
| Failure exit criteria: | The user is not verified. |
| Priority: | Ugrad: optional; Grad: optional |
| Notes: | Traditionally, this is a user name and password. The client leaves the specifics of what information is required to authenticate a user to the developer. |

# Appendix B

## Test Cases and Test Report



C:\Windows\system32\cmd.exe

```
C:\Users\Zekoff\git\gwaap\GWAAP\GWAAP>manage.py test gwaap
Creating test database for alias 'default'...
................................................................................
................................................................................
....
----------------------------------------------------------------------
Ran 164 tests in 3.008s

OK
Destroying test database for alias 'default'...

C:\Users\Zekoff\git\gwaap\GWAAP\GWAAP>
```

```
from GWAAP.gwaap.models import ApplicantProfile, Reference, Comment, Vote, \
    STATUS_CODE, RELATIVE_RANK, VOTE_CHOICES, GwaapProfile
from django.contrib.auth.models import Permission
from django.core import mail
from django.core.mail import send_mail
from django.db.models.fields.files import FieldFile
from django.db.utils import IntegrityError
from django.test import TestCase
from django.test.client import Client, RequestFactory
from models import Applicant, Application, User
import django.db.models
from django.core.files.base import File
from django.core.files.uploadedfile import UploadedFile
from django.http import HttpRequest
#from GWAAP.gwaap.views import userActions


class ViewTests(TestCase):

    def getRequest(self, address):
        # This allows tests to skip the url.py file while testing
        # BUT does not allow for sessions
        return RequestFactory().get(address)

#    def test_00010_userActionsViewExists(self):
#        client = Client()
#        response = client.get('/user/')
#        self.assertEqual(response.status_code, 200)

    def test_00020_userActionsIsCorrectPage(self):
        client = Client()
        user = User.objects.create(username='alan')
        user.set_password('password')
        user.save()
        client.login(username='alan', password='password')
        response = client.get('/user/')
        self.assertContains(response, 'User Actions')

#    def test_00021_applicantCannotLoginToUserArea(self):
#        client = Client()
#        applicant = Applicant.objects.create(username='app')
#        applicant.set_password('pass')
#        applicant.save()
#        client.login(username='app', password='pass')
#        response = client.get('/user/')
#        self.assertEqual(response.status_code, 302)

#    def test_00030_djangoResponseFactoryTest(self):
#        response = userActions(self.getRequest('/user/'))
#        self.assertEqual(response.status_code, 200)

    def test_00040_unauthenticatedUserGetsRedirected(self):
        client = Client()
        response = client.get('/user/')
        self.assertEqual(response.status_code, 302)

#    def test_00050_userLoginPresentsForm(self):
```

```
#        client = Client()

    def test_00050_isUserPermissionExists(self):
        permission = Permission.objects.get(codename='is_gwaap_user')
        self.assertIsInstance(permission, Permission)

    def test_00060_usersAutomaticallyGetUserPermission(self):
        user = User.objects.create(username="newuser")
        user.set_password("pass")
        user.save()
        self.assertTrue(user.has_perm('gwaap.is_gwaap_user'))

    def test_00070_applicantsDontHaveUserPermission(self):
        applicant = Applicant.objects.create(username="applicantman")
        applicant.set_password("pass")
        applicant.save()
        self.assertFalse(applicant.has_perm('gwaap.is_gwaap_user'))

    def test_00080_usersCanViewActionsPage(self):
        client = Client()
        user = User.objects.create(username="userman")
        user.set_password("passs")
        user.save()
        client.login(username="userman", password="passs")
        response = client.get('/user/')
        self.assertContains(response, 'User Actions')

    def test_00090_applicantsGetRedirected(self):
        client = Client()
        app = Applicant.objects.create(username="applicant")
        app.set_password("pass")
        app.save()
        client.login(username="applicant", password="pass")
        response = client.get('/user/')
        self.assertEqual(response.status_code, 302)

    def test_00100_unauthenticatedUserGoesToLogin(self):
        client = Client()
        response = client.get('/user/')
        self.assertRedirects(response, '/user/login/?next=/user/')

    def test_00110_userLoginFormIsRealForm(self):
        client = Client()
        response = client.get('/user/login/')
        self.assertContains(response, "<form")

    def test_00120_userLoginFormAcceptsPostDataAndFails(self):
        client = Client()
        data = dict(username='testuser', password='password')
        response = client.post('/user/login/', data, follow=True)
        self.assertContains(response, "Authentication failed")

    def test_00130_userLoginAcceptsGoodLoginData(self):
        client = Client()
        data = dict(username='testuser', password='password')
        user = User.objects.create(username='testuser')
        user.set_password('password')
        user.save()
        response = client.post('/user/login/', data, follow=True)
        self.assertContains(response, 'Login successful')

    def test_00140_applicantFailsUserLogin(self):
        client = Client()
        data = dict(username='applicant', password='pass')
        app = Applicant.objects.create(username='applicant')
        app.set_password('pass')
        app.save()
        response = client.post('/user/login/', data, follow=True)
        self.assertContains(response, 'Authentication failed')

    def test_00150_applicantHomePageExists(self):
        client = Client()
        response = client.get('/')
        self.assertTrue(response.status_code in [200, 302])
#        self.assertContains(response, 'Applicant Home')

    def test_00160_applicantHomePageRequiresLogin(self):
        client = Client()
        app = Applicant.objects.create(username='applicant')
        app.set_password('pass')
        app.save()
        # do NOT login user
        response = client.get('/', follow=True)
        self.assertContains(response, 'Applicant Login')

    def test_00170_applicantLoginPageExists(self):
        client = Client()
        response = client.get('/login/')
        self.assertContains(response, 'Applicant Login')

    def test_00180_applicantHomeRequiresApplicantPermission(self):
        client = Client()
        user = User.objects.create(username='user')
        user.set_password('pass')
        user.save()
        client.login(username='user', password='pass')
        response = client.get('/', follow=True)
        self.assertContains(response, 'Applicant Login')

    def test_00190_applicantsHaveApplicantPermission(self):
        client = Client()
        app = Applicant.objects.create(username='applicant')
        app.set_password('password')
        app.save()
        client.login(username='applicant', password='password')
        response = client.get('/')
        self.assertContains(response, 'Applicant Home')

    def test_00200_applicantPermissionExists(self):
        perm = Permission.objects.get(codename="is_gwaap_applicant")
        self.assertIsInstance(perm, Permission)

    def test_00210_applicantAutomaticallyGetsApplicantPermission(self):
        applicant = Applicant.objects.create(username="applicant")
        self.assertTrue(applicant.has_perm('gwaap.is_gwaap_applicant'))

    def test_00220_applicantLoginHasForm(self):
        client = Client()
        app = Applicant.objects.create(username='applicant')
        app.set_password('password')
        app.save()
        client.login(username='applicant', password='password')
        response = client.get('/login/')
        self.assertContains(response, '<form')

    def test_00230_applicantLoginAcceptsPost(self):
        client = Client()
        app = Applicant.objects.create(username='applicant')
        app.set_password('password')
        app.save()
        data = dict(username='applicant', password='password')
        response = client.post('/login/', data, follow=True)
        self.assertContains(response, 'Login successful')

    def test_00240_applicantLoginRejectsUsers(self):
        client = Client()
        user = User.objects.create(username="baduser")
        user.set_password("pass")
        user.save()
        data = dict(username='baduser', password='pass')
        response = client.post('/login/', data, follow=True)
        self.assertContains(response, 'Authentication failed')

    def test_00250_referenceExists(self):
        applicant = Applicant.objects.create(username='newapplicant')
        ref = Reference.objects.create(attached_to=applicant.get_application())
        self.assertIsInstance(ref, Reference)

    def test_00260_addMultipleReferencesToApplication(self):
        applicant = Applicant.objects.create(username='newapplicant')
        Reference.objects.create(attached_to=applicant.get_application())
        Reference.objects.create(attached_to=applicant.get_application())
self.assertEqual(len(Reference.objects.filter(attached_to=applicant.get_application())), 2)

    def test_00270_referenceHasEmailField(self):
        applicant = Applicant.objects.create(username='applicant')
        ref = Reference.objects.create(attached_to=applicant.get_application())
        ref.email = 'zekoff@gmail.com'
        ref.save()
        self.assertEqual(Reference.objects.get(attached_to=applicant).email,
'zekoff@gmail.com')

    def test_00280_referenceEmailRejectsNonEmail(self):
        applicant = Applicant.objects.create(username='applicant')
        ref = Reference.objects.create(attached_to=applicant.get_application())
        ref.email = 'bademail'
        ref.save()
        self.assertRaises(NameError, ref.save())

    def test_00290_sendsEmail(self):
```

```python
        send_mail("subject", "Here is the message...", 'gwaap@auburn.edu', ['zekoff@gmail.com'])
        self.assertEqual(len(mail.outbox), 1) #@UndefinedVariable

    def test_00300_sendsEmailToAddressFromReference(self):
        applicant = Applicant.objects.create(username="applicant")
        ref = Reference.objects.create(attached_to=applicant.get_application())
        ref.email = 'reference@company.com'
        ref.save()
        send_mail('Reference request', "Message content", 'gwaap@auburn.edu', [ref.email])
        self.assertEqual(mail.outbox[0].recipients()[0], 'reference@company.com') #@UndefinedVariable

    def test_00310_setupReferenceActionExistsForApplicants(self):
        applicant = Applicant.objects.create(username='applicant')
        applicant.set_password('pass')
        applicant.save()
        client = Client()
        client.login(username='applicant', password='pass')
        response = client.get('/')
        self.assertContains(response, 'Add Reference')

    def test_00320_addReferenceViewExists(self):
        applicant = Applicant.objects.create(username='applicant')
        applicant.set_password('pass')
        applicant.save()
        client = Client()
        client.login(username='applicant', password='pass')
        response = client.get('/add_reference/')
        self.assertContains(response, 'Add Reference')

    def test_00330_addingReferenceRequiresLogin(self):
        client = Client()
        response = client.get('/add_reference/', follow=True)
        self.assertContains(response, 'Applicant Login')

    def test_00340_completeReferenceViewExists(self):
        app = Applicant.objects.create(username='app')
        ref = Reference.objects.create(attached_to=app.get_application())
        ref.unique_id = '1'
        ref.save()
        response = Client().get('/reference/1/')
        self.assertTrue(response.status_code in [200, 302])

# No longer using this method of authentication
#    def test_00350_completeReferenceViewRequestsVerification(self):
#        app = Applicant.objects.create(username='app')
#        ref = Reference.objects.create(attached_to=app.get_application())
#        ref.unique_id = '1'
#        ref.save()
#        response = Client().get('/reference/1/')
#        self.assertContains(response, 'Verification code')

    def test_00360_completeReferenceAcceptsPost(self):
        app = Applicant.objects.create(username='app')
        ref = Reference.objects.create(attached_to=app.get_application())
        ref.unique_id = '1'
        ref.save()
        data = dict(comments='bbb', overall="0", reference_name="Jim",
reference_affiliation="Bob")
        data['integrity'] = 0
        data['development'] = 0
        data['communication'] = 0
        data['motivation'] = 0
        data['research'] = 0
        response = Client().post('/reference/1/', data)
        self.assertContains(response, 'POST accepted')

    def test_00361_completeReferenceAcceptsPost(self):
        app = Applicant.objects.create(username='app')
        ref = Reference.objects.create(attached_to=app.get_application())
        ref.unique_id = '1'
        ref.save()
        data = dict(comments='bbb', overall="0", reference_name="Jim",
reference_affiliation="Bob")
        response = Client().post('/reference/1/', data)
        self.assertContains(response, 'Error')

    def test_00370_completeReferenceGETUsesUniqueID(self):
        applicant = Applicant.objects.create(username='app')
        applicant.first_name = "Test"
        applicant.last_name = "Applicant"
        applicant.save()
        ref = Reference.objects.create(attached_to=applicant.get_application())
        ref.email = 'test@email.com'
        ref.save()
        response = Client().get('/reference/' + str(ref.unique_id) + '/')
        self.assertContains(response, 'Test Applicant')

    def test_00380_completeReferenceGETUsesUniqueID2(self):
        applicant = Applicant.objects.create(username='app')
        applicant.first_name = "Michael"
        applicant.last_name = "Zekoff"
        applicant.save()
        ref = Reference.objects.create(attached_to=applicant.get_application())
        ref.email = 'test@email.com'
        ref.save()
        response = Client().get('/reference/' + str(ref.unique_id) + '/')
        self.assertContains(response, 'Michael Zekoff')

    def test_00390_addReferenceHasForm(self):
        applicant = Applicant.objects.create(username='applicant')
        applicant.set_password('pass')
        applicant.save()
        client = Client()
        client.login(username='applicant', password='pass')
        response = client.get('/add_reference/')
        self.assertContains(response, '<form')

    def test_00400_addReferencePostGeneratesReference(self):
        applicant = Applicant.objects.create(username='applicant')
        applicant.set_password('pass')
        applicant.save()
        client = Client()
        client.login(username='applicant', password='pass')
        data = dict(email='reference@school.edu')
        client.post('/add_reference/', data)
        self.assertEqual('reference@school.edu',
Reference.objects.get(attached_to=applicant.get_application()).email)

    def test_00410_displayApplicantsViewExists(self):
        user = User.objects.create(username='user')
        user.set_password('pass')
        user.save()
        client = Client()
        client.login(username='user', password='pass')
        response = client.get('/user/display_applicants/')
        self.assertContains(response, 'Display Applicants')

    def test_00420_applicantCannotViewApplicants(self):
        app = Applicant.objects.create(username='applicant')
        app.set_password('pass')
        app.save()
        client = Client()
        client.login(username='applicant', password='pass')
        response = client.get('/user/display_applicants/', follow=True)
        self.assertContains(response, 'User Login')

    def test_00430_mustBeLoggedInToViewApplicants(self):
        client = Client()
        response = client.get('/user/display_applicants/', follow=True)
        self.assertContains(response, 'User Login')

    def test_00440_displayApplicantsShowsAllApplicants(self):
        for x in range(5):
            applicantName = 'applicant' + str(x)
            Applicant.objects.create(username=applicantName)
        user = User.objects.create(username='user')
        user.set_password('pass')
        user.save()
        client = Client()
        client.login(username='user', password='pass')
        response = client.get('/user/display_applicants/')
        self.assertContains(response, '<tr>', 6)

    def test_00450_userCanViewSingleApplicantInfo(self):
        for x in range(5):
            applicantName = 'applicant' + str(x + 1)
            Applicant.objects.create(username=applicantName)
        user = User.objects.create(username='user')
        user.set_password('pass')
        user.save()
        client = Client()
        client.login(username='user', password='pass')
        response = client.get('/user/view_applicant/1/')
        self.assertContains(response, 'applicant1')

    def test_00460_applicantCannotViewApplicantInfo(self):
        applicant = Applicant.objects.create(username='applicant')
        applicant.set_password('pass')
        applicant.save()
        client = Client()
        client.login(username='applicant', password='pass')
        response = client.get('/user/view_applicant/1/', follow=True)
        self.assertContains(response, 'User Login')

    def test_00470_displayApplicantsLinksToApplicantViews(self):
        for x in range(5):
            applicantName = 'applicant' + str(x + 1)
            Applicant.objects.create(username=applicantName)
        user = User.objects.create(username='user')
        user.set_password('pass')
        user.save()
        client = Client()
        client.login(username='user', password='pass')
        response = client.get('/user/display_applicants/')
        self.assertContains(response, '/user/view_applicant/1/')

    def test_00480_applicantViewPageGivesCommentOption(self):
        Applicant.objects.create(username='applicant')
        user = User.objects.create(username='user')
        user.set_password('pass')
        user.save()
        client = Client()
        client.login(username='user', password='pass')
        response = client.get('/user/view_applicant/1/', follow=True)
        self.assertContains(response, 'Comment')

    def test_00490_applicantViewPageGivesVoteOption(self):
        Applicant.objects.create(username='applicant')
        user = User.objects.create(username='user')
        user.set_password('pass')
        user.save()
        client = Client()
        client.login(username='user', password='pass')
        response = client.get('/user/view_applicant/1/', follow=True)
        self.assertContains(response, 'Vote')

    def test_00500_makeCommentViewExists(self):
        Applicant.objects.create(username='applicant')
        user = User.objects.create(username='user')
        user.set_password('pass')
        permission = Permission.objects.get(codename="can_comment")
        user.user_permissions.add(permission)
        user.save()
        client = Client()
        client.login(username='user', password='pass')
        response = client.get('/user/make_comment/1/')
        self.assertContains(response, 'Make Comment')

    def test_00510_makeCommentViewRequiresUserLogin(self):
        applicant = Applicant.objects.create(username='applicant')
        applicant.set_password('pass')
        applicant.save()
        user = User.objects.create(username='user')
        user.set_password('pass')
        permission = Permission.objects.get(codename="is_gwaap_user")
        user.user_permissions.add(permission)
        user.save()
        client = Client()
        client.login(username='applicant', password='pass')
        response = client.get('/user/make_comment/1/', follow=True)
        self.assertContains(response, 'User Login')

    def test_00520_makeCommentPostSavesComment(self):
        applicant = Applicant.objects.create(username='applicant')
        user = User.objects.create(username='user')
        user.set_password('pass')
        permission = Permission.objects.get(codename="can_comment")
        user.user_permissions.add(permission)
        user.save()
        client = Client()
        client.login(username='user', password='pass')
        data = dict(comment='Good applicant')
        client.post('/user/make_comment/1/', data)
        comment = Comment.objects.get(attached_to=applicant.get_application())
        self.assertEqual(comment.content, "Good applicant")

    def test_00530_makeCommentGetIncludesForm(self):
        Applicant.objects.create(username='applicant')
        user = User.objects.create(username='user')
        user.set_password('pass')
        permission = Permission.objects.get(codename="can_comment")
        user.user_permissions.add(permission)
        user.save()
        client = Client()
        client.login(username='user', password='pass')
        response = client.get('/user/make_comment/1/')
        self.assertContains(response, '<form')

    def test_00540_postCommentSavesUser(self):
        applicant = Applicant.objects.create(username='applicant')
        user = User.objects.create(username='user')
        user.set_password('pass')
        permission = Permission.objects.get(codename="can_comment")
        user.user_permissions.add(permission)
        user.save()
        client = Client()
        client.login(username='user', password='pass')
        data = dict(comment='Good applicant')
        client.post('/user/make_comment/1/', data)
        comment = Comment.objects.get(attached_to=applicant.get_application())
        self.assertEqual(comment.made_by, user)

    def test_00550_castVoteViewExists(self):
        Applicant.objects.create(username='applicant')
        user = User.objects.create(username='user')
        user.set_password('pass')
        permission = Permission.objects.get(codename="can_vote")
        user.user_permissions.add(permission)
        user.save()
        client = Client()
        client.login(username='user', password='pass')
        response = client.get('/user/cast_vote/1/')
        self.assertContains(response, 'Cast Vote')

    def test_00560_castVoteViewRequiresPermission(self):
        applicant = Applicant.objects.create(username='applicant')
        applicant.set_password('pass')
        applicant.save()
        user = User.objects.create(username='user')
        user.set_password('pass')
        permission = Permission.objects.get(codename="can_vote")
        user.user_permissions.add(permission)
        user.save()
        client = Client()
        client.login(username='applicant', password='pass')
        response = client.get('/user/cast_vote/1/', follow=True)
        self.assertContains(response, 'User Login')

    def test_00570_castVotePostSavesVote(self):
        applicant = Applicant.objects.create(username='applicant')
        user = User.objects.create(username='user')
        user.set_password('pass')
        permission = Permission.objects.get(codename="can_vote")
        user.user_permissions.add(permission)
        user.save()
        client = Client()
        client.login(username='user', password='pass')
        data = dict(vote=1)
        client.post('/user/cast_vote/1/', data)
        vote = Vote.objects.get(attached_to=applicant.get_application())
        self.assertEqual(str(vote), 'Weak Reject by user for applicant')

    def test_00580_castVoteGetIncludesForm(self):
        Applicant.objects.create(username='applicant')
        user = User.objects.create(username='user')
        user.set_password('pass')
        permission = Permission.objects.get(codename="can_vote")
        user.user_permissions.add(permission)
        user.save()
        client = Client()
        client.login(username='user', password='pass')
        response = client.get('/user/cast_vote/1/')
        self.assertContains(response, '<form')

    def test_00590_castVoteSavesUser(self):
        applicant = Applicant.objects.create(username='applicant')
        user = User.objects.create(username='user')
        user.set_password('pass')
        permission = Permission.objects.get(codename="can_vote")
        user.user_permissions.add(permission)
        user.save()
        client = Client()
        client.login(username='user', password='pass')
        data = dict(vote=1)
        client.post('/user/cast_vote/1/', data)
        vote = Vote.objects.get(attached_to=applicant.get_application())
        self.assertEqual(vote.made_by, user)

    def test_00600_logoutView(self):
        user = User.objects.create(username='user')
        user.set_password('pass')
        user.save()
        client = Client()
        client.login(username='user', password='pass')
        response = client.get('/logout/')
        self.assertContains(response, "logged out")

    def test_00610_searchApplicantsViewExists(self):
        user = User.objects.create(username="user")
        user.set_password("pass")
        user.save()
        client = Client()
        client.login(username="user", password='pass')
        response = client.post('/user/search_applicants/', {}, follow=True)
        self.assertContains(response, "Search Applicants")

    def test_00620_searchApplicantsRequiresUserLogin(self):
        client = Client()
        response = client.post('/user/search_applicants/', {}, follow=True)
        self.assertContains(response, "User Login")

    def test_00630_searchApplicantsAcceptsPostData(self):
        user = User.objects.create(username="user")
        user.set_password("pass")
        user.save()
        client = Client()
        client.login(username="user", password='pass')
        data = dict()
        response = client.post('/user/search_applicants/', data)
        self.assertContains(response, 'POST accepted')

    def test_00640_searchPageGetsSearchString(self):
        user = User.objects.create(username="user")
        user.set_password("pass")
        user.save()
        client = Client()
        client.login(username="user", password='pass')
        data = dict(search_string="testing")
        response = client.post('/user/search_applicants/', data)
        self.assertContains(response, 'Search String: testing')

    def test_00650_resultsIncludeSearchByUsername(self):
        for x in range(5):
            username = "app" + str(x)
            first = "Test" + str(x)
            last = "Applicant" + str(x)
            applicant = Applicant.objects.create(username=username)
            applicant.first_name = first
            applicant.last_name = last
            applicant.save()
        user = User.objects.create(username="user")
        user.set_password("pass")
        user.save()
        client = Client()
        client.login(username="user", password='pass')
        data = {'search_string':'app'}
        response = client.post('/user/search_applicants/', data)
        self.assertContains(response, 'app1')

    def test_00660_resultsIncludeSearchByFirstName(self):
        for x in range(5):
            username = "app" + str(x)
            first = "Test" + str(x)
            last = "Applicant" + str(x)
            applicant = Applicant.objects.create(username=username)
            applicant.first_name = first
            applicant.last_name = last
            applicant.save()
        user = User.objects.create(username="user")
        user.set_password("pass")
        user.save()
        client = Client()
        client.login(username="user", password='pass')
        data = {'search_string':'Test1'}
        response = client.post('/user/search_applicants/', data)
        self.assertContains(response, 'app1')

    def test_00670_resultsIncludeSearchByLastName(self):
        for x in range(5):
            username = "app" + str(x)
            first = "Test" + str(x)
            last = "Applicant" + str(x)
            applicant = Applicant.objects.create(username=username)
            applicant.first_name = first
            applicant.last_name = last
            applicant.save()
        user = User.objects.create(username="user")
        user.set_password("pass")
        user.save()
        client = Client()
        client.login(username="user", password='pass')
        data = {'search_string':'Applicant3'}
        response = client.post('/user/search_applicants/', data)
        self.assertContains(response, 'app3')

    def test_00680_resultsDontIncludeExtras(self):
        for x in range(5):
            username = "app" + str(x)
            first = "Test" + str(x)
            last = "Applicant" + str(x)
            applicant = Applicant.objects.create(username=username)
            applicant.first_name = first
            applicant.last_name = last
            applicant.save()
        user = User.objects.create(username="user")
        user.set_password("pass")
        user.save()
        client = Client()
        client.login(username="user", password='pass')
        data = {'search_string':'Applicant3'}
        response = client.post('/user/search_applicants/', data)
        self.assertContains(response, 'Number of results: 1')

    def test_00690_applicantViewApplication(self):
        a = Applicant.objects.create(username='applicant')
        a.set_password('pass')
        a.save()
        client = Client()
        client.login(username='applicant', password='pass')
        response = client.get('/view_application/')
        self.assertContains(response, "Application Details")

    def test_00700_viewingApplicationRequiresLogin(self):
        a = Applicant.objects.create(username='applicant')
        a.set_password('pass')
        a.save()
        client = Client()
        response = client.get('/view_application/', follow=True)
        self.assertContains(response, "Applicant Login")

    def test_00710_userCannotLoginToApplicantApplicationView(self):
        user = User.objects.create(username='user')
        user.set_password('pass')
        user.save()
        client = Client()
        client.login(username='user', password='pass')
        response = client.get('/view_application/', follow=True)
        self.assertContains(response, 'Applicant Login')

    def test_00720_applicantProfileInfoViewExists(self):
        app = Applicant.objects.create(username="app")
        app.set_password('pass')
        app.save()
        client = Client()
        client.login(username="app", password='pass')
        response = client.get('/view_profile/')
        self.assertContains(response, "Applicant Profile")

    def test_00730_applicantProfileRequiresLogin(self):
        app = Applicant.objects.create(username="app")
        app.set_password('pass')
        app.save()
        client = Client()
        response = client.get('/view_profile/', follow=True)
        self.assertContains(response, "Applicant Login")

    def test_00740_submitResumeViewExists(self):
        app = Applicant.objects.create(username="app")
        app.set_password('pass')
        app.save()
        client = Client()
        client.login(username='app', password='pass')
        response = client.get('/upload_resume/', follow=True)
        self.assertContains(response, 'Upload Resume')

    def test_00750_submitResumeRequiresApplicantLogin(self):
        app = User.objects.create(username="app")
        app.set_password('pass')
        app.save()
        client = Client()
        client.login(username='app', password='pass')
        response = client.get('/upload_resume/', follow=True)
        self.assertContains(response, 'Applicant Login')

    def test_00760_submitResumeContainsForm(self):
        app = Applicant.objects.create(username="app")
        app.set_password('pass')
        app.save()
        client = Client()
        client.login(username='app', password='pass')
        response = client.get('/upload_resume/', follow=True)
        self.assertContains(response, '<form')

    def test_00770_submitResumeFailsIfNoFile(self):
        app = Applicant.objects.create(username="app")
        app.set_password('pass')
        app.save()
        client = Client()
        client.login(username='app', password='pass')
        data = dict()
        response = client.post('/upload_resume/', data, follow=True)
        self.assertContains(response, "Error submitting file.")

# This test case works, but is deactivated to prevent the creation of actual files on the
filesystem during testng.
#    def test_00780_submitResumeWithFile(self):
#        app = Applicant.objects.create(username="app")
#        app.set_password('pass')
#        app.save()
#        client = Client()
#        client.login(username='app', password='pass')
#        resume_file = open("resumefile", "w")
#        resume_file.write("testing")
#        resume_file.close()
#        file_to_upload = UploadedFile(open('resumefile'))
#        data = dict(resume=file_to_upload)
#        response = client.post('/upload_resume/', data, follow=True)
#        self.assertEqual(u"applicant_files/app/resumefile",
app.get_application().resume.name)

# See test case 00780
#    def test_00790_fileSubmissionVerifiesContentTypeFailure(self):
#        app = Applicant.objects.create(username="app")
#        app.set_password('pass')
#        app.save()
#        client = Client()
#        client.login(username='app', password='pass')
#        resume_file = open("resumefile", "w")
#        resume_file.write("testing")
#        resume_file.close()
#        file_to_upload = UploadedFile(open('resumefile'))
#        file_to_upload.content_type = "badtype"
#        data = dict(resume=file_to_upload)
#        response = client.post('/upload_resume/', data, follow=True)
#        self.assertEqual("", app.get_application().resume.name)

# The functionality is there, but the test case cannot be coerced like this.
#    def test_00800_fileSubmissionVerifiesContentTypePass(self):
```

```
#        app = Applicant.objects.create(username="app")
#        app.set_password('pass')
#        app.save()
#        client = Client()
#        client.login(username='app', password='pass')
#        resume_file = open("resumefile", "w")
#        resume_file.write("testing")
#        resume_file.close()
#        file_to_upload = UploadedFile(open('resumefile'))
#        file_to_upload.content_type = "application/pdf"
#        data = dict(resume=file_to_upload)
#        response = client.post('/upload_resume/', data, follow=True)
#        print response
#        self.assertContains(response, "Resume uploaded.")

    def test_00810_submitLetterOfIntentViewExists(self):
        app = Applicant.objects.create(username='app')
        app.set_password('pass')
        app.save()
        client = Client()
        client.login(username='app', password="pass")
        response = client.get('/upload_letter/', follow=True)
        self.assertContains(response, "Upload Letter of Intent")

    def test_00820_submitLetterViewRequiresLogin(self):
        app = User.objects.create(username='app')
        app.set_password('pass')
        app.save()
        client = Client()
        client.login(username='app', password="pass")
        response = client.get('/upload_letter/', follow=True)
        self.assertContains(response, "Applicant Login")

    def test_00830_submitLetterContainsForm(self):
        app = Applicant.objects.create(username='app')
        app.set_password('pass')
        app.save()
        client = Client()
        client.login(username='app', password="pass")
        response = client.get('/upload_letter/', follow=True)
        self.assertContains(response, "<form")

    def test_00840_submitLetterFailsIfNoFile(self):
        app = Applicant.objects.create(username="app")
        app.set_password('pass')
        app.save()
        client = Client()
        client.login(username='app', password='pass')
        data = dict()
        response = client.post('/upload_resume/', data, follow=True)
        self.assertContains(response, "Error submitting file.")

    def test_00850_testReferencesPending(self):
        app = Applicant.objects.create(username='app')
        app.set_password('pass')
        app.save()
        for x in range(3):
            Reference.objects.create(attached_to=app.get_application())
        client = Client()
        client.login(username='app', password='pass')
        response = client.get('/view_application/')
        self.assertContains(response, "References Pending")

    def test_00860_testReferencesIncomplete(self):
        app = Applicant.objects.create(username='app')
        app.set_password('pass')
        app.save()
        for x in range(2):
            Reference.objects.create(attached_to=app.get_application())
        client = Client()
        client.login(username='app', password='pass')
        response = client.get('/view_application/')
        self.assertContains(response, "References Incomplete")

    def test_00870_testReferencesComplete(self):
        app = Applicant.objects.create(username='app')
        app.set_password('pass')
        app.save()
        for x in range(3):
            ref = Reference.objects.create(attached_to=app.get_application())
            ref.complete = True
            ref.save()
        client = Client()
        client.login(username='app', password='pass')
        response = client.get('/view_application/')
        self.assertContains(response, "References Complete")

    def test_00880_transcriptStatusComplete(self):
        app = Applicant.objects.create(username='app')
        app.set_password('pass')
        app.save()
        application = app.get_application()
        application.transcript_status = 0
        application.save()
        client = Client()
        client.login(username='app', password='pass')
        response = client.get('/view_application/')
        self.assertContains(response, "Transcript Complete")

    def test_00890_greStatusComplete(self):
        app = Applicant.objects.create(username='app')
        app.set_password('pass')
        app.save()
        application = app.get_application()
        application.gre_status = 0
        application.save()
        client = Client()
        client.login(username='app', password='pass')
        response = client.get('/view_application/')
        self.assertContains(response, "GRE Complete")

    def test_00900_toeflStatusComplete(self):
        app = Applicant.objects.create(username='app')
        app.set_password('pass')
```

```python
        app.save()
        application = app.get_application()
        application.toefl_status = 0
        application.save()
        client = Client()
        client.login(username='app', password='pass')
        response = client.get('/view_application/')
        self.assertContains(response, "TOEFL Complete")

    def test_00910_referenceViewRejectsIfReferenceAlreadyComplete(self):
        app = Applicant.objects.create(username="app")
        ref = Reference.objects.create(attached_to=app.get_application())
        ref.complete = True
        ref.save()
        client = Client()
        response = client.get('/reference/' + str(ref.unique_id) + "/", follow=True)
        self.assertContains(response, "Thank you")


class ModelTests(TestCase):

    def getFreshApplicant(self, thisUsername="username"):
        applicant = Applicant(username=thisUsername)
#        applicant.save()
#        ap = ApplicantProfile(user=applicant)
#        ap.save()
#        app = Application(applicant_profile=ap)
#        app.save()
        applicant.save()
        return applicant

    def test_00010_userExists(self):
        user = User()
        self.assertIsInstance(user, User)

    def test_00020_userHasUsername(self):
        user = User()
        self.assertEqual(user.username, '')

    def test_00021_setUsername(self):
        user = User()
        user.username = "jimbob"
        self.assertEqual(user.username, "jimbob")

    def test_00030_userNameWorks(self):
        user = User()
        user.first_name = "Joe"
        user.last_name = "Tester"
        self.assertEqual(user.get_full_name(), "Joe Tester")

    def test_00040_userPassword(self):
        user = User()
        user.set_password("password42")
        self.assertTrue(user.check_password("password42"))

    def test_00050_applicantExists(self):
        applicant = Applicant()
        self.assertIsInstance(applicant, Applicant)

    def test_00060_applicationExists(self):
        app = Application()
        self.assertIsInstance(app, Application)

#    def test_0070_applicationHasNameField(self):
#        app = Application()
#        app.first_name = "Joe"
#        app.save()
#        self.assertEqual(app.first_name, "Joe")

#    def test_0071_applicationHasApplicantForeignKey(self):
#        app = Application()
#        applicant = Applicant()
#        applicant.save()
#        app.owner = applicant
#        app.save()
#        app = Application.objects.get(pk=1)
#        self.assertEqual(applicant, app.owner)

    def test_00080_applicantProfileExists(self):
        applicantProfile = ApplicantProfile()
        self.assertIsInstance(applicantProfile, ApplicantProfile)

    def test_00090_applicantProfileIsModel(self):
        ap = ApplicantProfile()
        self.assertIsInstance(ap, django.db.models.Model)

    def test_00091_applicantProfileHasId(self):
        app = Applicant(username="applicant")
        app.save()
        apID = app.get_profile().id
        self.assertEqual(apID, 1)

    def test_00100_applicantHasProfile(self):
        applicant = Applicant.objects.create(username="user")
        applicant.save()
        self.assertIsInstance(applicant.get_profile(), ApplicantProfile)

    def test_00101_gettingIdOfApplicant(self):
        app = Applicant()
        app.save()
        appID = app.id
        self.assertEqual(appID, 1)

#    def test_0102_gettingIdOfSecondApplicant(self):
#        app = Applicant.objects.create()
#        app.save()
#        app2 = Applicant.objects.create()
#        app2.save()
#        appID = app2.id
#        self.assertEqual(appID, 2)


#This test really isn't asking anything I want to ask
#    def test_0110_applicantProfileHasApplication(self):
```

```
#        ap = ApplicantProfile()
#        self.assertIsInstance(ap.get_profile(), Application)

#    def test_0111_applicantHasProfileTwoCopies(self):
#        applicant3 = Applicant.objects.create()
#        applicant3.save()
#        applicant2 = Applicant.objects.create()
#        applicant2.save()
#        self.assertIsInstance(applicant2.get_profile(), ApplicantProfile)

    def test_00120_makeTwoApplicantsWithProfiles(self):
        applicant = Applicant(username="mr. applicant")
        applicant.save()
        applicant2 = Applicant(username="mrs. applicant")
        applicant2.save()
        self.assertEqual(applicant2.id, 2)

    def test_00130_makeSingleApplicant(self):
        applicant = Applicant(username="applicant")
        applicant.save()
        self.assertIsInstance(applicant, Applicant)

    def test_00140_getApplicantById(self):
        applicant = Applicant(username="applicant")
        applicant.save()
        appKey = Applicant.objects.get(username="applicant").pk
        self.assertEqual(appKey, 1)

    def test_00150_getApplicantByIdMultipleApplicants(self):
        applicant = Applicant(username="applicant")
        applicant.save()
        applicant = Applicant(username="applicant2")
        applicant.save()
        applicant = Applicant(username="applicant3")
        applicant.save()
        appKey = Applicant.objects.get(username="applicant2").pk
        self.assertEqual(2, appKey)

    def test_00160_duplicateUsernamesRaisesError(self):
        applicant = Applicant(username="applicant")
        applicant.save()
        applicant = Applicant(username="applicant")
        self.assertRaises(IntegrityError, applicant.save)

    def test_00170_deleteProfileAndCreateNewOne(self):
        app = Applicant(username="name")
        app.save()
        ApplicantProfile.objects.get(user=app).delete()
        ap = ApplicantProfile()
        ap.user = app
        ap.save()

    def test_00180_applicantProfileHasApplication(self):
        applicant = self.getFreshApplicant()
        application = Application.objects.get(applicant_profile=applicant.get_profile())
        self.assertIsInstance(application, Application)

```

```
#    def test_00190_trySignalProcessing(self):
#        applicant = Applicant(username="name")
#        applicant.save()
#        
self.assertEqual(Application.objects.get(applicant_profile=applicant.get_profile()).intTest, 
1)

    def test_00200_getApplicationConvenienceMethod(self):
        applicant = Applicant(username="user")
        applicant.save()
        self.assertIsInstance(applicant.get_application(), Application)

    def test_00210_referenceHasUniqueURL(self):
        applicant = Applicant.objects.create(username="app")
        ref = Reference.objects.create(attached_to=applicant.get_application())
        correct_id = ref.unique_id
        self.assertEqual(ref, Reference.objects.get(unique_id=correct_id))

    def test_00220_commentModelExists(self):
        comment = 
Comment.objects.create(attached_to=Applicant.objects.create(username='test').get_application(
))
        self.assertIsInstance(comment, Comment)

    def test_00230_commentForeignKeyIsApplication(self):
        app = Applicant.objects.create(username='applicant')
        comment = Comment.objects.create(attached_to=app.get_application())
        self.assertIsInstance(comment.attached_to, Application)

    def test_00240_commentMadeByUser(self):
        app = Applicant.objects.create(username='applicant')
        user = User.objects.create(username='user')
        comment = Comment.objects.create(attached_to=app.get_application())
        comment.made_by = user
        comment.save()
        comment = Comment.objects.get(attached_to=app.get_application())
        self.assertEqual(comment.made_by, user)

    def test_00250_commentContainsContent(self):
        app = Applicant.objects.create(username='applicant')
        comment = Comment.objects.create(attached_to=app.get_application())
        comment.content = "This is a good applicant"
        comment.save()
        comment = Comment.objects.get(attached_to=app.get_application())
        self.assertEqual(comment.content, "This is a good applicant")

    def test_00260_voteModelExists(self):
        vote = 
Vote.objects.create(attached_to=Applicant.objects.create(username='app').get_application())
        self.assertIsInstance(vote, Vote)

    def test_00270_voteMadeByUser(self):
        app = Applicant.objects.create(username='app')
        user = User.objects.create(username='user')
        vote = Vote.objects.create(attached_to=app.get_application())
        vote.made_by = user
        vote.save()
        vote = Vote.objects.get(attached_to=app.get_application())
        self.assertIsInstance(vote.made_by, User)

    def test_00280_voteModelContainsVote(self):
        app = Applicant.objects.create(username='app')
        vote = Vote.objects.create(attached_to=app.get_application())
        vote.content = 1
        vote.save()
        vote = Vote.objects.get(attached_to=app.get_application())
        self.assertEqual(vote.content, 1)

    def test_00280_voteModelUnicodeGivesCorrectString(self):
        app = Applicant.objects.create(username='app')
        vote = Vote.objects.create(attached_to=app.get_application())
        vote.content = 2
        vote.save()
        vote = Vote.objects.get(attached_to=app.get_application())
        self.assertEqual(vote.__unicode__(), VOTE_CHOICES[2][1] + " by None for app")

    def test_00290_commentPermissionExists(self):
        perm = Permission.objects.get(codename='can_comment')
        self.assertIsInstance(perm, Permission)

    def test_00300_votePermissionExists(self):
        perm = Permission.objects.get(codename='can_vote')
        self.assertIsInstance(perm, Permission)

    def test_00310_referenceCanMakeFreeformComments(self):
        applicant = Applicant.objects.create(username='applicant')
        reference = Reference.objects.create(attached_to=applicant.get_application())
        reference.comments = "good applicant"
        reference.save()
        refer = Reference.objects.get(attached_to=applicant.get_application())
        self.assertEqual(refer.comments, "good applicant")

    def test_00320_manyToManyUserAndCommentRelationships(self):
        user1 = User.objects.create(username='user1')
        user2 = User.objects.create(username="user2")
        applicant1 = Applicant.objects.create(username='applicant1')
        applicant2 = Applicant.objects.create(username='applicant2')
        comment11 = Comment.objects.create(attached_to=applicant1.get_application())
        comment11.made_by = user1
        comment11.save()
        comment12 = Comment.objects.create(attached_to=applicant2.get_application())
        comment12.made_by = user1
        comment12.save()
        comment21 = Comment.objects.create(attached_to=applicant1.get_application())
        comment21.made_by = user2
        comment21.save()
        comment22 = Comment.objects.create(attached_to=applicant2.get_application())
        comment22.made_by = user2
        comment22.save()

    def test_00330_manyToManyUserAndCommentRelationships(self):
        user1 = User.objects.create(username='user1')
        user2 = User.objects.create(username="user2")
        applicant1 = Applicant.objects.create(username='applicant1')
        applicant2 = Applicant.objects.create(username='applicant2')
        comment11 = Vote.objects.create(attached_to=applicant1.get_application())
        comment11.made_by = user1
        comment11.save()
        comment12 = Vote.objects.create(attached_to=applicant2.get_application())
        comment12.made_by = user1
        comment12.save()
        comment21 = Vote.objects.create(attached_to=applicant1.get_application())
        comment21.made_by = user2
        comment21.save()
        comment22 = Vote.objects.create(attached_to=applicant2.get_application())
        comment22.made_by = user2
        comment22.save()

    def test_00340_referenceModelContainsName(self):
        app = Applicant.objects.create(username="app")
        ref = Reference.objects.create(attached_to=app.get_application())
        ref.name = 'Mr. Reference'
        ref.save()
        ref = Reference.objects.get(attached_to=app.get_application())
        self.assertEqual("Mr. Reference", ref.name)

    def test_00350_referenceModelContainsAffiliation(self):
        app = Applicant.objects.create(username="app")
        ref = Reference.objects.create(attached_to=app.get_application())
        ref.affiliation = 'Big Company'
        ref.save()
        ref = Reference.objects.get(attached_to=app.get_application())
        self.assertEqual("Big Company", ref.affiliation)

    def test_00360_applicationModelHasResumeFileField(self):
        app = Applicant.objects.create(username='app')
        application = app.get_application()
        self.assertIsInstance(application.resume, FieldFile)

    def test_00370_applicationModelHasLetterOfIntentFileField(self):
        app = Applicant.objects.create(username="app")
        application = app.get_application()
        self.assertIsInstance(application.letter_of_intent, FieldFile)

# Passes, but disabled to prevent writing trash files to filesystem
#    def test_00380_applicationSavesFile(self):
#        app = Applicant.objects.create(username="app")
#        application = app.get_application()
#        resume_file = File(file("resumefile", "w"))
#        application.resume = resume_file
#        application.save()
#        app.save()
#        self.assertIsInstance(app.get_application().resume, FieldFile)

    def test_00390_modelsModuleHasStatusTuple(self):
        status = 0
        self.assertTrue("Complete" in STATUS_CODE[status])

    def test_00400_applicationModelHasGreStatusField(self):
        app = Applicant.objects.create(username='app')
        application = app.get_application()
        application.gre_status = 0
        application.save()
        application = Applicant.objects.get(username='app').get_application()
        self.assertTrue("Complete" in STATUS_CODE[application.gre_status])

    def test_00410_applicationModelHasToeflStatusField(self):
        app = Applicant.objects.create(username='app')
        application = app.get_application()
        application.toefl_status = 1
        application.save()
        application = Applicant.objects.get(username='app').get_application()
        self.assertTrue("Incomplete" in STATUS_CODE[application.toefl_status])

    def test_00420_applicationModelHasTranscriptStatusField(self):
        app = Applicant.objects.create(username='app')
        application = app.get_application()
        application.transcript_status = 0
        application.save()
        application = Applicant.objects.get(username="app").get_application()
        self.assertTrue("Complete" in STATUS_CODE[application.transcript_status])

    def test_00430_referenceHasIntegrityField(self):
        app = Applicant.objects.create(username='app')
        ref = Reference.objects.create(attached_to=app.get_application())
        ref.integrity = 0
        ref.save()
        ref = Reference.objects.get(attached_to=app)
        self.assertTrue(ref.integrity in RELATIVE_RANK[ref.integrity])

    def test_00440_referenceHasDevelopmentField(self):
        app = Applicant.objects.create(username='app')
        ref = Reference.objects.create(attached_to=app.get_application())
        ref.development = 0
        ref.save()
        ref = Reference.objects.get(attached_to=app)
        self.assertTrue(ref.development in RELATIVE_RANK[ref.development])

    def test_00450_referenceHasCommunicationField(self):
        app = Applicant.objects.create(username='app')
        ref = Reference.objects.create(attached_to=app.get_application())
        ref.communication = 0
        ref.save()
        ref = Reference.objects.get(attached_to=app)
        self.assertTrue(ref.communication in RELATIVE_RANK[ref.communication])

    def test_00460_referenceHasMotivationField(self):
        app = Applicant.objects.create(username='app')
        ref = Reference.objects.create(attached_to=app.get_application())
        ref.motivation = 0
        ref.save()
        ref = Reference.objects.get(attached_to=app)
        self.assertTrue(ref.motivation in RELATIVE_RANK[ref.motivation])

    def test_00470_referenceHasResearchField(self):
        app = Applicant.objects.create(username='app')
        ref = Reference.objects.create(attached_to=app.get_application())
        ref.research = 0
```

```python
        ref.save()
        ref = Reference.objects.get(attached_to=app)
        self.assertTrue(ref.research in RELATIVE_RANK[ref.research])

    def test_00480_referenceHasOverallField(self):
        app = Applicant.objects.create(username='app')
        ref = Reference.objects.create(attached_to=app.get_application())
        ref.overall = 0
        ref.save()
        ref = Reference.objects.get(attached_to=app)
        self.assertTrue(ref.overall in RELATIVE_RANK[ref.overall])

    def test_00490_applicationHasStatusField(self):
        app = Applicant.objects.create(username="app")
        app.get_application().status = 0
        app.get_application().save()
        self.assertEqual(app.get_application().status, 0)

    def test_00500_profileModelExists(self):
        app = Applicant.objects.create(username="app")
        app_profile = app.get_profile()
#        GwaapProfile.objects.create(applicant_profile=app_profile)
        self.assertIsInstance(GwaapProfile.objects.get(applicant_profile=app_profile), GwaapProfile)

    def test_00510_applicantHasConvenienceMethodForProfile(self):
        app = Applicant.objects.create(username="app")
#        GwaapProfile.objects.create(applicant_profile=app.get_profile())
        gwaap_profile = app.get_gwaap_profile()
        self.assertIsInstance(gwaap_profile, GwaapProfile)

    def test_00520_profileHasMiddleName(self):
        app = Applicant.objects.create(username="app")
#        GwaapProfile.objects.create(applicant_profile=app.get_profile())
        p = app.get_gwaap_profile()
        p.middle_name = "middle"
        p.save()
        self.assertEqual(app.get_gwaap_profile().middle_name, "middle")

    def test_00530_profileHasStreet1(self):
        app = Applicant.objects.create(username="app")
#        GwaapProfile.objects.create(applicant_profile=app.get_profile())
        p = app.get_gwaap_profile()
        p.street1 = "middle"
        p.save()
        self.assertEqual(app.get_gwaap_profile().street1, "middle")

    def test_00540_profileHasStreet2(self):
        app = Applicant.objects.create(username="app")
#        GwaapProfile.objects.create(applicant_profile=app.get_profile())
        p = app.get_gwaap_profile()
        p.street2 = "middle"
        p.save()
        self.assertEqual(app.get_gwaap_profile().street2, "middle")

    def test_00550_profileHasCity(self):
        app = Applicant.objects.create(username="app")
```

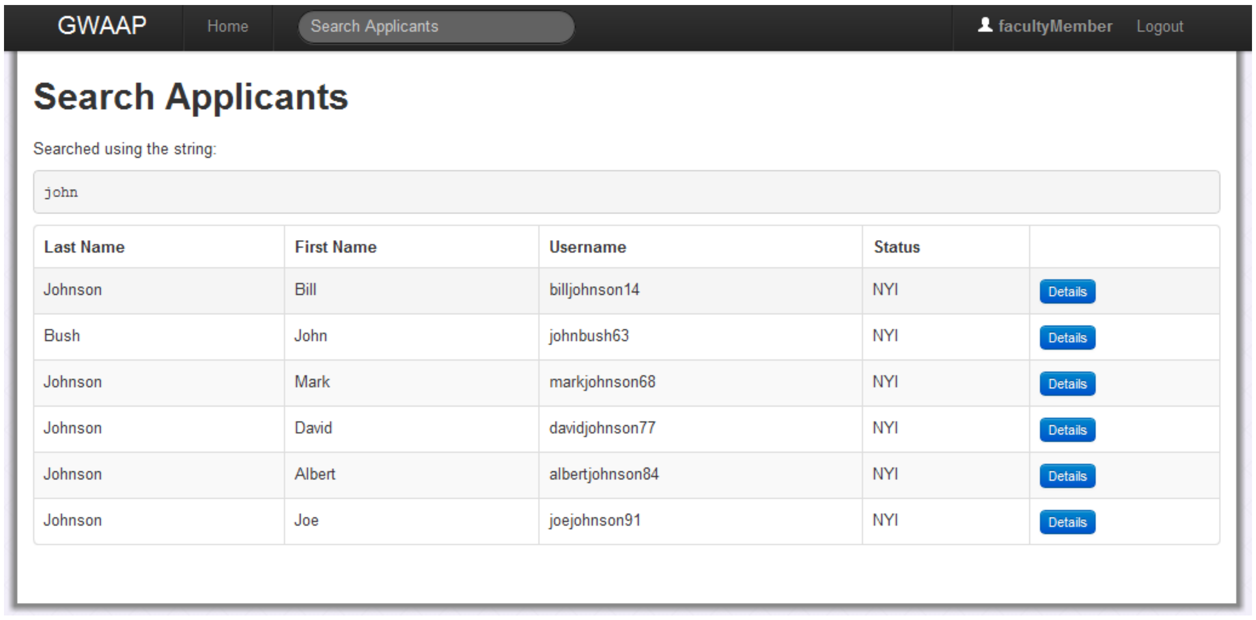
```python
#        GwaapProfile.objects.create(applicant_profile=app.get_profile())
        p = app.get_gwaap_profile()
        p.city = "middle"
        p.save()
        self.assertEqual(app.get_gwaap_profile().city, "middle")

    def test_00560_profileHasProvince(self):
        app = Applicant.objects.create(username="app")
#        GwaapProfile.objects.create(applicant_profile=app.get_profile())
        p = app.get_gwaap_profile()
        p.province = "middle"
        p.save()
        self.assertEqual(app.get_gwaap_profile().province, "middle")

    def test_00570_profileHasState(self):
        app = Applicant.objects.create(username="app")
#        GwaapProfile.objects.create(applicant_profile=app.get_profile())
        p = app.get_gwaap_profile()
        p.state = "middle"
        p.save()
        self.assertEqual(app.get_gwaap_profile().state, "middle")

    def test_00570_profileHasCountry(self):
        app = Applicant.objects.create(username="app")
#        GwaapProfile.objects.create(applicant_profile=app.get_profile())
        p = app.get_gwaap_profile()
        p.country = "middle"
        p.save()
        self.assertEqual(app.get_gwaap_profile().country, "middle")

    def test_00580_profileHasZip(self):
        app = Applicant.objects.create(username="app")
#        GwaapProfile.objects.create(applicant_profile=app.get_profile())
        p = app.get_gwaap_profile()
        p.zip_code = "middle"
        p.save()
        self.assertEqual(app.get_gwaap_profile().zip_code, "middle")

    def test_00590_profileHasTelephone(self):
        app = Applicant.objects.create(username="app")
#        GwaapProfile.objects.create(applicant_profile=app.get_profile())
        p = app.get_gwaap_profile()
        p.phone = "middle"
        p.save()
        self.assertEqual(app.get_gwaap_profile().phone, "middle")

    def test_00600_profileHasBirthday(self):
        app = Applicant.objects.create(username="app")
#        GwaapProfile.objects.create(applicant_profile=app.get_profile())
        p = app.get_gwaap_profile()
        p.birthday = "middle"
        p.save()
        self.assertEqual(app.get_gwaap_profile().birthday, "middle")

    def test_00610_profileHasGender(self):
        app = Applicant.objects.create(username="app")
#        GwaapProfile.objects.create(applicant_profile=app.get_profile())
```

```
        p = app.get_gwaap_profile()
        p.gender = "middle"
        p.save()
        self.assertEqual(app.get_gwaap_profile().gender, "middle")

    def test_00620_profileHasCountry_Birth(self):
        app = Applicant.objects.create(username="app")
#        GwaapProfile.objects.create(applicant_profile=app.get_profile())
        p = app.get_gwaap_profile()
        p.country_birth = "middle"
        p.save()
        self.assertEqual(app.get_gwaap_profile().country_birth, "middle")

    def test_00630_profileHasCitizenship(self):
        app = Applicant.objects.create(username="app")
#        GwaapProfile.objects.create(applicant_profile=app.get_profile())
        p = app.get_gwaap_profile()
        p.citizenship = "middle"
        p.save()
        self.assertEqual(app.get_gwaap_profile().citizenship, "middle")

    def test_00640_profileHasRef_Number(self):
        app = Applicant.objects.create(username="app")
#        GwaapProfile.objects.create(applicant_profile=app.get_profile())
        p = app.get_gwaap_profile()
        p.ref_number = "middle"
        p.save()
        self.assertEqual(app.get_gwaap_profile().ref_number, "middle")

    def test_00650_profileHasDate_Apply(self):
        app = Applicant.objects.create(username="app")
#        GwaapProfile.objects.create(applicant_profile=app.get_profile())
        p = app.get_gwaap_profile()
        p.date_apply = "middle"
        p.save()
        self.assertEqual(app.get_gwaap_profile().date_apply, "middle")

    def test_00660_profileHasEnter_Qtr(self):
        app = Applicant.objects.create(username="app")
#        GwaapProfile.objects.create(applicant_profile=app.get_profile())
        p = app.get_gwaap_profile()
        p.enter_qtr = "middle"
        p.save()
        self.assertEqual(app.get_gwaap_profile().enter_qtr, "middle")

    def test_00670_profileHasEnter_YEAR(self):
        app = Applicant.objects.create(username="app")
#        GwaapProfile.objects.create(applicant_profile=app.get_profile())
        p = app.get_gwaap_profile()
        p.enter_year = "middle"
        p.save()
        self.assertEqual(app.get_gwaap_profile().enter_year, "middle")

    def test_00670_profileHasDegree(self):
        app = Applicant.objects.create(username="app")
#        GwaapProfile.objects.create(applicant_profile=app.get_profile())
        p = app.get_gwaap_profile()
```

```python
        p.degree = "middle"
        p.save()
        self.assertEqual(app.get_gwaap_profile().degree, "middle")

    def test_00680_profileHasMajor(self):
        app = Applicant.objects.create(username="app")
#        GwaapProfile.objects.create(applicant_profile=app.get_profile())
        p = app.get_gwaap_profile()
        p.major = "middle"
        p.save()
        self.assertEqual(app.get_gwaap_profile().major, "middle")

    def test_00690_profileHasGRE_TAKEN(self):
        app = Applicant.objects.create(username="app")
#        GwaapProfile.objects.create(applicant_profile=app.get_profile())
        p = app.get_gwaap_profile()
        p.gre_taken = "middle"
        p.save()
        self.assertEqual(app.get_gwaap_profile().gre_taken, "middle")

    def test_00700_profileHasO_GRE_V(self):
        app = Applicant.objects.create(username="app")
#        GwaapProfile.objects.create(applicant_profile=app.get_profile())
        p = app.get_gwaap_profile()
        p.o_gre_v = "middle"
        p.save()
        self.assertEqual(app.get_gwaap_profile().o_gre_v, "middle")

    def test_00710_profileHasO_GRE_Q(self):
        app = Applicant.objects.create(username="app")
#        GwaapProfile.objects.create(applicant_profile=app.get_profile())
        p = app.get_gwaap_profile()
        p.o_gre_q = "middle"
        p.save()
        self.assertEqual(app.get_gwaap_profile().o_gre_q, "middle")

    def test_00720_profileHasO_GRE_A(self):
        app = Applicant.objects.create(username="app")
#        GwaapProfile.objects.create(applicant_profile=app.get_profile())
        p = app.get_gwaap_profile()
        p.o_gre_a = "middle"
        p.save()
        self.assertEqual(app.get_gwaap_profile().o_gre_a, "middle")

    def test_00730_profileHasO_GRE_W(self):
        app = Applicant.objects.create(username="app")
#        GwaapProfile.objects.create(applicant_profile=app.get_profile())
        p = app.get_gwaap_profile()
        p.o_gre_w = "middle"
        p.save()
        self.assertEqual(app.get_gwaap_profile().o_gre_w, "middle")

    def test_00740_profileHasTOEFL_TAKEN(self):
        app = Applicant.objects.create(username="app")
#        GwaapProfile.objects.create(applicant_profile=app.get_profile())
        p = app.get_gwaap_profile()
        p.toefl_taken = "middle"
        p.save()
        self.assertEqual(app.get_gwaap_profile().toefl_taken, "middle")

    def test_00750_profileHasO_TOEFL_SCORE(self):
        app = Applicant.objects.create(username="app")
#        GwaapProfile.objects.create(applicant_profile=app.get_profile())
        p = app.get_gwaap_profile()
        p.o_toefl_score = "middle"
        p.save()
        self.assertEqual(app.get_gwaap_profile().o_toefl_score, "middle")

    def test_00760_applicantsGetProfilesAutomatically(self):
        app = Applicant.objects.create(username='app')
        p = app.get_gwaap_profile()
        self.assertIsInstance(p, GwaapProfile)


```

# Appendix C

## Screenshots

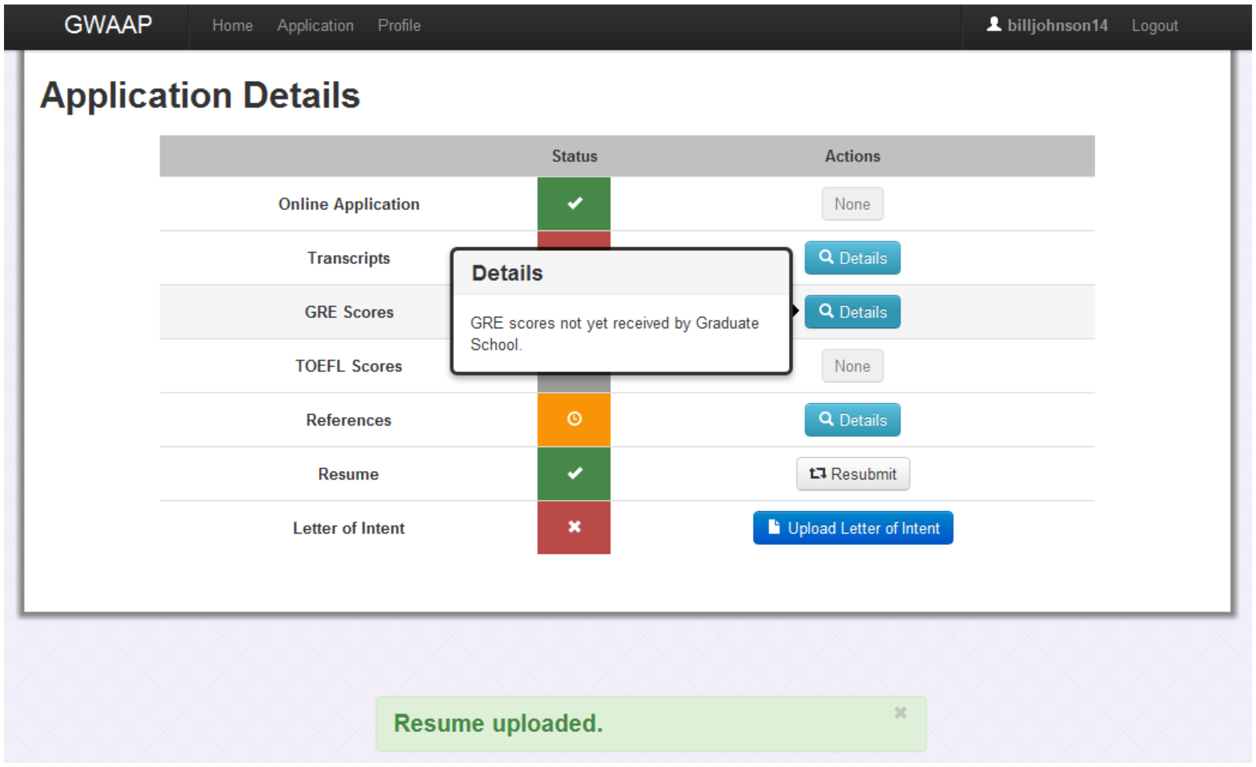


### Search Applicants

Searched using the string:

```
john
```

| Last Name | First Name | Username | Status | |
|---|---|---|---|---|
| Johnson | Bill | billjohnson14 | NYI | Details |
| Bush | John | johnbush63 | NYI | Details |
| Johnson | Mark | markjohnson68 | NYI | Details |
| Johnson | David | davidjohnson77 | NYI | Details |
| Johnson | Albert | albertjohnson84 | NYI | Details |
| Johnson | Joe | joejohnson91 | NYI | Details |



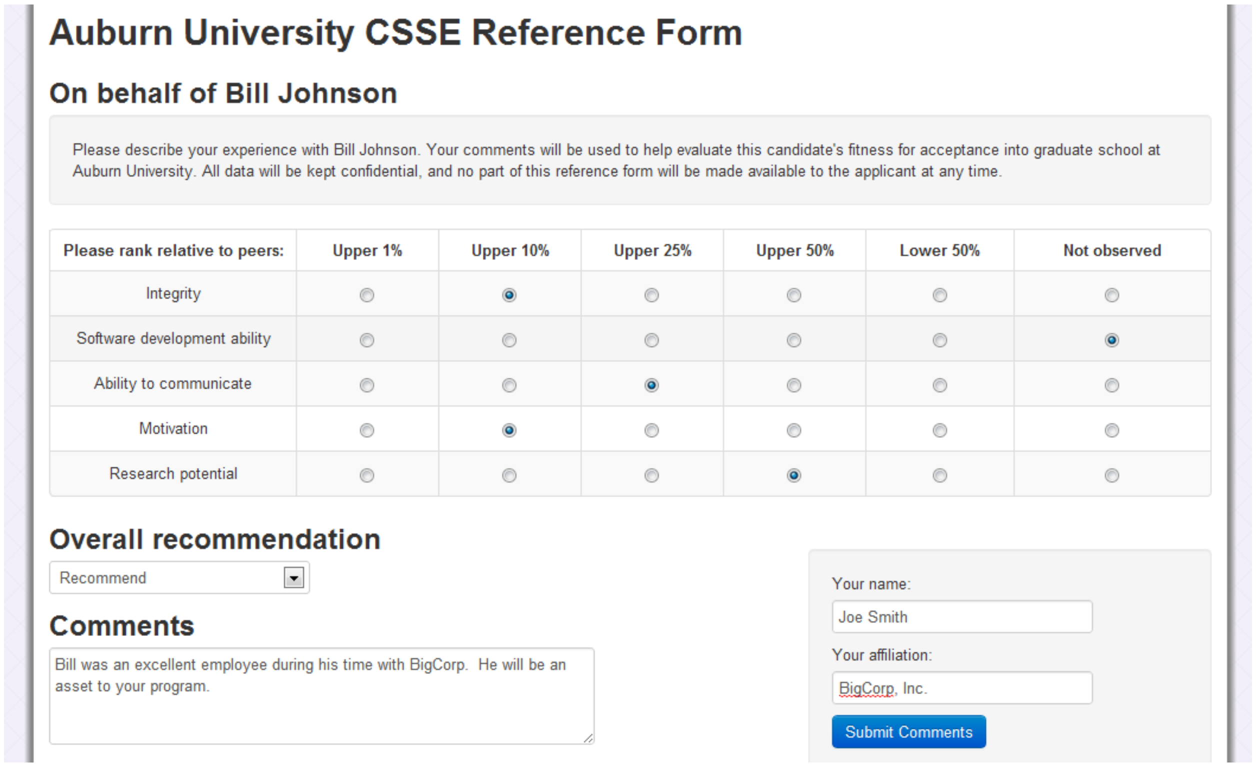
### Application Details

| | Status | Actions |
|---|---|---|
| Online Application | ✓ | None |
| Transcripts | 🟥 | Details |
| GRE Scores | | Details |
| TOEFL Scores | | None |
| References | 🟧 | Details |
| Resume | ✓ | Resubmit |
| Letter of Intent | ✖ | Upload Letter of Intent |

**Details**

GRE scores not yet received by Graduate School.

Resume uploaded.



## Auburn University CSSE Reference Form

### On behalf of Bill Johnson

Please describe your experience with Bill Johnson. Your comments will be used to help evaluate this candidate's fitness for acceptance into graduate school at Auburn University. All data will be kept confidential, and no part of this reference form will be made available to the applicant at any time.

| Please rank relative to peers: | Upper 1% | Upper 10% | Upper 25% | Upper 50% | Lower 50% | Not observed |
|---|---|---|---|---|---|---|
| Integrity | ○ | ● | ○ | ○ | ○ | ○ |
| Software development ability | ○ | ○ | ○ | ○ | ○ | ● |
| Ability to communicate | ○ | ○ | ● | ○ | ○ | ○ |
| Motivation | ○ | ● | ○ | ○ | ○ | ○ |
| Research potential | ○ | ○ | ○ | ● | ○ | ○ |

### Overall recommendation

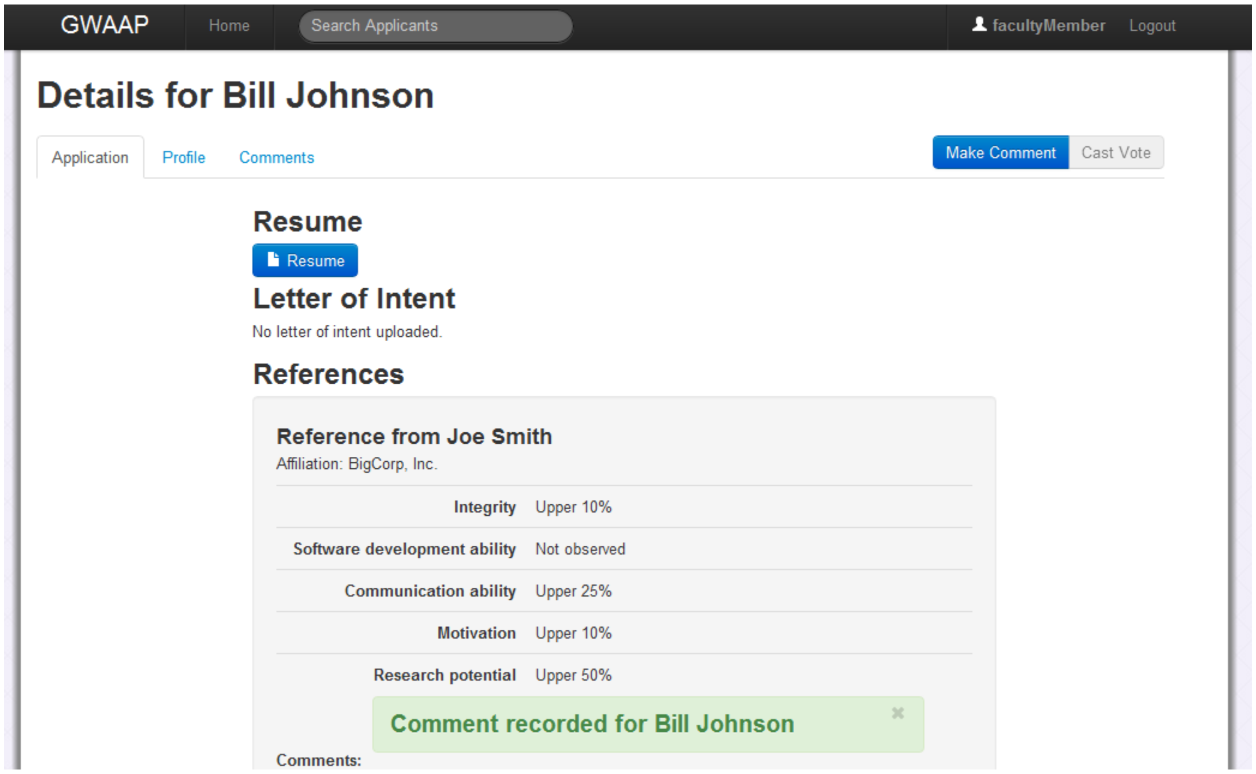
Recommend

### Comments

Bill was an excellent employee during his time with BigCorp. He will be an asset to your program.

Your name: Joe Smith

Your affiliation: BigCorp, Inc.



## Details for Bill Johnson

### Resume

### Letter of Intent

No letter of intent uploaded.

### References

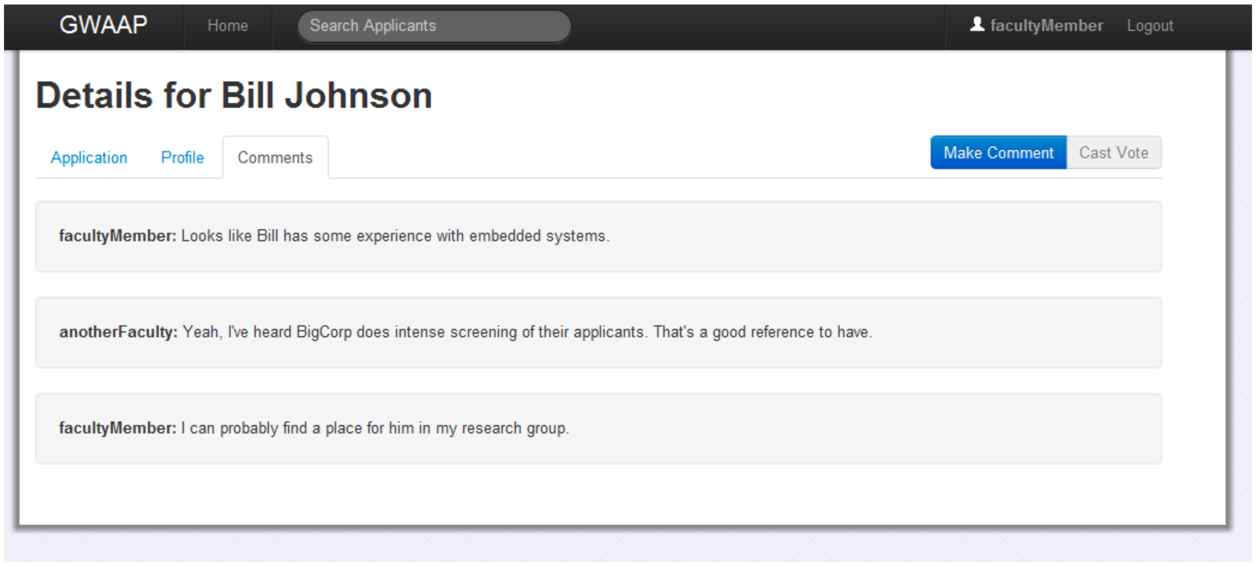
**Reference from Joe Smith**

Affiliation: BigCorp, Inc.

| | |
|---|---|
| **Integrity** | Upper 10% |
| **Software development ability** | Not observed |
| **Communication ability** | Upper 25% |
| **Motivation** | Upper 10% |
| **Research potential** | Upper 50% |

Comment recorded for Bill Johnson

**Comments:**



Details for Bill Johnson

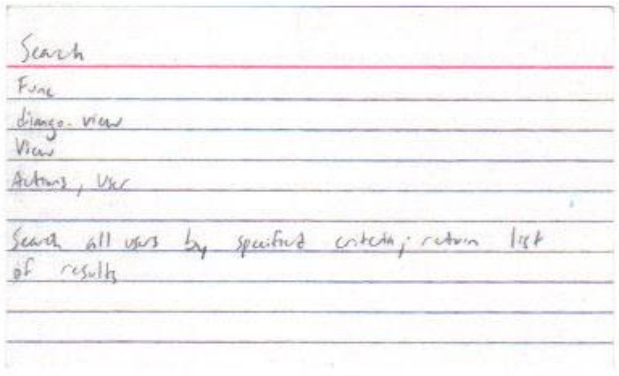
**facultyMember:** Looks like Bill has some experience with embedded systems.

**anotherFaculty:** Yeah, I've heard BigCorp does intense screening of their applicants. That's a good reference to have.

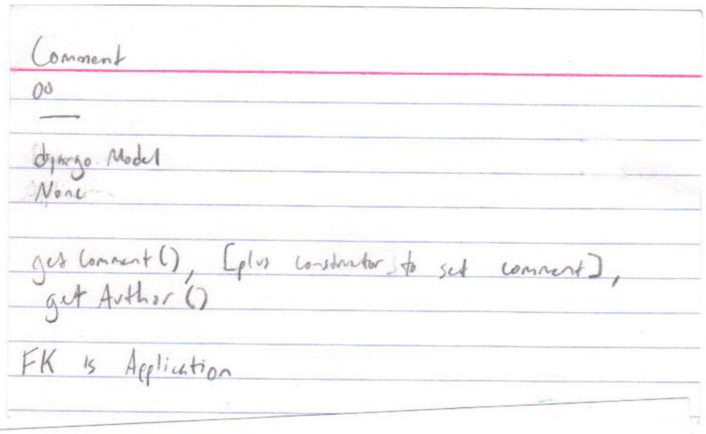
**facultyMember:** I can probably find a place for him in my research group.

# Appendix D

## PCSE Process Documentation

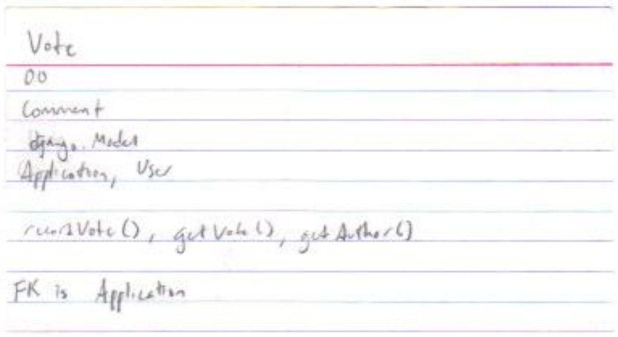

Search
Func
django. view
View
Actions, User

Search all users by specified criteria; return list of results


Comment
OO
—
django. Model
None

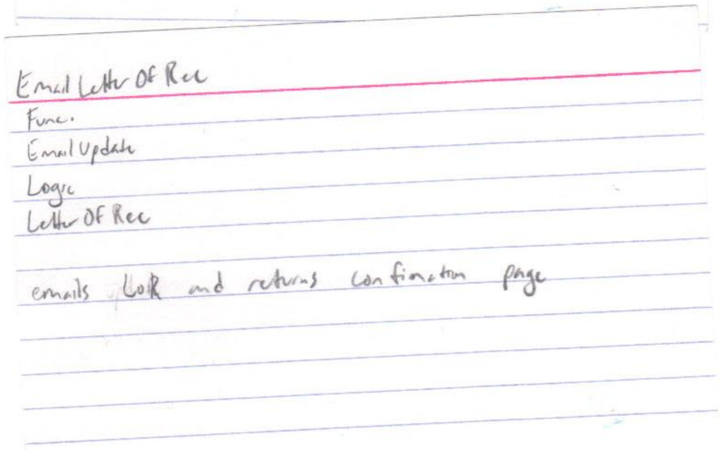
get Comment(), [plus constructor to set comment],
get Author()

FK is Application

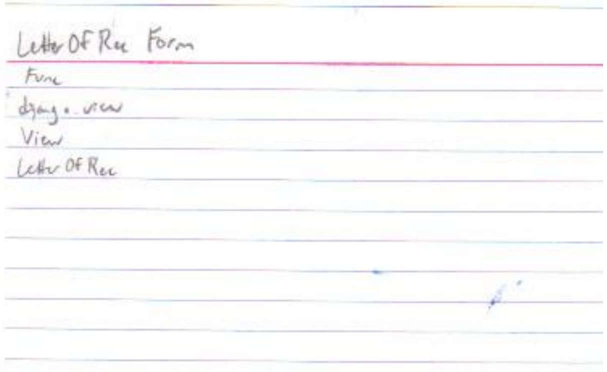

Vote
OO
Comment
django. Model
Application, User

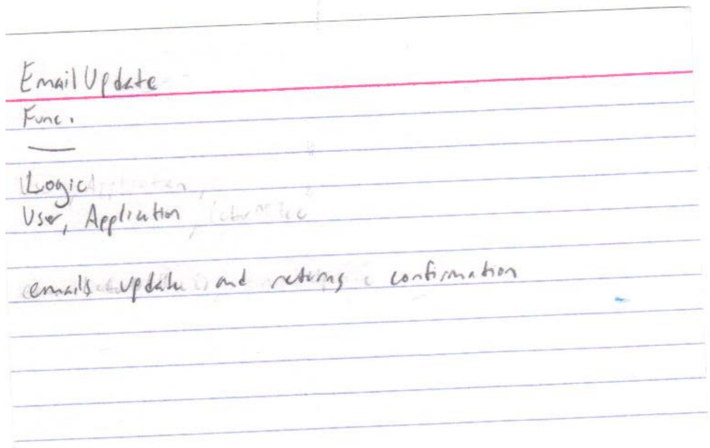
read Vote(), get Vote(), get Author()

FK is Application


Email Letter Of Rec
Func.
Email Update
Logic
Letter Of Rec

emails LoR and returns confirmation page


Letter Of Rec Form
Func
django. view
View
Letter Of Rec


Email Update
Func.
—
Logic
User, Application

emails update and returns confirmation

Update Application Progress
Func.
django.view
View
Application, User

update application status, send confirmation emails,
send confirmation page

Cast Vote
Func.
django.view
View
User, Vote, Application

register vote and return confirmation

Make Comment
Func.
django.view
View
Comment, Application, User

Record comment appropriately, return confirmation

Get GWAAP Dump
Func.
django.view
View
Applicant, Application

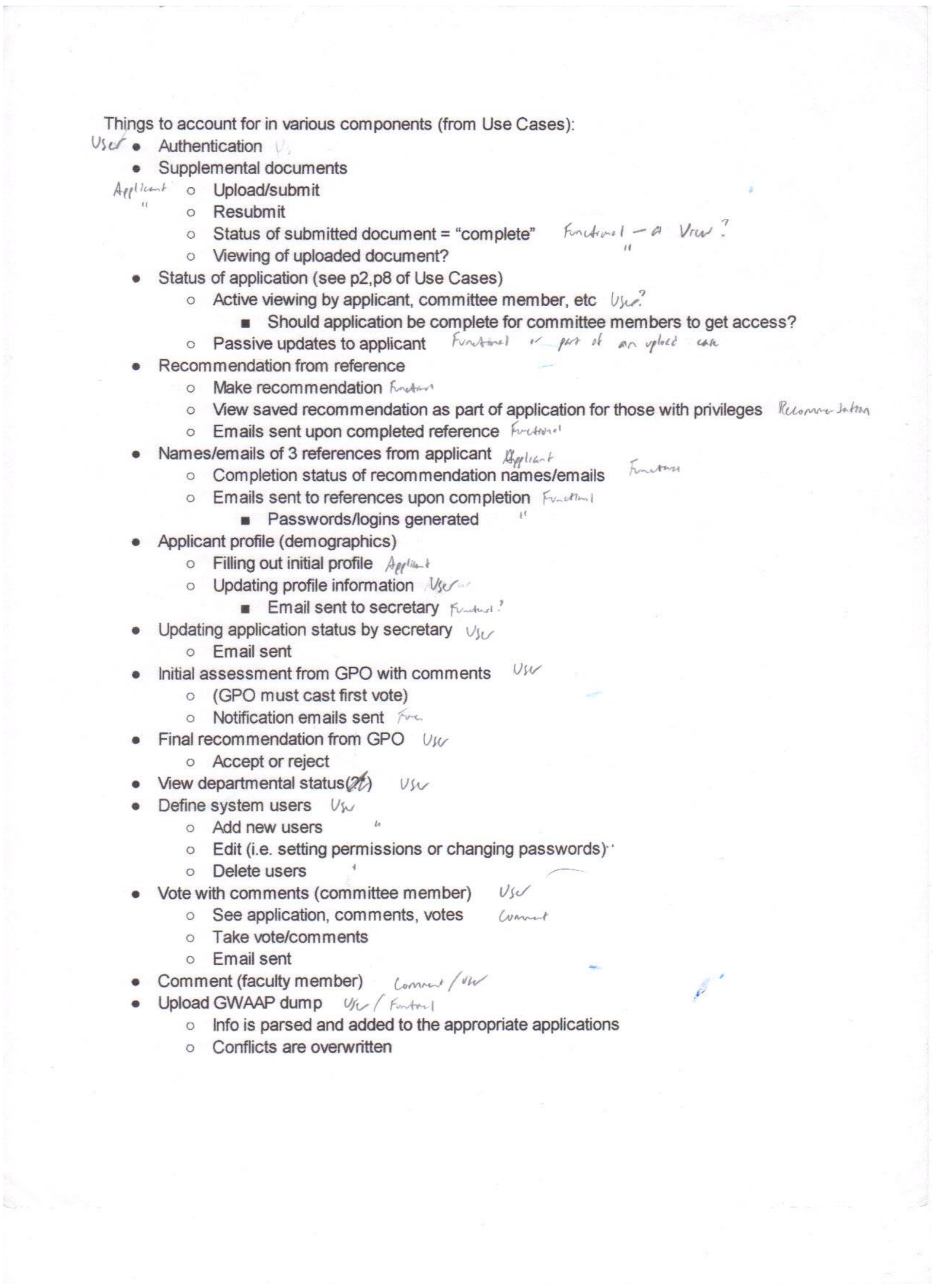
starts a new Applicant, emails confirmations

Write Comment
Func.
django.view
View
Comment, Application, User

Provide a form and accept comments

Letter Of Rec
OO
———
django Model
Application
recordResponses(), getResponses(), [any miscellaneous processing
methods needed]

ForeignKey is an Application?



Things to account for in various components (from Use Cases):

- User Authentication Us
- Supplemental documents
  - Applicant Upload/submit
  - " Resubmit
  - Status of submitted document = "complete" Functional — a View?
  - Viewing of uploaded document? "
- Status of application (see p2,p8 of Use Cases)
  - Active viewing by applicant, committee member, etc User?
    - Should application be complete for committee members to get access?
  - Passive updates to applicant Functional or part of an upload case
- Recommendation from reference
  - Make recommendation Functional
  - View saved recommendation as part of application for those with privileges Recommendation
  - Emails sent upon completed reference Functional
- Names/emails of 3 references from applicant Applicant
  - Completion status of recommendation names/emails Functional
  - Emails sent to references upon completion Functional
    - Passwords/logins generated "
- Applicant profile (demographics)
  - Filling out initial profile Applicant
  - Updating profile information User
    - Email sent to secretary Functional?
- Updating application status by secretary User
  - Email sent
- Initial assessment from GPO with comments User
  - (GPO must cast first vote)
  - Notification emails sent Func.
- Final recommendation from GPO User
  - Accept or reject
- View departmental status(~~2~~) User
- Define system users User
  - Add new users "
  - Edit (i.e. setting permissions or changing passwords) "
  - Delete users "
- Vote with comments (committee member) User
  - See application, comments, votes Comment
  - Take vote/comments
  - Email sent
- Comment (faculty member) Comment / User
- Upload GWAAP dump User / Functional
  - Info is parsed and added to the appropriate applications
  - Conflicts are overwritten

- Applicants not previously registered are sent welcome email

Authentication: could a user ever need to fulfill more than one exclusive role?

- Applicants really could never be in the system
- GPO can comment as a faculty member if desired; no problem there
- Administrative tasks could be handled by people with other permissions or people without

...so, no, not really.

Models can have methods

Views just pull from models and optionally stick stuff into a template

No passively-activated functionality
↳ everything can be activated via a View

In this system, Views are really just where the system comes up for input or output

## Permission Flags

Most work is done in one method, either in View, or in the User base class

↳ assemble available options

Either have one User class and set permissions at creation time

OR subclass User and each class sets appropriate permissions in its constructor

No duplication of code

↳ But, if you want to present a different UI entirely to different types of Users, hard to do that w/ this method

Entirely different method (one class per user type)

Each class gets a presentOptions() method or some such

Class actually composes and returns [illegible] a response and
[illegible] links that could be implemented elsewhere

But, this definitely violates MVC distinctions

Motivation:

The Controller for all these options will not be present in the model class. The Model only really has to present what options are available.

All has to be used by a View that will tell
what to display

View could call "get Permissions" and [illegible] that gets posted

View could call all different possible permissions and assemble
the result (via permission flags or method calls for each
possible permission)

## 3 OPTIONS

- Permission flags
- Single-inheritance (increasingly permissive subclasses)
- Multiple-inheritance (one class for every permission

→ Treat permissions as a list that gets populated

→ Present all permissions and disable the ones that
aren't accessible

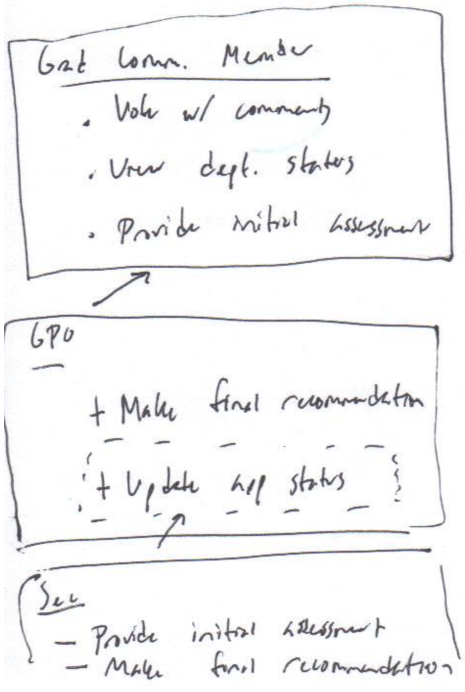
## Single Inheritance

The big theoretical pro of SI is that you only have to implement each method once and then you get it for free by calling up the super() chain. Every subclass just adds more functionality while keeping the old.

This is also the con. If functionality is derived from a class higher in the hierarchy, ~~[illegible]~~ subclasses have to remove the functionality if they don't need it.

BAD

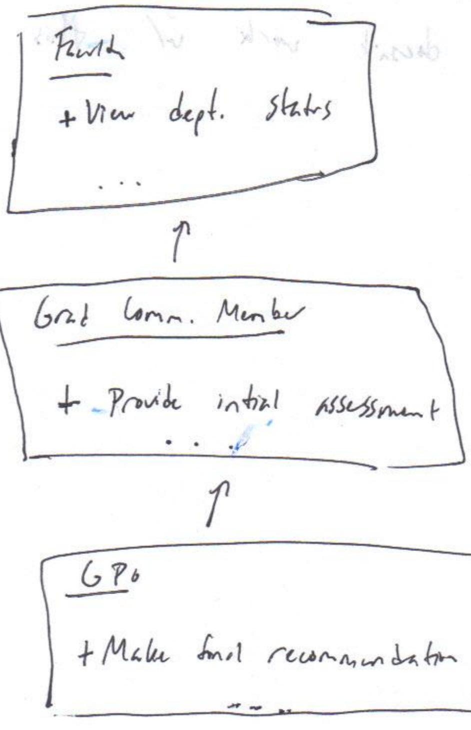
Example: Update application status



Grad Comm. Member
. Vote w/ comments
. View dept. status
. Provide initial assessment
GPO
+ Make final recommendation
+ Update app status
Sec
− Provide initial assessment
− Make final recommendation

GOOD

Example: Provide initial assessment



Pros:

In a scheme where SI works, it's very elegant.

↳ Call getPermissions(permissions[])

Class calls super.getPermissions(), and every base class it inherits from adds permissions to the list

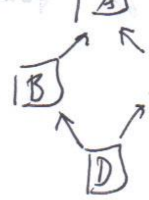
↳ minimum # of classes

Cons:

It doesn't work w/ this system

## Multiple Inheritance

Would be nice ~~to~~ call getPermissions() and get a full list of all permissions from all ~~base~~ base classes, but you get the diamond problem.



Once the method resolves to any class w/ an implementation, it's over.

Other option w/ multiple inheritance would be calling a method that represents each permission, and only implement each of those methods in a single base class. If an object implements the method, it must derive from that base class and thus have that permission.

Side effect
Lots of classes — one for every permission, plus one for each possible combination of permissions

☆ ~~[illegible]~~ Use of Python's "super()" built-in makes MI approach possible, however, because it knows how to traverse the entire MRO.

Pros:

Very clear what permissions any given user type has, clear at-a-glance

Easy to add new user types or take permissions away from all user types at once.

Cons:

Quite difficult to use in practice

↳ Have to call a method representing each type of permission and handle error if it doesn't implement that method.

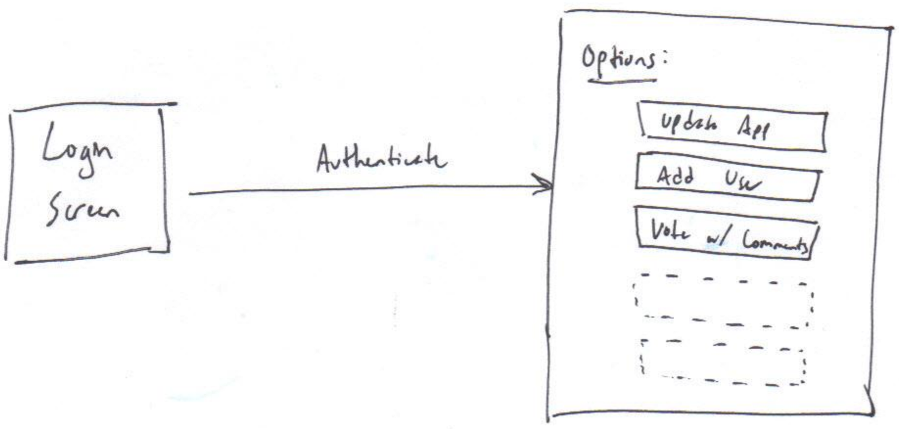
The other way would be to have each base class constructor set a permission flag during construction

↳ But then why not just use permission flags

A View will present these options to the user.

→ So, ultimately, the Model needs to be responsible for just reporting what kinds of things it has available.

~~Authenticate~~



So, View needs a list of possible permissions, OR to cycle through all possible permissions and check for them.

## Timetable

## Component-Iteration Map

| Component | Iteration 1 | Iteration 2 | Iteration 3 | Iteration 4 | Iteration 5 | Iteration 6 |
|---|---|---|---|---|---|---|
| Application | Mockup | Mockup | | | | |
| ApplicantActions | | Mockup | Production | | | |
| ApplicantLogin | | Mockup | Production | | | |
| User | Mockup | Production | | | | |
| Applicant | Mockup | Production | | | | Production |
| ViewApplicant (User) | | | | Production | | |
| UserActions | | Mockup | | | | |
| UserLogin | | Mockup | Production | | | |
| DisplayApplicants | | Mockup | | Production | | |
| ApplicantProfile | | Production | | | | |
| NameReference | | | Production | | | Production |
| Reference | | | Production | | | |
| CompleteReference | | | Mockup | Mockup | | Production |
| Comment | | | | Production | | |
| Vote | | | | Production | | |
| MakeComment | | | | Production | | |
| CastVote | | | | Production | | |
| ViewApplication (Applicant) | | | | | Mockup | Production |
| Base Template | | | | | Production | |
| UserLogin Template | | | | | Production | |
| ApplicantLogin Template | | | | | Production | |
| UserActions Template | | | | | Production | |
| ApplicantHome Template | | | | | Mockup | Production |
| DisplayApplicants Template | | | | | Production | |
| ViewApplicant Template | | | | | Production | |
| ViewApplication Template | | | | | Production | |
| AddReference Template | | | | | Production | |
| CompleteReference Template | | | | | Production | |
| MakeComment Template | | | | | Production | |
| CastVote Template | | | | | Production | |
| SearchApplicants View | | | | | Production | |
| SearchApplicants Template | | | | | Production | |
| ApplicantProfile | | | | | | Production |
| Template graphics/polish | | | | | | Mockup |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | 0 | 0 | 0 | 0 | 0 | 0 |

## WBS

| Iteration Number | Planned Effort (minutes) | Cumulative Planned Effort | Planned Velocity | Cumulative Planned Velocity | Planned Completion Date |
|---|---|---|---|---|---|
| 1 | 120 | 120 | 3 | 3 | 11/25/2011 |

| | | | | | |
|---|---|---|---|---|---|
| 2 | 600 | 720 | 10 | 13 | 2/21/2012 |
| 3 | 360 | 1080 | 6 | 19 | 3/6/2012 |
| 4 | 630 | 1710 | 7 | 26 | 3/18/2012 |
| 5 | 390 | 2100 | 12 | 38 | 3/26/2012 |
| 6 | 300 | 2400 | 7 | 45 | 3/31/2012 |
| 7 | | 2400 | | 45 | |
| 8 | | 2400 | | 45 | |
| 9 | | 2400 | | 45 | |
| 10 | | 2400 | | 45 | |

## Calendar

| Day # | Date | Available Minutes | Cumulative Minutes | Planned Velocity | Cumulative Planned Velocity |
|---|---|---|---|---|---|
| 1 | 11/7/2011 | 0 | 0 | | 0 |
| 2 | 11/8/2011 | 0 | 0 | | 0 |
| 3 | 11/9/2011 | 0 | 0 | | 0 |
| 4 | 11/10/2011 | 0 | 0 | | 0 |
| 5 | 11/11/2011 | 0 | 0 | | 0 |
| 6 | 11/12/2011 | 60 | 60 | 1 | 1 |
| 7 | 11/13/2011 | 60 | 120 | 1 | 2 |
| 8 | 2/6/2012 | 90 | 210 | 1 | 3 |
| 9 | 2/7/2012 | 30 | 240 | 1 | 4 |
| 10 | 2/8/2012 | | 240 | | 4 |
| 11 | 2/9/2012 | | 240 | | 4 |
| 12 | 2/10/2012 | 90 | 330 | 1 | 5 |
| 13 | 2/11/2012 | 60 | 390 | 1 | 6 |
| 14 | 2/12/2012 | 30 | 420 | 1 | 7 |
| 15 | 2/13/2012 | 90 | 510 | 1 | 8 |
| 16 | 2/14/2012 | | 510 | | 8 |
| 17 | 2/15/2012 | 30 | 540 | 1 | 9 |
| 18 | 2/16/2012 | | 540 | | 9 |
| 19 | 2/17/2012 | 90 | 630 | 1 | 10 |
| 20 | 2/18/2012 | | 630 | | 10 |
| 21 | 2/19/2012 | | 630 | | 10 |
| 22 | 2/20/2012 | 90 | 720 | 2 | 12 |
| 23 | 2/21/2012 | 60 | 780 | 1 | 13 |
| 24 | 2/22/2012 | | 780 | | 13 |
| 25 | 2/23/2012 | | 780 | | 13 |
| 26 | 2/24/2012 | | 780 | | 13 |
| 27 | 2/25/2012 | | 780 | | 13 |
| 28 | 2/26/2012 | | 780 | | 13 |
| 29 | 2/27/2012 | 90 | 870 | 1 | 14 |
| 30 | 2/28/2012 | | 870 | | 14 |
| 31 | 2/29/2012 | 60 | 930 | | 14 |
| 32 | 3/1/2012 | 60 | 990 | 1 | 15 |
| 33 | 3/2/2012 | 60 | 1050 | 1 | 16 |
| 34 | 3/3/2012 | 90 | 1140 | 1 | 17 |
| 35 | 3/4/2012 | 90 | 1230 | 1 | 18 |
| 36 | 3/5/2012 | 90 | 1320 | 1 | 19 |

| | | | | | |
|---|---|---|---|---|---|
| 37 | 3/6/2012 | | 1320 | | 19 |
| 38 | 3/7/2012 | | 1320 | | 19 |
| 39 | 3/8/2012 | | 1320 | | 19 |
| 40 | 3/9/2012 | | 1320 | | 19 |
| 41 | 3/10/2012 | | 1320 | | 19 |
| 42 | 3/11/2012 | | 1320 | | 19 |
| 43 | 3/12/2012 | 90 | 1410 | 1 | 20 |
| 44 | 3/13/2012 | 90 | 1500 | 1 | 21 |
| 45 | 3/14/2012 | 90 | 1590 | 1 | 22 |
| 46 | 3/15/2012 | 90 | 1680 | 1 | 23 |
| 47 | 3/16/2012 | 90 | 1770 | 1 | 24 |
| 48 | 3/17/2012 | 90 | 1860 | 1 | 25 |
| 49 | 3/18/2012 | 90 | 1950 | 1 | 26 |
| 50 | 3/19/2012 | 90 | 2040 | 1 | 27 |
| 51 | 3/20/2012 | 60 | 2100 | 0 | 27 |
| 52 | 3/21/2012 | 30 | 2130 | 0 | 27 |
| 53 | 3/22/2012 | 60 | 2190 | 1 | 28 |
| 54 | 3/23/2012 | | 2190 | | 28 |
| 55 | 3/24/2012 | | 2190 | | 28 |
| 56 | 3/25/2012 | 75 | 2265 | | 28 |
| 57 | 3/26/2012 | 60 | 2325 | 1 | 29 |
| 58 | 3/27/2012 | 120 | 2445 | 2 | 31 |
| 59 | 3/28/2012 | 60 | 2505 | 1 | 32 |
| 60 | 3/29/2012 | 60 | 2565 | 1 | 33 |
| 61 | 3/30/2012 | 60 | 2625 | 1 | 34 |
| 62 | 3/31/2012 | 60 | 2685 | 1 | P |
| 63 | 4/1/2012 | | 2685 | | #VALUE! |
| 64 | 4/2/2012 | | 2685 | | #VALUE! |

## Time Recording Log

| Date | Start Time | Stop Time | Interrupt | Delta | Activity | Iteration | Comments |
|---|---|---|---|---|---|---|---|
| 9/28/2011 | 12:15 AM | 1:30 AM | 15 | 60 | Architecture | NA | |
| 9/29/2011 | 12:30 AM | 1:45 AM | | 75 | Architecture | NA | |
| 10/24/2011 | 12:00 PM | 1:00 PM | 15 | 45 | Architecture | NA | |
| 11/6/2011 | 10:15 PM | 11:00 PM | | 45 | Planning | 1 | |
| 11/7/2011 | 12:30 PM | 1:30 PM | 15 | 45 | Planning | 1 | |
| 11/24/2011 | 9:45 PM | 10:45 PM | | 60 | Construction | 1 | Worked on User class; found django User base class |
| 11/26/2011 | 8:40 PM | 9:30 PM | 15 | 35 | Construction | 1 | Worked on Applicant/Application relationship |
| 11/28/2011 | 12:55 PM | 1:20 PM | | 25 | Sandbox | 1 | Corrected for change to Applicant class |
| 2/5/2012 | 9:00 PM | 9:45 PM | | 45 | Planning | 2 | |
| 2/12/2012 | 7:45 PM | 9:30 PM | 5 | 100 | Construction | 2 | Worked on ApplicantProfile class |
| 2/19/2012 | 11:00 AM | 12:00 PM | 15 | 45 | Construction | 2 | ApplicantProfile |
| 2/19/2012 | 12:00 PM | 12:15 PM | | 15 | Sandbox | 2 | ApplicantProfile sandbox time |
| 2/19/2012 | 12:15 PM | 12:40 PM | | 25 | Construction | 2 | ApplicantProfile |
| 2/19/2012 | 1:50 PM | 2:50 PM | 20 | 40 | Sandbox | 2 | |
| 2/19/2012 | 4:00 PM | 5:50 PM | 20 | 90 | Construction | 2 | Refactoring Applicant creation to work around signals bug |
| 2/19/2012 | 6:00 PM | 6:50 PM | | 50 | Sandbox | 2 | User login view |
| 2/21/2012 | 3:30 PM | 5:20 PM | 15 | 95 | Construction | 2 | User/Applicant login views |
| 2/26/2012 | 10:00 PM | 10:30 PM | | 30 | Planning | 3 | |
| 3/3/2012 | 4:00 PM | 4:45 PM | 10 | 35 | Sandbox | 3 | Sandboxing User/Applicant change |
| 3/3/2012 | 7:30 PM | 8:40 PM | | 70 | Sandbox | 3 | Sandboxing User/Applicant change and permissions |
| 3/4/2012 | 1:30 PM | 2:00 PM | | 30 | Sandbox | 3 | Sandboxing redirects |
| 3/4/2012 | 4:00 PM | 4:30 PM | | 30 | Construction | 3 | Implementing User/Applicant changes |
| 3/4/2012 | 8:00 PM | 9:10 PM | 10 | 60 | Sandbox | 3 | Sandboxing login forms |
| 3/5/2012 | 9:30 AM | 10:00 AM | 5 | 25 | Construction | 3 | Implementing login forms |
| 3/5/2012 | 12:30 PM | 1:20 PM | 5 | 45 | Construction | 3 | Implementing login forms |
| 3/5/2012 | 2:30 PM | 3:00 PM | 10 | 20 | Sandbox | 3 | Sandboxing Reference model and view |
| 3/5/2012 | 5:30 PM | 6:30 PM | | 60 | Sandbox | 3 | Sandboxing Reference emailing |
| 3/6/2012 | 11:15 AM | 11:50 AM | 10 | 25 | Sandbox | 3 | Sandboxing Reference emailing |
| 3/6/2012 | 11:50 AM | 12:10 PM | | 20 | Construction | 3 | Reference model and additions to View |
| 3/6/2012 | 2:00 PM | 2:30 PM | 10 | 20 | Sandbox | 3 | Sending reference requests to references |
| 3/6/2012 | 6:10 PM | 6:30 PM | | 20 | Sandbox | 3 | Preparing for reference email implementation |
| 3/6/2012 | 6:30 PM | 7:20 PM | 10 | 40 | Construction | 3 | CompleteReference implementation |
| 3/6/2012 | 9:30 PM | 11:30 PM | 30 | 90 | Construction | 3 | CompleteReference implementation |
| 3/17/2012 | 4:00 PM | 4:30 PM | | 30 | Refactoring | 3 | Fixing broken things with User models |
| 3/17/2012 | 4:40 PM | 5:10 PM | | 30 | Refactoring | 3 | Adding unicode methods, making admin usable |
| 3/17/2012 | 5:20 PM | 5:50 PM | 10 | 20 | Planning | 4 | |
| 3/17/2012 | 6:10 PM | 6:45 PM | | 35 | Construction | 4 | Implementing Comment and Vote models |
| 3/17/2012 | 9:15 PM | 10:30 PM | 10 | 65 | Construction | 4 | Implementing views to show all applicants and specific applicants |
| 3/18/2012 | 2:45 PM | 3:50 PM | 10 | 55 | Construction | 4 | Comment and Vote views |
| 3/18/2012 | 3:50 PM | 4:15 PM | | 25 | Refactoring | 4 | Solidifying CompleteReference view |
| 3/18/2012 | 4:15 PM | 4:30 PM | | 15 | Review | 4 | |
| 3/18/2012 | 7:00 PM | 7:15 PM | | 15 | Planning | 5 | |
| 3/19/2012 | 5:00 AM | 5:30 AM | | 30 | Sandbox | 5 | Templates |
| 3/19/2012 | 5:30 AM | 7:15 AM | 45 | 60 | Construction | 5 | Base, User Section, User Login, User Home templates |
| 3/19/2012 | 8:45 AM | 9:45 AM | 10 | 50 | Construction | 5 | Implementing Django messages |
| 3/19/2012 | 12:15 PM | 2:00 PM | 45 | 60 | Construction | 5 | Polishing messages, adding CSRF protection |
| 3/19/2012 | 2:30 PM | 4:30 PM | 30 | 90 | Construction | 5 | More templates |
| 3/19/2012 | 5:15 PM | 8:00 PM | 40 | 125 | Construction | 5 | Finished templates, adding UX |
| 3/19/2012 | 8:30 PM | 8:45 PM | | 15 | Review | 5 | Fixing database relationship b/w User and Comment/Vote |
| 3/20/2012 | 9:00 AM | 10:30 AM | 15 | 75 | Construction | 5 | Implementing Applicant Search for Users |
| 3/20/2012 | 11:15 AM | 11:40 AM | | 25 | Construction | 5 | Applicant search |
| 3/20/2012 | 11:40 AM | 11:50 AM | | 10 | Refactoring | 5 | Converting applicant display table to a template that can be included |
| 3/20/2012 | 12:40 PM | 12:50 PM | | 10 | Review | 5 | Updating some lessons learned |
| 3/20/2012 | 12:50 PM | 1:30 PM | 15 | 25 | Planning | 5 | Designing View Application UI |
| 3/20/2012 | 1:30 PM | 2:45 PM | 45 | 30 | Construction | 5 | First steps on View Application |
| 3/20/2012 | 7:30 PM | 8:30 PM | | 60 | Construction | 5 | View Application and template |
| 3/20/2012 | 8:30 PM | 9:00 PM | 5 | 25 | Review | 5 | Evaluating system |
| 3/21/2012 | 10:40 AM | 11:30 AM | | 50 | Planning | 6 | |
| 3/26/2012 | 9:30 PM | 9:40 PM | | 10 | Planning | 6 | Identifying components |
| 3/26/2012 | 10:00 PM | 11:00 PM | | 60 | Sandbox | 6 | Sandboxing file uploads |
| 3/27/2012 | 1:30 PM | 2:30 PM | 30 | 30 | Construction | 6 | Adding resume and letter of intent to models, views, templates |
| 3/27/2012 | 6:00 PM | 7:00 PM | | 60 | Construction | 6 | |
| 3/27/2012 | 9:00 PM | 10:15 PM | | 75 | Construction | 6 | Adding smart recognition of file uploads to application details |
| 3/28/2012 | 9:00 AM | 10:45 AM | 30 | 75 | Construction | 6 | Adding additional status flags to application fields |
| 3/28/2012 | 3:30 PM | 5:30 PM | 15 | 105 | Construction | 6 | Readying reference system for deployment |
| 3/28/2012 | 5:30 PM | 7:00 PM | | 90 | Construction | 6 | Updating applicant home page with status display |
| 3/29/2012 | 11:30 AM | 12:00 PM | | 30 | Construction | 6 | GWAAP profile information model, view, and template |
| 3/30/2012 | 10:45 AM | 1:00 PM | | 135 | Construction | 6 | Making faculty UI more friendly for commenting/voting on applicants |
| 3/30/2012 | 5:00 PM | 5:30 PM | 10 | 20 | Construction | 6 | Adding checks to see if faculty member has already voted |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |

## Architecture

| | |
|---|---|
| Component Name: | Application |
| Design Approach: | Object-oriented |
| Parent Component: | django.Model |
| Component Type: | Model |
| Collaborators: | LetterOfRec, Comment, Vote |
| Operations: | [Django built-ins] |

| | |
|---|---|
| Component Name: | ApplicantActions |
| Design Approach: | Functional |
| Parent Component: | django.View |
| Component Type: | View |
| Collaborators: | Applicant |
| Operations: | displayApplicantActions |

| | |
|---|---|
| Component Name: | ApplicantLogin |
| Design Approach: | Functional |
| Parent Component: | django.View |
| Component Type: | View |
| Collaborators: | Applicant, ApplicantActions |
| Operations: | displayApplicantLogin |

| | |
|---|---|
| Component Name: | User |
| Design Approach: | Object-oriented |
| Parent Component: | django.Model |
| Component Type: | Model |
| Collaborators: | Applicant, Comment, Vote |
| Operations: | [Django built-ins] |

| | |
|---|---|
| Component Name: | Applicant |
| Design Approach: | Object-oriented |
| Parent Component: | django.Model |
| Component Type: | Model |
| Collaborators: | Application |
| Operations: | [Django built-ins] |

| | |
|---|---|
| Component Name: | ViewApplication |
| Design Approach: | Functional |
| Parent Component: | django.View |
| Component Type: | View |
| Collaborators: | Application |
| Operations: | displayApplication |

| | |
|---|---|
| Component Name: | UserActions |
| Design Approach: | Functional |
| Parent Component: | django.View |
| Component Type: | View |
| Collaborators: | ViewApplication, Vote, Comment (does it really collaborate if all it does is send the user to that page?) |
| Operations: | displayUserActions |

| | |
|---|---|
| Component Name: | UserLogin |
| Design Approach: | Functional |
| Parent Component: | django.View |
| Component Type: | View |
| Collaborators: | User, UserActions |
| Operations: | displayUserLogin |

| | |
|---|---|
| Component Name: | DisplayApplicants |
| Design Approach: | Functional |
| Parent Component: | django.View |

| | |
|---|---|
| Component Type: | View |
| Collaborators: | Applicant |
| Operations: | displayApplicants |

| | |
|---|---|
| Component Name: | Comment |
| Design Approach: | Object-oriented |
| Parent Component: | django.Model |
| Component Type: | Model |
| Collaborators: | |
| Operations: | setComment, setAuthor, getComment, getAuthor |

| | |
|---|---|
| Component Name: | GetGwaapDump |
| Design Approach: | Functional |
| Parent Component: | django.View |
| Component Type: | View |
| Collaborators: | Applicant, Application |
| Operations: | |

| | |
|---|---|
| Component Name: | LetterOfRec |
| Design Approach: | Object-oriented |
| Parent Component: | django.Model |
| Component Type: | Model |
| Collaborators: | |
| Operations: | recordResponses, getResponses |

| | |
|---|---|
| Component Name: | Vote |
| Design Approach: | Object-oriented |
| Parent Component: | django.Model |
| Component Type: | Model |
| Collaborators: | |
| Operations: | setVote, getVote, getAuthor, setAuthor |

| | |
|---|---|
| Component Name: | WriteComment |
| Design Approach: | Functional |
| Parent Component: | django.View |
| Component Type: | View |
| Collaborators: | Comment, Application, User |
| Operations: | displayCommentForm |

| | |
|---|---|
| Component Name: | CastVote |
| Design Approach: | Functional |
| Parent Component: | django.View |
| Component Type: | View |
| Collaborators: | Vote, Application, User |
| Operations: | displayVoteForm |

| | |
|---|---|
| Component Name: | NameReference |
| Design Approach: | Functional |
| Parent Component: | django.View |
| Attributes (optional): | |
| Component Type: | View |
| Collaborators: | Reference, UserProfile |
| Operations: | nameReference |

| | |
|---|---|
| Component Name: | Reference |
| Design Approach: | Object-oriented |
| Parent Component: | django.Model |
| Attributes (optional): | emailAddress, hasResponded, [response fields] |
| Component Type: | Model |
| Collaborators: | NameReferences, UserProfile, CompleteReference |
| Operations: | |

| | |
|---|---|
| Component Name: | CompleteReference |

| | |
|---|---|
| Design Approach: | Functional |
| Parent Component: | django.View |
| Attributes (optional): | |
| Component Type: | View |
| Collaborators: | UserProfile, CompleteReference |
| Operations: | completeReference |

| | |
|---|---|
| Component Name: | MakeComment |
| Design Approach: | Functional |
| Parent Component: | django.View |
| Attributes (optional): | |
| Component Type: | View |
| Collaborators: | User, Application |
| Operations: | makeComment |

| | |
|---|---|
| Component Name: | CastVote |
| Design Approach: | Functional |
| Parent Component: | django.View |
| Attributes (optional): | |
| Component Type: | View |
| Collaborators: | User, Application |
| Operations: | castVote |

## Scenarios

| | |
|---|---|
| **Feature** | User logs in to check an Application |
| **Spec Type:** | User |

| Tuple # | Type | Actor | Description | Example |
|---|---|---|---|---|
| 1 | Event | User | Visits website | User types "admin.gwaap.edu" into browser |
| 2 | Response | Blackbox | System responds with a login page | Page with title, username and password fields, and a Submit button |
| 3 | Event | User | User enters username, password, clicks "Submit" | User: admin, password: pass |
| 4 | Response | Blackbox | System authenticates user properly and displays "User Actions" page | List of all actions available to the user, including "View Applications" |
| 5 | Event | User | User chooses "View Application" | Click "View application" option |
| 6 | Response | Blackbox | System presents list of all applicants currently in the system to represent their current applications | List of applicants, including "Joe Smith" |
| 7 | Event | User | User clicks on Joe Smith's application | <click> |
| 8 | Response | Blackbox | System presents Joe Smith's application on the screen | Application data: name, address, phone, GRE score, etc. |
| 9 | | | | |
| 10 | | | | |
| 11 | | | | |
| 12 | | | | |
| 13 | | | | |
| 14 | | | | |
| 15 | | | | |

|  |  |  |  |  |
|---|---|---|---|---|
| 16 |  |  |  |  |
| 17 |  |  |  |  |
| 18 |  |  |  |  |
| 19 |  |  |  |  |
| 20 |  |  |  |  |
| 21 |  |  |  |  |
| 22 |  |  |  |  |
| 23 |  |  |  |  |
| 24 |  |  |  |  |
| 25 |  |  |  |  |

**Feature:** Applicant logs in to check their own application

**Spec Type:** User

| **Tuple #** | **Type** | **Actor** | **Description** | **Example** |
|---|---|---|---|---|
| 1 | Event | User | Applicant decides to check application and visits website | Applicant types "gwaap.edu" into browser |
| 2 | Response | Blackbox | System presents login page | Page w/ title, username/password fields, submit button |
| 3 | Event | User | Applicant enters valid login information, clicks submit | User: applicant1, password: pass |
| 4 | Response | Blackbox | System authenticates user and presents list of user options | System displays ApplicantActions page |
| 5 | Event | User | User chooses "View Application" | <click> |
| 6 | Response | Blackbox | System displays user's application details | Page w/ application info |

| | | | | |
|---|---|---|---|---|
| 7 | | | | |
| 8 | | | | |
| 9 | | | | |
| 10 | | | | |
| 11 | | | | |
| 12 | | | | |
| 13 | | | | |
| 14 | | | | |
| 15 | | | | |
| 16 | | | | |
| 17 | | | | |
| 18 | | | | |
| 19 | | | | |
| 20 | | | | |
| 21 | | | | |
| 22 | | | | |
| 23 | | | | |

| | | | | |
|---|---|---|---|---|
| 24 | | | | |
| 25 | | | | |

**Feature:** System prevents unauthorized access

**Spec Type:** User

| Tuple # | Type | Actor | Description | Test Case |
|---|---|---|---|---|
| 1 | Event | User | User wants to log in to another applicant's profile | "gwaap.edu" |
| 2 | Response | Blackbox | System provides login page | Title, username/password, submit |
| 3 | Event | User | User types incorrect data and submits it | Username: applicant1, password: badguess |
| 4 | Response | Blackbox | System returns to login page with a message "incorrect login info" | System attempts to login via ApplicantActions, is rejected, redirects to Login with failure message added to message queue |
| 5 | | | | |
| 6 | | | | |
| 7 | | | | |
| 8 | | | | |
| 9 | | | | |
| 10 | | | | |
| 11 | | | | |
| 12 | | | | |
| 13 | | | | |
| 14 | | | | |

## Change Log

| Number | Date | Type | Inject Activity | Inject Iteration | Remove Activity | Remove Iteration | Fix Time | Fix Reference | Description |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 11/6/2011 | Product interfa | Analysis | NA | Planning | 1 | 10 | | Added "DisplayApplicants" component so that Users can view all applicants in the system |
| 2 | 11/24/2011 | Product interfa | Architecture | NA | Construction | 1 | 15 | | Changed base class of "User" and "Applicant" to django.contrib.auth.models.User |
| 3 | 11/28/2011 | Product interfa | Architecture | NA | Construction | 1 | 45 | 2 | Add ApplicantProfile module to attach Application (model) to Applicant (user) |
| 4 | 2/26/2012 | Product logic | Architecture | NA | Planning | 3 | 180 | 2 | Django doesn't like multiple subclasses of User in the same project. |
| 5 | 3/3/2012 | Product logic | Architecture | NA | Construction | 3 | 60 | | Using Django signals to hook up GWAAP user types with their permissions and attribute models |
| 6 | 3/19/2012 | Product logic | Planning | 4 | Construction | 5 | 15 | | Incorrect relationship between Users and Comment/Vote objects |
| 7 | | | | | | | | | |
| 8 | | | | | | | | | |
| 9 | | | | | | | | | |
| 10 | | | | | | | | | |
| 11 | | | | | | | | | |
| 12 | | | | | | | | | |
| 13 | | | | | | | | | |
| 14 | | | | | | | | | |
| 15 | | | | | | | | | |
| 16 | | | | | | | | | |
| 17 | | | | | | | | | |
| 18 | | | | | | | | | |
| 19 | | | | | | | | | |
| 20 | | | | | | | | | |
| 21 | | | | | | | | | |
| 22 | | | | | | | | | |
| 23 | | | | | | | | | |
| 24 | | | | | | | | | |
| 25 | | | | | | | | | |
| 26 | | | | | | | | | |
| 27 | | | | | | | | | |
| 28 | | | | | | | | | |
| 29 | | | | | | | | | |
| 30 | | | | | | | | | |
| 31 | | | | | | | | | |
| 32 | | | | | | | | | |
| 33 | | | | | | | | | |
| 34 | | | | | | | | | |
| 35 | | | | | | | | | |
| 36 | | | | | | | | | |
| 37 | | | | | | | | | |
| 38 | | | | | | | | | |
| 39 | | | | | | | | | |
| 40 | | | | | | | | | |
| 41 | | | | | | | | | |
| 42 | | | | | | | | | |
| 43 | | | | | | | | | |
| 44 | | | | | | | | | |
| 45 | | | | | | | | | |
| 46 | | | | | | | | | |
| 47 | | | | | | | | | |
| 48 | | | | | | | | | |
| 49 | | | | | | | | | |
| 50 | | | | | | | | | |
| 51 | | | | | | | | | |
| 52 | | | | | | | | | |
| 53 | | | | | | | | | |
| 54 | | | | | | | | | |
| 55 | | | | | | | | | |
| 56 | | | | | | | | | |
| 57 | | | | | | | | | |
| 58 | | | | | | | | | |
| 59 | | | | | | | | | |
| 60 | | | | | | | | | |
| 61 | | | | | | | | | |
| 62 | | | | | | | | | |
| 63 | | | | | | | | | |
| 64 | | | | | | | | | |