# Handoff Document

## Getting Started

Welcome to the wonderful world of PharmGenius. If you’re reading this you might be concerned that you are about to be working with legacy code you know nothing about. That’s absolutely valid. We’ve written this document to help you along the way.

This program has been written in a way that abstracts a lot of details away to the point where you don’t necessarily need to know how all of the code works. We have tried to comment every utility method to describe what its arguments are and what it returns. That does not mean that you shouldn’t learn how everything works, but that you don’t necessarily have to.

The code you will be working with is primarily in Python and Javascript. In addition to that, there is quite a bit of HTML and some CSS to become familiar with but this should be trivial.

<https://cloud.google.com/appengine/docs/python/>

This link is a good resource to help you learn how to work with Google App Engine’s Python SDK. I recommend running through the guestbook example before working with this codebase to make sure that you have a good background in the necessary syntax.   
  
<https://cloud.google.com/appengine/docs/python/gettingstartedpython27/creating-guestbook>  
  
After you are comfortable with Google App Engine and Python, you can begin to dive into code.

## The Code

There are three primary folders you should become familiar with. The root folder contains the important Python, /static/ contains some resources, css, js, and audio files, and /templates/ contains the markdown.

The first part of code to look at is probably the data structures. These are in the models.py file in the root folder. The User, Answer, Question, Score, and Category models are at the top of the file. These are the data structures that make up this program. Under these are helper functions that create, update, and delete these fields from Google’s NBD, which is the database that saves all of our data. If you can familiarize yourself with these data structures, their helper functions, and how those helper functions work you are well on your way to understanding the program.

It is important to access everything in the database by key, rather than using a query. We had issues with consistency when using queries, and had to sleep() on many page refreshes so the database would have time to process changes. User’s key is its id, Category’s key is the category string, and the Question key is passed around directly. It takes a little effort to understand how the NDB keys work, but it’s the only way to ensure strong consistency. As of right now, Answers and Scores never need to be referenced individually, so they can be retrieved with a query.

Category

Question

User

Answer

Score

After reading over models.py you can move to main.py. This is the middle layer of the application. From here we gather information from queries to the database and send that data along to the frontend in the form of json, passed in through page\_params. As an example I’ll explain how our main page is called.

class MainPageHandler(webapp2.RequestHandler):  
 def get(*self*):  
 id = get\_user\_id()  
 is\_admin = 0  
 if users.is\_current\_user\_admin():  
 is\_admin = 1  
 logging.warning(models.getCategoryList())  
 newList = models.getCategoryList()  
 page\_params = {  
 'catList': newList,  
 'user\_email': get\_user\_email(),  
 'login\_url': users.create\_login\_url('/firstLogin'),  
 'logout\_url': users.create\_logout\_url('/'),  
 'user\_id': id,  
 'admin' : is\_admin  
 }  
 render\_template(self, 'index.html', page\_params)

The first line is the name of the class. The name of it, MainPageHandler, has a mapping at the bottom of main.py that looks like this: ('/', MainPageHandler),

This means that when the user visits the root of our site they are redirected to this python class. The class first uses the function get\_user\_id() to determine if the user is an administrator or not. The admin homepage looks different than the normal user homepage so we need this distinction. Next we run the query models.getCategoryList() and store it in the variable called newList. We create a variable named page\_params, load it with the category list and some other variables, and then render index.html (which lives in the templates folder).

From there we will look at index.html. First we’ll run through the javascript at the top of the page.

$(document).ready(function() {  
 $('#on-play').click(function(){  
 $('#description').animate({  
 left:"-150%"} , "slow");  
 $('#dropdown').animate({left:"0%"} , "slow");  
 document.getElementById("dropdown").style.display = "block";  
 });  
 $('#on-back').click(function(){  
 $('#description').animate({  
 left:"0%"} , "slow");  
 $('#dropdown').animate({left:"15%"} , "slow");  
 document.getElementById("dropdown").style.display = "none";  
 });  
 var curr = {{catList|safe}};  
 var dropdown = document.getElementById("category");  
 var option = document.createElement("option");  
 for (x = 0; x < curr.length; x++){  
 var option = document.createElement("option");  
 option.innerHTML = curr[x].category;  
 dropdown.appendChild(option);  
 }  
 });

This is a jquery/javascript function that runs after the page is finished loading. First we specify functions that give the play button its functionality. Upon first click we shift the play button off the screen and bring the play game dropdown onto it. Clicking the back button reverses the process. This next part is critical.

var curr = {{catList|safe}};

catList is an argument in the page\_params that contains a json representation of the list of categories returned from models.getCategoryList(). The double bracket notation is how we access those variables. The ‘catList|safe’ is going to allow this variable to act as a readable array of categories. This piece of code is critical.

Next we iterate through that variable and add a value to the play button’s category dropdown for every category in the list. This is standard js and is pretty readable but if you miss any of these pieces that can cause a bit of a headache.

So what happens when you hit run a quiz?

### **Quiz Functionality**

The form action on the Start Quiz button as follows: <form action="/takeQuiz">.

We go into the python and look at the bottom of main.py, which looks like this

mappings = [  
 ('/', MainPageHandler),  
 ('/profile', ProfileHandler),  
 ('/submitNew', SubmitPageHandler),  
 ...  
 ('/takeQuiz', categoryQuiz),  
 ...  
]

In this mappings we redirect ‘/takeQuiz’ to the python function ‘categoryQuiz.’

Because the form action’s default is get, at the top of categoryQuiz we have ‘def get(self):’ which handles all GET calls to it. We get the form data with:

category = self.request.get('category')  
number = self.request.get('number')

And run the following query

questions = models.getQuestionsCat(category,int(number))

The next code is pretty self explanatory:

if questions is None:  
 num = 0  
 jList = []  
 else:  
 num = len(questions)  
 qList = []  
 for q in questions:  
 temp = q.to\_dict(*exclude*=['category','creator','accepted','up\_voters','down\_voters','create\_date'])  
 qList.append(temp)  
 jList = json.dumps(qList, *default*=obj\_dict)

We have handling for if the query returned no questions, and if it does have questions we proceed. First we make an empty array called qList and iterate through the query. We strip some unnecessary question data from each question object and then append it to qList. Finally we make sure to convert qList into a json object, named jList.

We send jList as a page\_param and render answerQuestionsCat.html, which runs the quiz.

Hopefully the code above has been relatively easy to understand and interpret but answerQuestionsCat is anything but. This HTML file has close to 400 lines of javascript and over 100 lines of markdown that make it anything but readable. You should not have to change any of it but I will try to guide you into understanding it.

The css at the beginning of the file is used on the quizResults page that runs at the end of a quiz. The js at the top of the page does a number of things. It contains our timer, our results page functionality, loads each question in the quiz, and scores answers. Some of these are better commented than others.

That markdown has the individual question page, an answerResults modal, a report modal, and a hidden quizResults table that is shown when the quiz is complete. What follow is all javascript.

We have a vote function that provides functionality for upvoting and downvoting questions. This should be easy to understand.

Next is a 150 line javascript function called submit that is run whenever the user answers a question. First it gathers the variables it is going to need and calculates how much score a user would get for each question if they get it correct. Next we check if the answer is correct or not.

After this we populate the result into quizResults and prepare the answerResult modal to be loaded. This loads the answers into the modal, highlights the user answer, makes the incorrect red and the correct green, and does play a sound that is determined by the answer’s correctness. If you can figure out this function the next few will be trivial.

We have runNextQuestion which first resets the modal and then runs the next question.

sendReport handles the report modal and sends an ajax request to the python backend.

sendAnswer sends the user answer to the python backend in an ajax request.

These ajax requests were tricky so I’ll try to explain them as well. Let’s look at the one in sendAnswer.

$.ajax({  
 type: "POST",  
 url: "/answerSingle",  
 contentType: "application/json; charset=utf-8",  
 data: JSON.stringify({"userID": userID, "qKey": questKey, "userSelection": userAnswer, "score": score})  
 });

$.ajax() is jquery’s lovely abstraction of an ajax call. The URL is sent to /answerSingle, which is mapped to answerSingle in python.

The inside of that looks like this:

data = json.loads(self.request.body)  
  
 question = models.getQuestionFromURL(data['qKey'])  
 models.createAnswer(data['userID'],question.key,str(data['userSelection']), int(data['score']))

I put this here to demonstrate both sides of an ajax call. In JS data looks like this:

data: JSON.stringify({"userID": userID, "qKey": questKey, "userSelection": userAnswer, "score": score})

In python interpreting looks like this:

models.createAnswer(data['userID'],question.key,str(data['userSelection']), int(data['score']))

This syntax looks intuitive and is easy to use once you have it correct but getting this to work the very first time was *awful.* Thankfully we did that and you don’t have to.

### **Admin vs Standard User**

Within App Engine you are able to give users different privilege levels, these can be changed by going into the permissions page of the App Engine Dashboard. The Admin user has access to more pages than the regular user. As explained above there is a helper function that can be used to get the permission level and that is passed to the page. Core functionality is given to admins such as the ability to remove and add categories for which quizzes can be taken as well as the ability to vote, edit, and delete questions that have been submitted by the users. The navbars are where the options differ. There are three different html versions of the navbar, \_base.html, \_baseHome.html, and \_baseCreate.html. These three are needed for small differences within the pages they are being used on. \_base.html is the primary one used on most of the pages the other two are only used once. The \_baseHome.html is only used on the homepage and the difference is just within the style of the bar. \_baseCreate.html is used on the createProfile.html page to eliminate options when a new user is on the page. {% extends \_base.html %} is used to denote the navbar being displayed on the page.

## **Future Features**

* More permission levels system
  + Levels:
    - Highest: Full Admin Control like App Engine’s
    - Remove Ability to Deploy and Mess With Database in Dev Console
      * Maybe the highest within the database, whereas the highest will just be manually entered into App Engine
    - Remove Edit/Delete Ability in Question and Category
      * Basically only vote
    - Lowest: Normal User
  + Will probably need a page alongside this system to make these edits
  + Alongside this comes a new user model where we can also accept other, non-gmail email addresses.
    - Confirmation emails?
* Play game button elsewhere besides front page
  + Maybe animation on the navbar or another page
* Badge Reward System
* General Enhancement of Profile Page
  + Not very great to begin with
* Clickable Links in Leaderboard to Profile of User
* Tags on questions
  + Beyond just one category
* Multi Category Quiz
* Permanently Delete Categories
* Limit leaderboard size
  + Currently returns a list of all of the people who have a profile, even when score is 0