Artur Miranda Guimaraes

Carlos Rafael de Oliveira do Amaral Leitão

João Felipe Farias Mendes

Summer Project- MedJobFinder

This documentation include information about the summer project created by the study abroad students from the Brazilian program Science Without Borders at Goldsmiths in the summer of 2015. The Students were tutored by Dr. Lahcen Ouarbya.

London - 2015

**Chapter 1 - Introduction:**

This report is one of the requirements of the Summer Project subject for the Science Without Borders program. This program sponsored the authors of the document for the 2014-2015 academic year in Goldsmiths, University of London.

Artur Miranda Guimaraes: BSc Business Computing

Carlos Rafael de Oliveira do Amaral Leitao: BSc Computer Science

Joao Felipe Farias Mendes: BSc Business Computing

For the Summer Project, we were required to get in contact with professionals/tutors from the computing department to develop together a project that would apply the skills learned in the academic year at Goldsmiths, University of London. Therefore, we contacted **Dr. Lahcen Ouarbya**, and together we identified a gap for a possible project with a computing-based solution. This project is named MedJobFinder.

MedJobFinder is a web app that aims to connect potential employers and employees from the healthcare industry. We identified that there is currently no easy way for a doctor to find a job close to his location, so we created MedJobFinder to test that hypothesis and fill that gap in the market.

MedJobFinder uses the location of the potential employers to suggest them to employees and provides tools for the employees to easily identify jobs that attend their requirements. Based on information contained in a doctor’s profile, the website can suggest potential jobs near his address that are related to his expertize, or that are related to his desires for a job.

In addition, doctors can also visualize a map with all job offers inserted in the system for any city. For example, if a doctor lives in London, the default will be for him to visualize the jobs available in London.

For employers a.k.a. hospitals: They can create a job offer, which is based on their location and they can specify working hours, wage and job description. And also there is a list of doctors available for them which they can contact at any time via messages.

**Introduction - Implementation**

To implement the website, we used a diverse set of programming languages. We divide the website in two main sections: Front-end and Back-end. The Front-end is the part that display information to the user and collects user input, which is also characterized as the interface between the user and the back-end. To implement this part we used HTML5, CSS3, Javascript and JQuery.

The Back-End is the part that processes data and return that to the front-end so it can be displayed to the user. To implement this we used Python as the main programming language, due to the easiness and availability of external libraries. Python also has a framework web called Django, which is vital and actually is the core for our entire project. Django is a lightweight and standalone webserver made for quick web development. Django uses the MVC framework, which is an architectural pattern for implementing user interfaces and the “Don’t Repeat Yourself” (DRY) principle which enables the reuse of code and reduces development time. Other libraries used in the project were the Google Maps API in Javascript, as well as several sub-libraries from python. All these libraries are inside the requirements.txt part of the project.

In addition, for project management we used a modified version of Scrum, which is an interactive and incremental agile software management framework. To keep track of the development using Scrum, we chose Asana, a webapp which we can define and control tasks between multiple people. Asana also enabled us to keep all the development in schedule and to plan next requirements.

Figure 1 illustrates the Asana with the tasks:

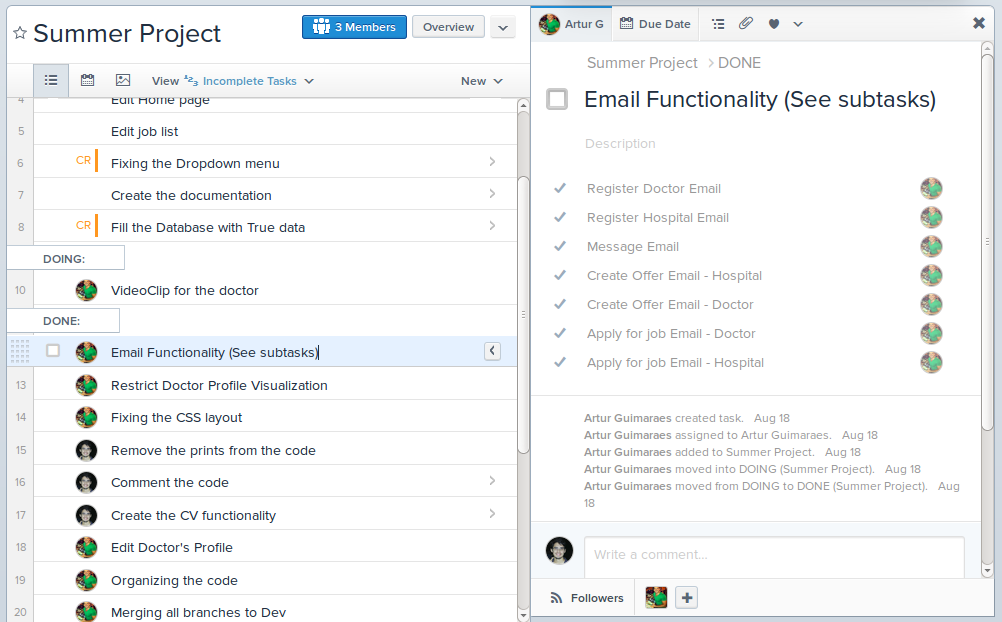


Figure 1 - Tasks in Asana

For code version change and issue tracking, we used Git, which is a version control system and bitbucket, that is an online repository for code. Figure 2 shows the development and commits (changes) made to the remote repository at multiple times.

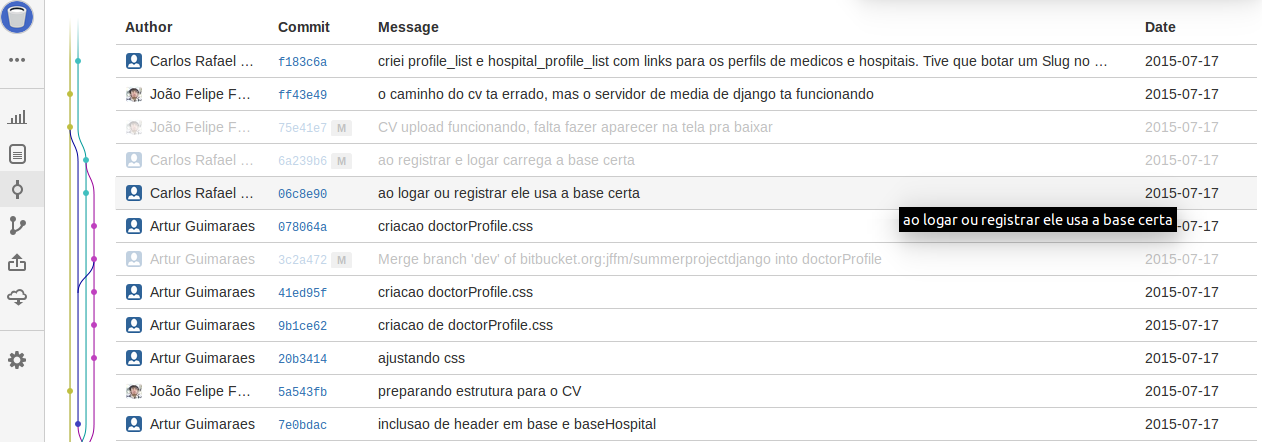


Figure 2 - multiple commits created in Bitbucket

**Why did we chose these technologies?**

HTML5: Mobile support and Audio and Video support;

CSS3: Diverse set of stylization for design and user experience purposes;

Javascript: Javascript is one of the most simple, versatile and effective languages used to extend functionality in websites. Uses range from on screen visual effects to processing and calculating data on web pages with ease. In addition, Javascript is easy to implement and relatively fast to the end user.

Python: Very clean and straightforward programming language, with plenty of libraries available to the programmers. Also, with Django, a great support for developing web apps.

**Explaining Django’s MVC Structure:**

Django uses a Model-View-Template (MVT) structure, similar to the MVC model.

The central component of MVT, the *model*, captures the behavior of the application in terms of its problem domain, independent of the user interface. The model directly manages the data, logic and rules of the application. A *view* can be any output representation of information, such as a chart or a diagram; multiple views of the same information are possible, such as a bar chart for management and a tabular view for accountants (controls what a user sees). The third part, the *template*, accepts input and converts it to commands for the model or view and also controls how a user sees. In addition, In Django the controller is the framework itself, with the middleware and the URL parsing that redirect to the views.

### **Interactions**

In addition to dividing the application into three kinds of components, the model–view–template design defines the interactions between them.

* A **template** can send commands to the model to update the model's state (e.g., editing a document). It can also send commands to its associated view to change the view's presentation of the model (e.g., by scrolling through a document).
* A **model** stores data that is retrieved according to commands from the controller and displayed in the view.
* A **view** generates an output presentation to the user based on changes in model. In some cases, data and logic can also be made on a view, which is the case on this project.

**Chapter 2 - Design Section:**

We have designed a database system to support all the data used in the website. The database is implemented in SQLite 3, a light and fast type of database suitable for quick development. This database is organized in 8 tables.

The following diagram (Figure 3) illustrates the database:

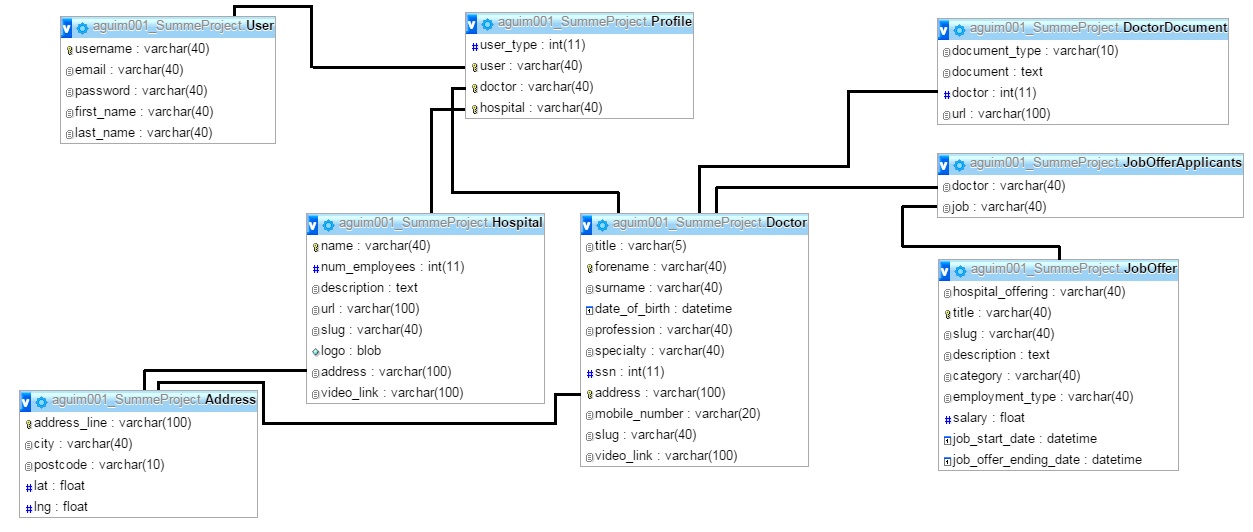


Figure 3 - Database Tables

This database is generated by classes in a file called models.py in the project root, this models.py defines the models of the MVT framework explained on Section 1. Django is a very interesting framework for building databases, for example, the following code snippet is a description of the table Doctor in Python.

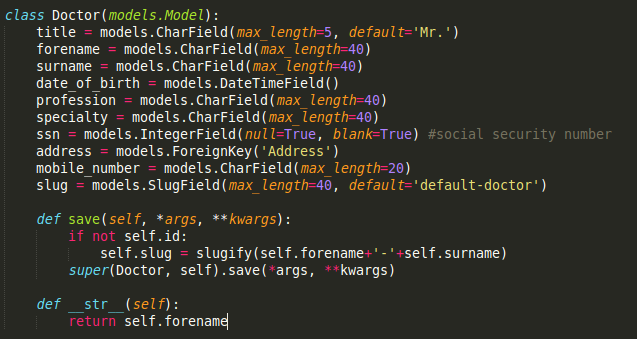


Figure 4 - Doctor Model

Django also makes it very simple to create and modify a database. For example, if we add a new attribute “field= models.CharField(max\_length=40)” to the model Doctor (or Class Doctor), we don’t need to generate any SQL to make the changes in the database, Django already does that for us with a few commands, these commands are illustrated by the following picture:

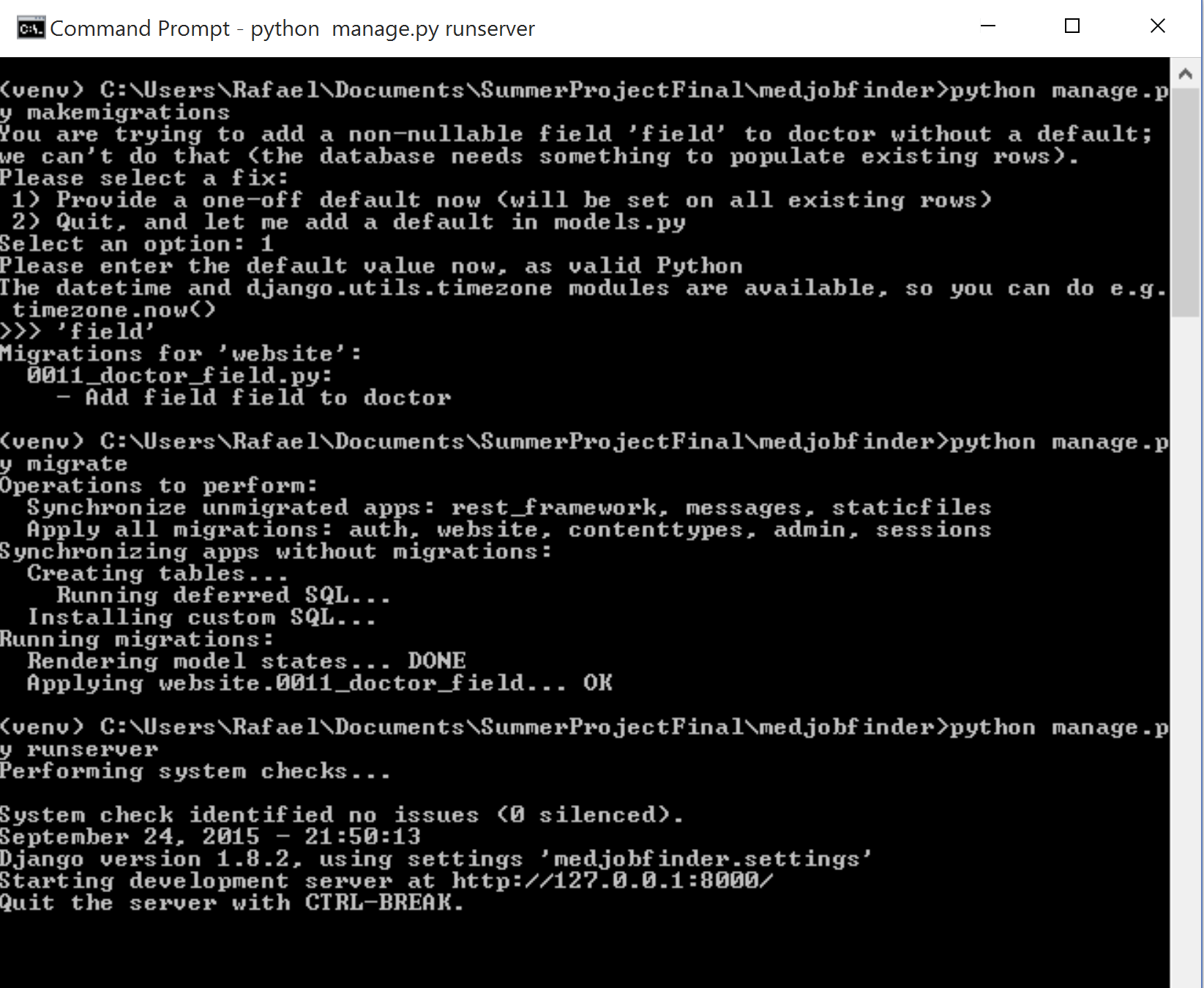


Figure 5 - Changing the database

After adding the field in the model, we just execute a “manage.py makemigrations” command, which will create a python script that generate a SQL code and can alter the table when we need to. To modify the database, we execute “manage.py migrate” and all migrations will be applied to the database. If this step was successfully completed, the server can run normally.

Here is a brief explanation of the tables and its attributes:

* User
  + id (Integer)
  + username (Varchar)
  + email (Varchar)
  + password (Varchar): Uses PBKDF2 algorithm with a SHA256 hash
  + first\_name (Varchar)
  + last\_name (Varchar)
* Hospital
  + id (Integer)
  + name (Varchar)
  + num\_employees (Integer)
  + description (Text)
  + url (Varchar)
  + slug (Varchar): Unique identifier to be used in links
  + logo (Image)
  + address (Varchar): Foreign Key referencing the Address table
  + video\_link (Varchar)



Figure 6 - Hospital Model

* Doctor
  + id (Integer)
  + title (Varchar)
  + forename (Varchar)
  + surname (Varchar)
  + date\_of\_birth (Datetime)
  + profession (Varchar)
  + speciality (Varchar)
  + ssn (Integer)
  + address (Varchar): Foreign Key referencing the Address table
  + mobile\_number (Varchar)
  + slug (Varchar): Unique identifier to be used in links
  + video\_link (Varchar)

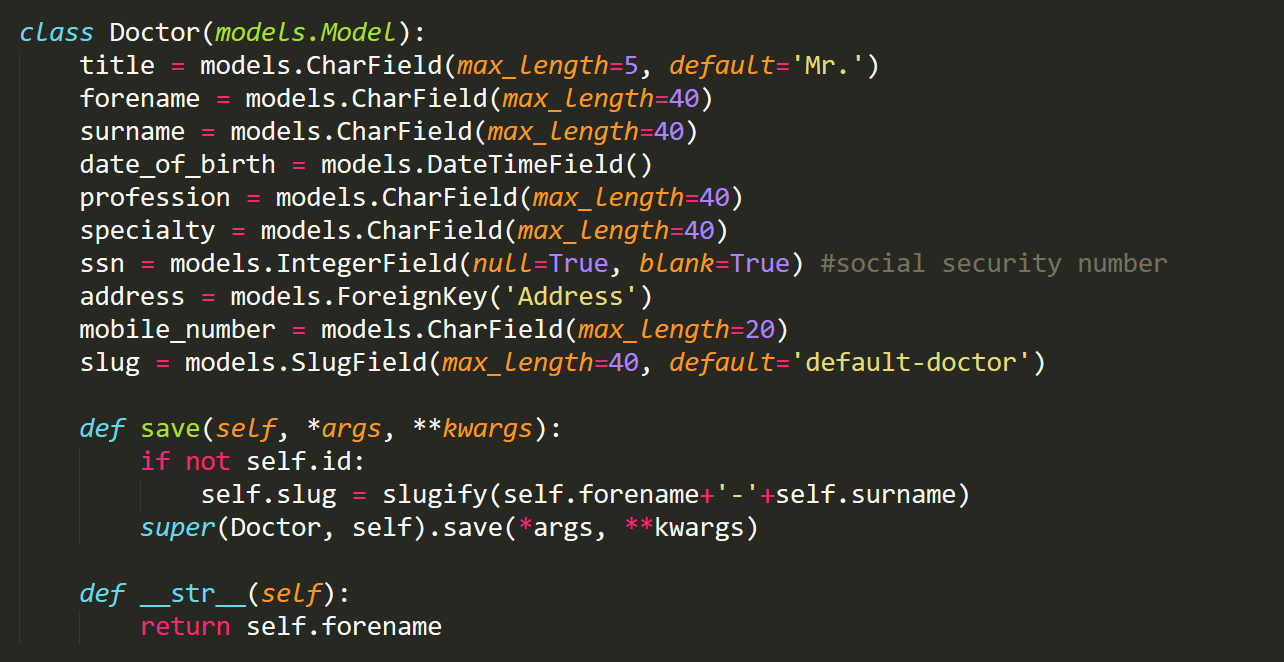


Figure 7 - Doctor Model

* Profile
  + id (Integer)
  + user\_type (Integer)
  + doctor (Varchar): Foreign Key referencing the Doctor table
  + hospital (Varchar): Foreign Key referencing the Hospital table

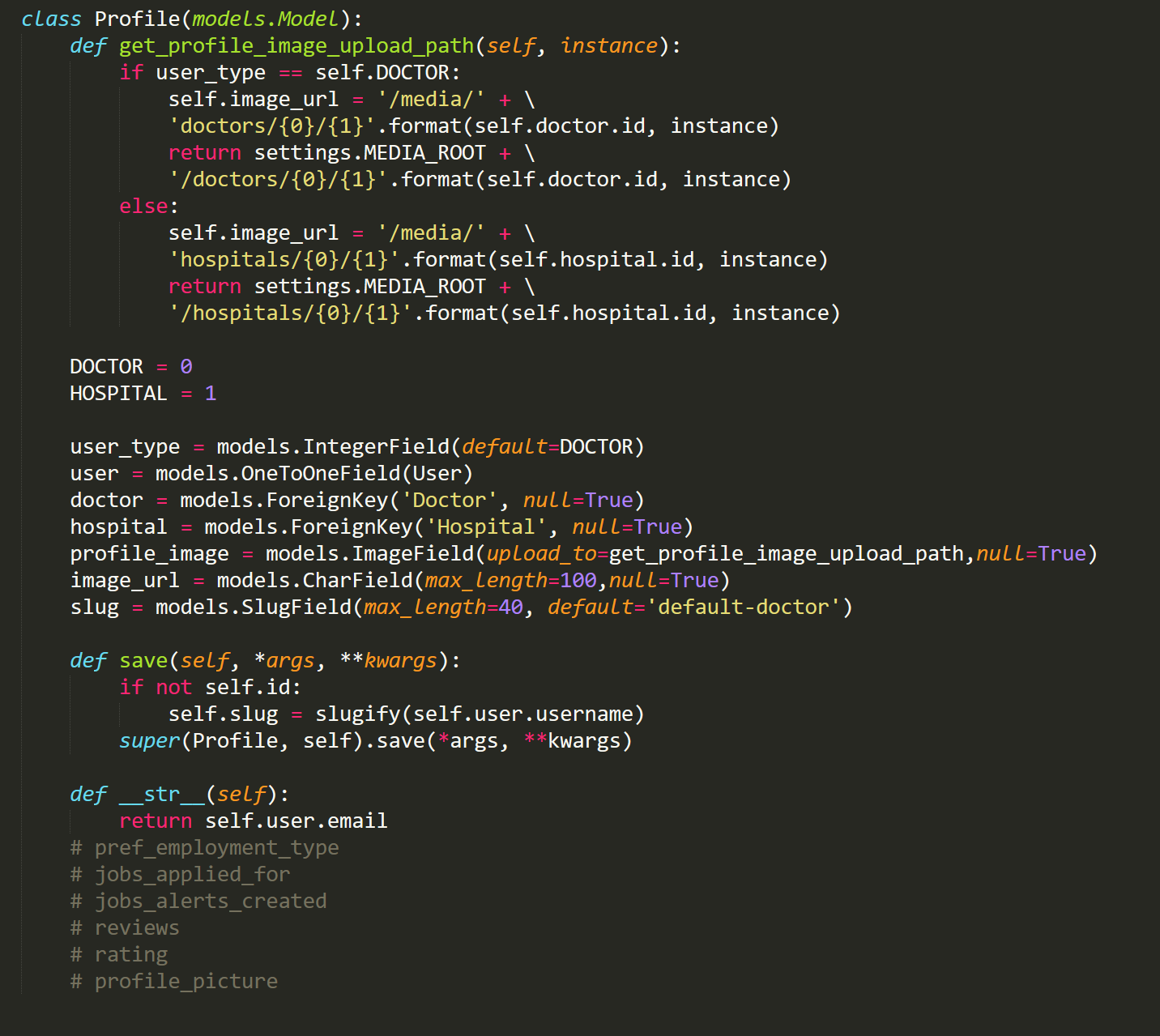


Figure 8 - Profile Model

* Address
  + id (Integer)
  + city(Varchar)
  + postcode (Varchar)
  + lat (Float)
  + lgn (Float)

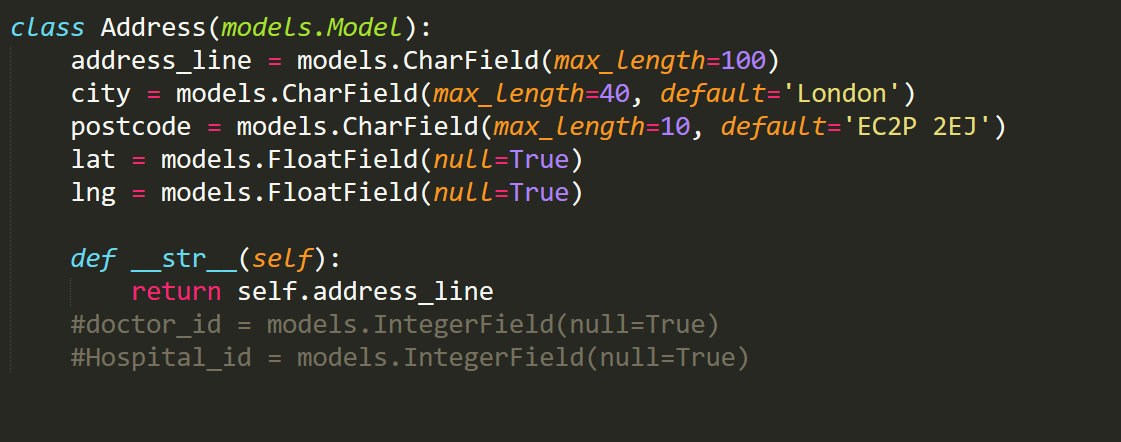


Figure 9 - Address Model

* DoctorDocument
  + id (Integer)
  + document\_type (Integer)
  + document (Text)
  + doctor (Integer): Foreign Key referencing the Doctor table
  + url (Varchar)

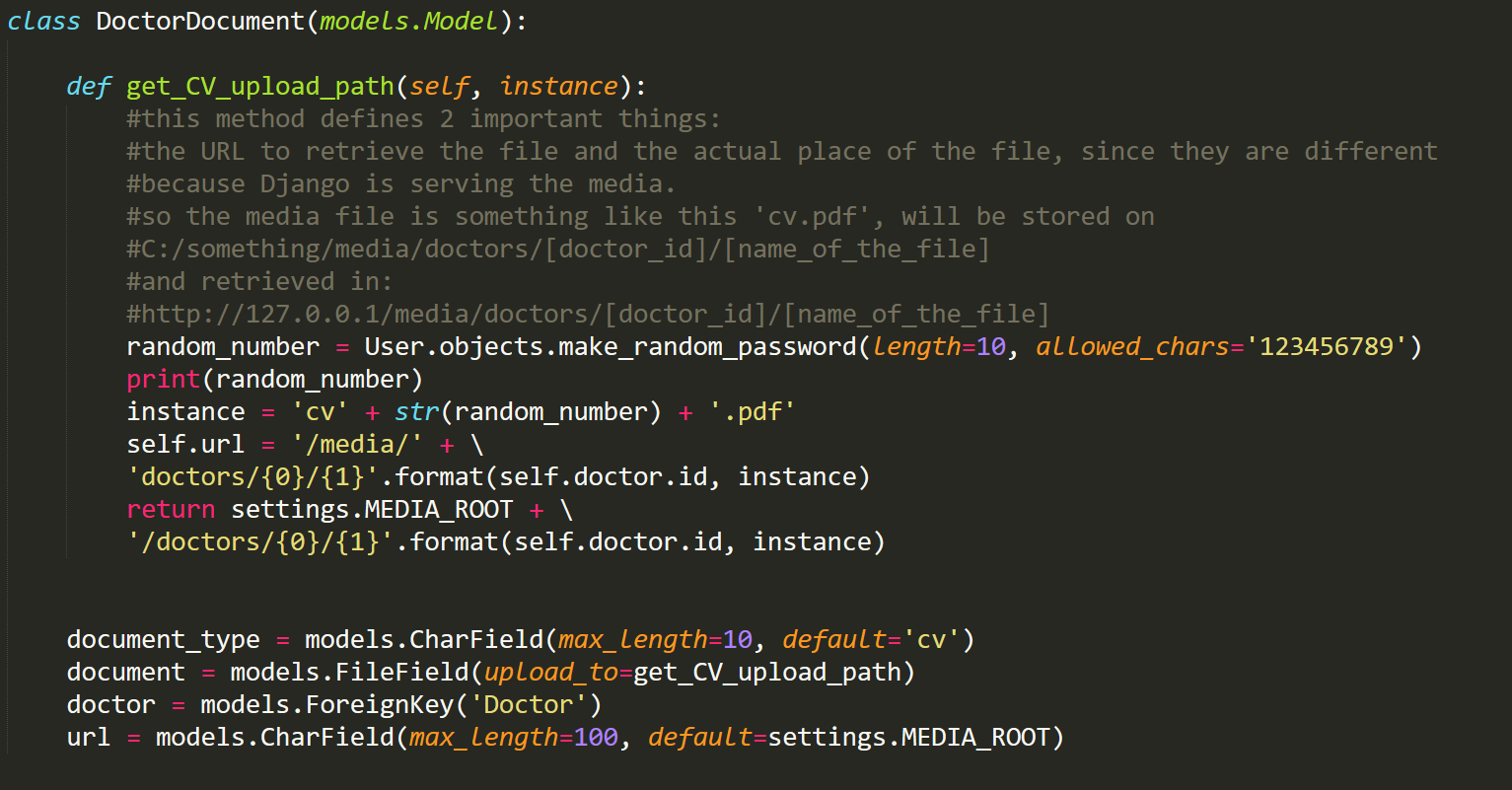


Figure 10 - DoctorDocument Model

* JobOffer
  + id (Integer)
  + hospital\_offering (Varchar): Foreign Key referencing the Hospital table
  + title (Varchar)
  + slug (Varchar): Unique identifier to be used in links
  + description (Text)
  + category (Varchar)
  + employment\_type (Varchar)
  + salary (Float)
  + job\_start\_date (Datetime): Time the user created the Job Offer
  + job\_end\_date (Datetime): Time the Job Offer will expire

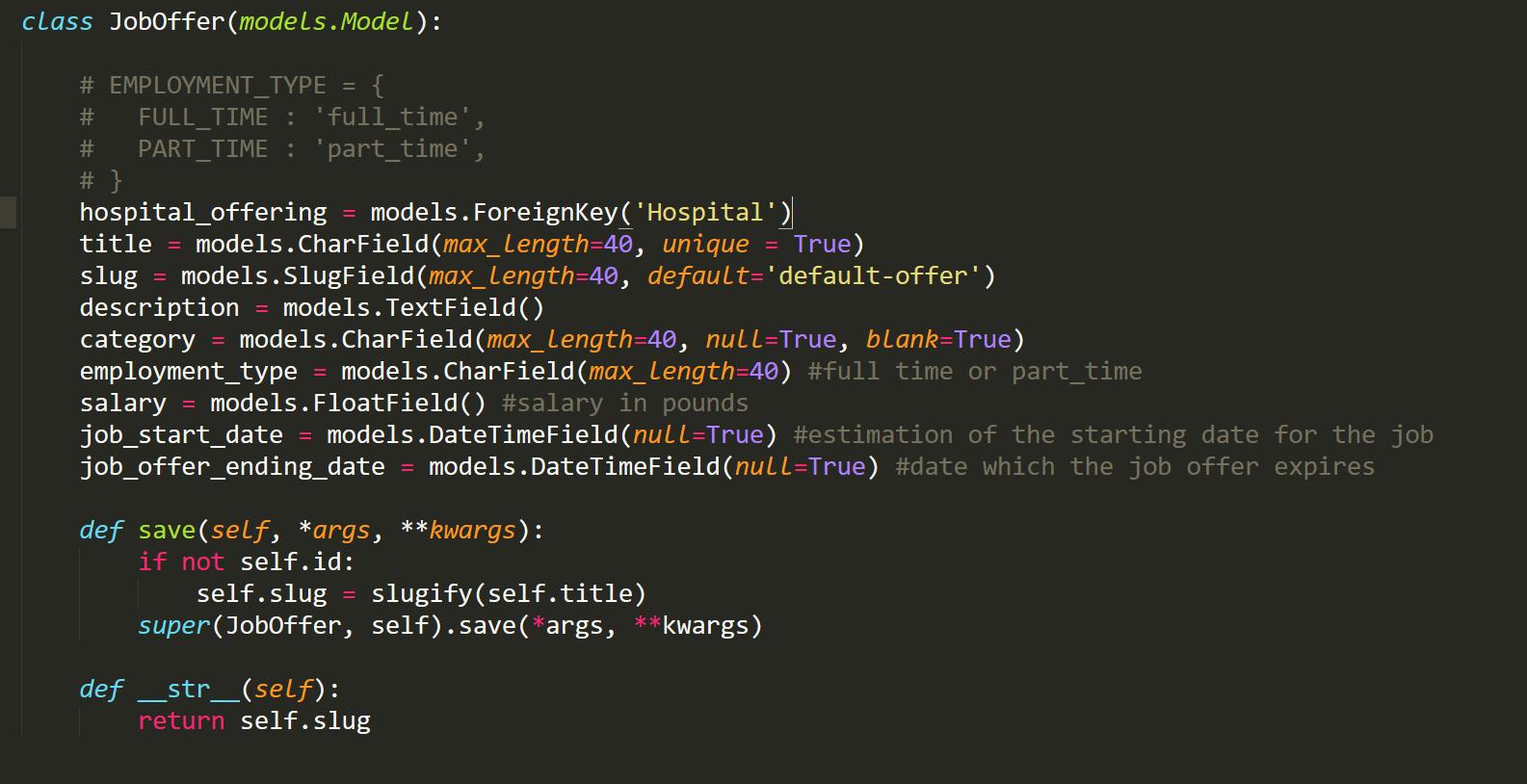


Figure 11 - JobOffer Model

* JobOfferApplicants
  + id (Integer)
  + doctor (Varchar): Foreign Key referencing the Doctor table
  + job (Varchar): Foreign Key referencing the JobOffer table



Figure 12 - JobOfferAplicants Model

**Chapter 3 - Implementation:**

For the implementation chapter, we explain the main pieces of code. Both Front-end and Back-end code snippets are included in this section.

**Code snippet for Map handler (front-end):**

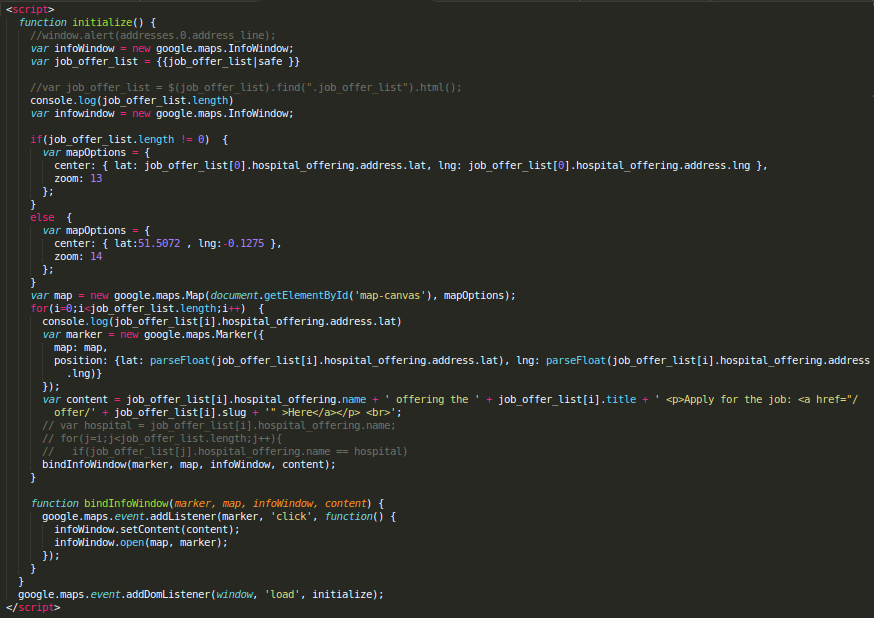


Figure 13 - Map Handler functions

In this snippet of code we created the Location functionality. This snippet is called in every page of the website, since we are calling it in the header.html file, but we only load it when we are in the page displaying the map.

This is a JavaScript code and we only have two functions, initialize() and bindInfoWindow(). In the first one we initialize the map and all its features. We start creating an infoWindow, which displays content (usually text or images) in a popup window above the map, at a given location, and a variable that receives the job offers list that will later be displayed in the infoWindows. We then do an If-Else statement to verify whether we have the list with the job offers or not and create a variable that later we will use to define how the map will be displayed, its center and its zoom. If we have received the job offers list we define the center of the map as the address of the hospital offering the first job offer of the list, otherwise we center the map in a location in London.

After this we create a map attaching it to the ‘map-canvas’ element and with the zoom and center as defined by mapOptions variable we have just created. We then do a For loop through the elements of the job\_offer\_list creating a marker for each element. A marker just identifies a location on a map and usually just need two attributes to be constructed: map and position. We define the map as the variable we had created and the location based on the latitude and longitude of the address of the hospital offering the offer. Besides the marker, we also create a message to be displayed with the marker showing the name of the hospital, the job offer and the link for the user to apply for it.

To display the marker with the message we have to bind them to a infoWindow, we do so calling a function. In the bindInfoWindow we add an event listener to the marker, we set the content to be the message we had created before so when clicking the marker the map will display the content.

**Code snippet for Map handler (back-end):**

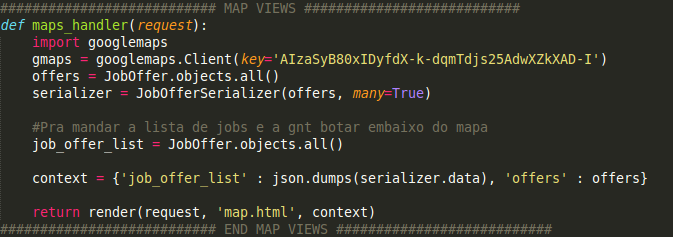


Figure 14 - views.maps\_handler

This function handles the google map client for the back-end. First we are initializing the client using our API-key, a requirement for the utilization of the google maps API. Then we get all the job offers inside the database using the JobOffer.objects.all(), and prepare a serializer to wrap the offers data for the front-end. Also, we pass the job offer list for the context in a json format (so the front-end can desencapsulate that data) and return the request for the proper template (map.html) with the context (python variables that can be accessed in the HTML/Javascript).

**Code snippet for views.register\_doctor:**

We use the register\_doctor function when the doctor user is signing up for our service and after completing the form of registration it sends all the data submitted:



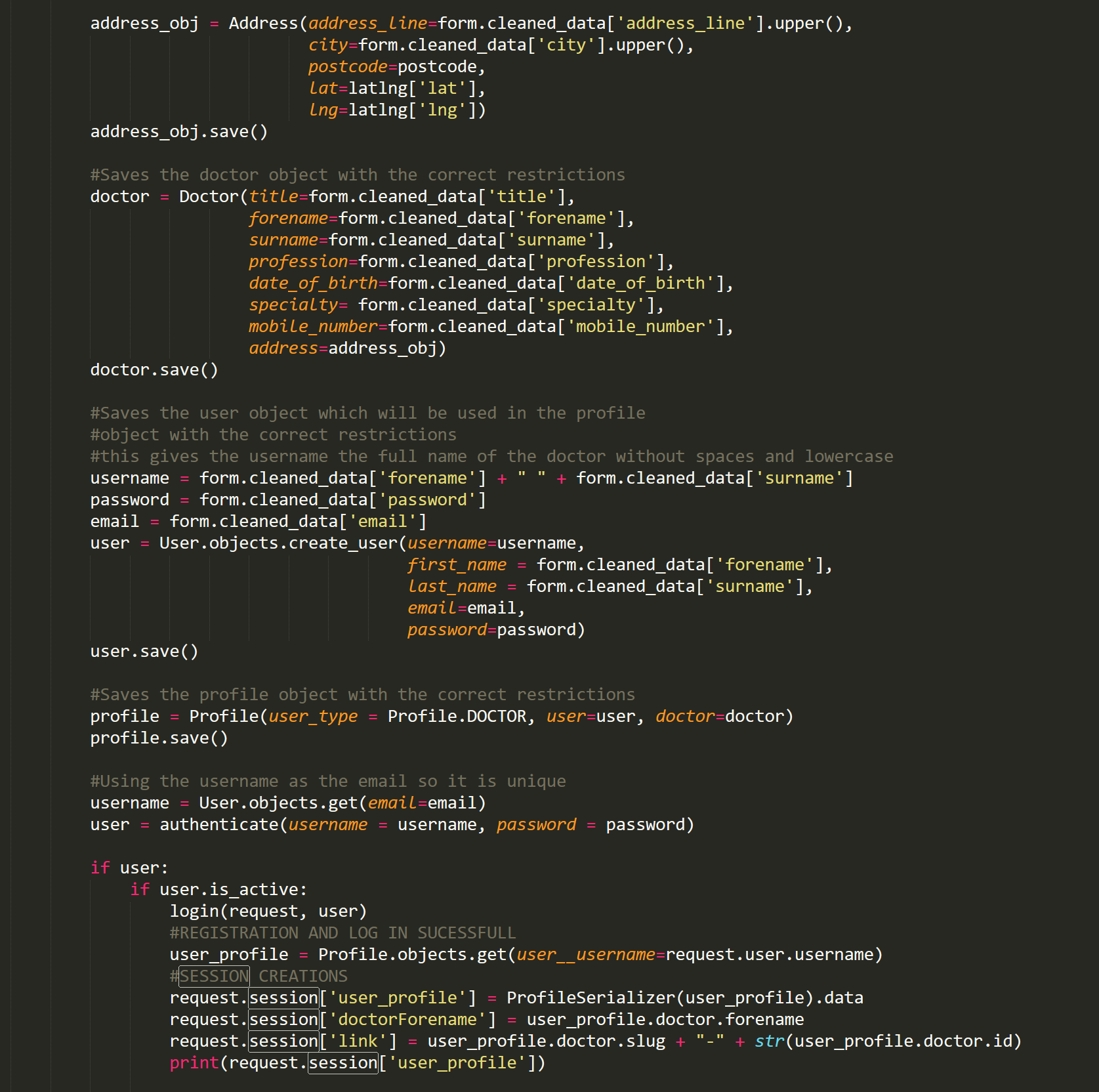




Figure 15 - views.register\_doctor

In Django when a page is requested, Django creates an [HttpRequest](https://docs.djangoproject.com/en/1.8/ref/request-response/#django.http.HttpRequest) object that contains metadata about the request. Then Django loads the appropriate view, passing the [HttpRequest](https://docs.djangoproject.com/en/1.8/ref/request-response/#django.http.HttpRequest) as the first argument to the view function. So in our function we receive the HttpRequest object and we first see whether its method is POST or not.

If the answer is no we will get in the else clause, which means that no data has been sent. Since that was no data received we create a variable ‘form’ that contains the RegisterDoctorForm, defined in the forms.py(file responsible for all the forms created).

If the method of the HttpRequest object is POST the user has sent some data after submitting the form so now we will handle this. First we define the variable ‘form’ as the RegisterDoctorForm the user sent in the POST method. If the form is not valid we display a message to the user. If the form is valid but the fields email and retypeEmail or password and retypePassword are not the same we display another message to the user saying that they do not match. If the form was valid we get the postcode submitted and try to get the Latitude and Longitude of this postcode, if it is not possible we set the location to London.

We then create a address, doctor, user and profile objects with the data sent by the form and save them. For the user we have the password and the username as the email and then we try to authenticate the user. If the user does not exist we return a HttpResponse with “Invalid Login”. If the user exist we check whether it is active or not. If the user is active we log in the user and create the variable user\_profile with the profile object associated with the user logged in. Now we create sessions( which are a framework that lets us store and retrieve arbitrary data on a per-site-visitor basis) that store information we will use later in other functions while the user is logged in. To finish we create a success message to be displayed in the page after registration and the variables subject and message. We then try to send an email to the user calling the send\_mail\_message function from Django.

The create\_offer function is just called when the user is logged in as an Hospital and submit the form with what is required to create a new job offer:

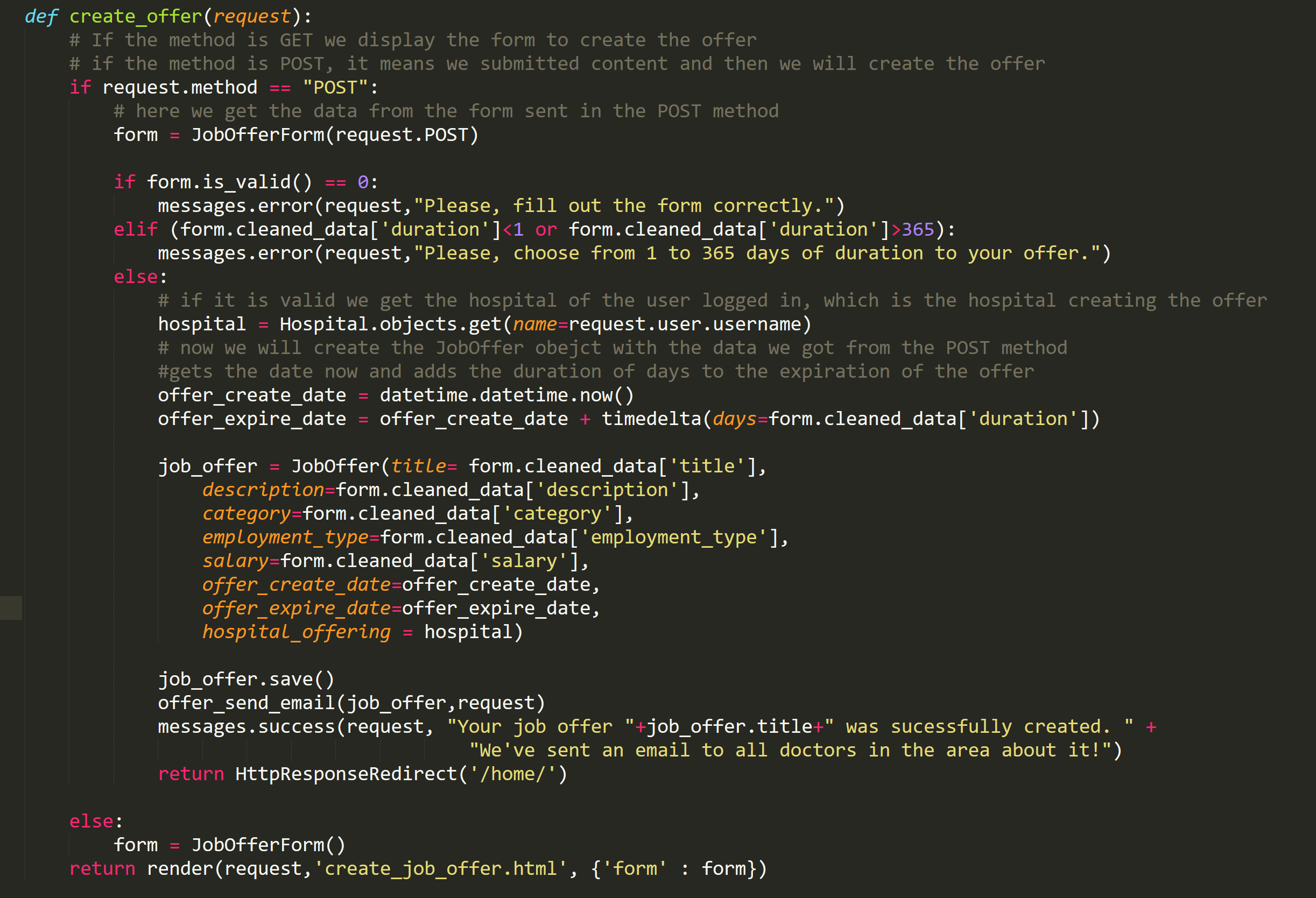


Figure 16 - views.create\_offer

We receive the HttpRequest as the parameter of the function and check its method. If the method is not POST we create a form with the JobOfferForm with all the fields required to create a job offer. Since that was no data submitted and the method was GET the return of the function will be template of the ‘create\_job\_offer.html’ with the form we created.

If the method of the HttpRequest was POST, there was data submitted. We get the form with all the information submitted and check if it is valid. If it is not valid or the duration which the job offer will be visible is not in the range we display a message to the user. If the form is valid, we get the hospital of the user logged in and define the variables for the create date and expire date of the offer that we will use when creating the job offer object. Then we create the job offer object, save it, send an email with the offer created, show a success message and redirect the user to the home page.

**Code snippet for views.offer\_send\_email:**

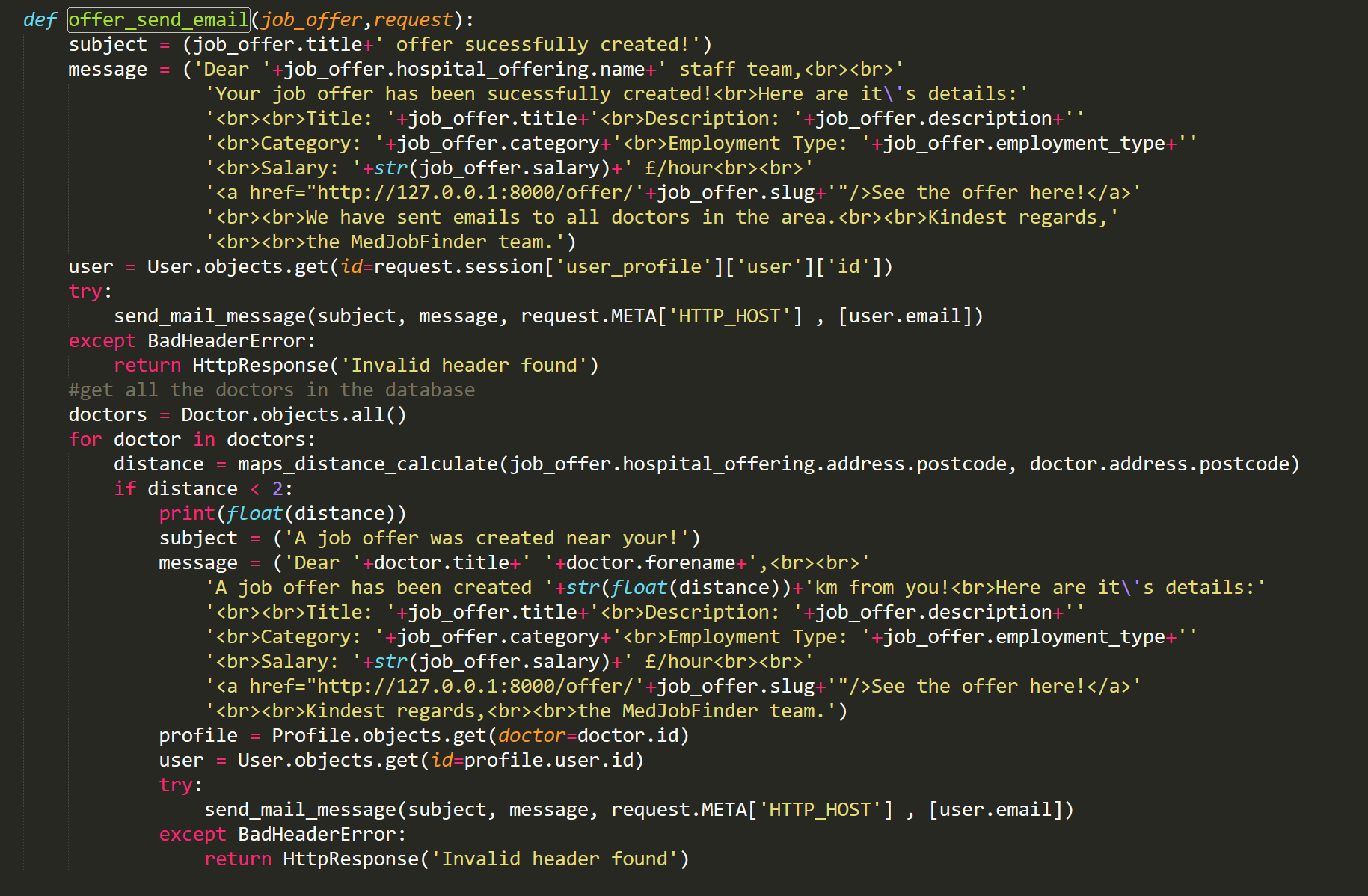


Figure 17 - views.offer\_send\_email

This function is responsible for sending emails after the hospital has created a new job offer. In the end of the create\_offer() we call this function passing the HttpRequest and the job\_offer object. Inside the offer\_send\_email() we create the subject and message retrieving data from the job\_offer object and get the user who is logged in this session and created the offer then we try to send the email. After sending the email to the hospital informing the success of the creation we send emails to doctors near the hospital location, who might be interested in the job offer. To achieve that we get all the doctors in the database and then for every doctor we calculate the distance between its home and the hospital address. If the distance is less than 2km we set the subject and content of the message, get the profile associated with the doctor, get the user and then try to send the email.

**Code snippet for the cv upload**:

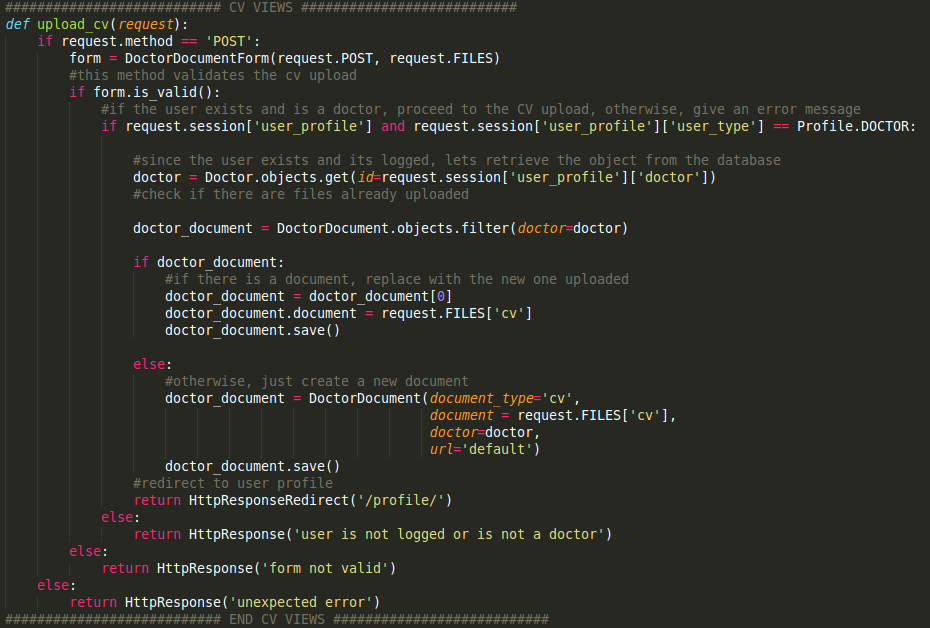


Figure 18 - views.upload\_cv

Since the CV is uploaded via POST by the front-end, first we need to check if there is a method POST in the request so we can continue to process the data. We define a form which will receive the CV data (request.FILES) and check if this file is valid for the type of form (a DoctorDocumentForm, which contains an FileField attribute). If this form is valid, we can continue with the checks for the type of user. In this version of code just the Doctor can upload a CV, so the next step is to check if the user currently logged is a Doctor. After this step we retrieve the Doctor object from the database and check if there is already a Document uploaded. If so, we just replace the document and return to the page that called the function. If not, a new DoctorDocument object will be created, saved in the server’s file system and the function will return to the page that called the function. The else cases contained in this function include the cases where the user is not logged, the request method is not POST or any other unexpected error.

**Conclusions**

This project was a great ending for our study abroad at Goldsmiths, University of London. We were able to use the skills learned in the subjects and apply them to a real world problem.

Some challenges also came in our direction in which we were able to solve quickly, such as the programming language definition. With the advice gave by Dr. Ouarbya, we defined that Python would be an appropriate language for this kind of problem, because it could be expanded with machine learning algorithms afterwards. With this platform, we believe that doctors will be able to connect truly and effectively with potential employers. This web app is a first step to the confirmation of the hypothesis that there is a gap in the market and that even students like us can use computing to build something that makes someone else’s lives easier and better.

Next steps for this platform is to put that in motion with some hospitals and doctors, apply some Artificial Intelligence, improve the security of the system and use some Design Thinking techniques to find more requirements and understand more closely the problem that we are facing. We look forward for the next steps.

We also would like to thank all the tutors from the Computing department and the Institute of Management Studies, which made this year an amazing one for not only academic learning but also for a huge cultural learning that is going to impact positively in our careers in the future.