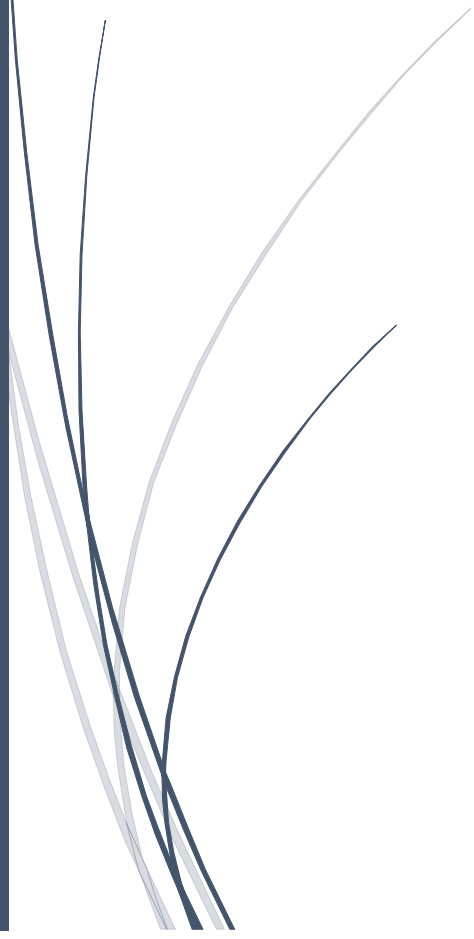
11/13/2015

Crisis Management System

Architecture Design Document – V2.0

DONTCRYSIS SOFTWARE SOLUTIONS



DontCrysis Overview

DontCrysis deals with different type of crisis/events like Fire, Industrial Accident, Thunderstorm, etc. A person can report a crisis or an event to a call center employee by calling the Call Center Hotline. The trained official then verifies the validity of the event and creates a crisis event in the system. All the information regarding the event is stored in the system database. The official can view the list of events, edit, activate or deactivate the events using the dashboard, when deemed necessary. Email notifications are sent to the subscribers within the area where an event is reported whenever an event is reported or updated. Also, a SMS notification is sent to the relevant agencies so that necessary steps can be taken.

The event is also instantly updated on the map by the system (using Google Map API) and it is accessible to public through 'DontCrysis' website link. After the event is over, the official deactivates it and the marker for that particular event is removed from the map and users are notified accordingly. Also, a report is sent to the Prime Minister's Office after every 30 minutes for them to monitor the situation of various events happening around Singapore.

From Use Case to Architecture

1. Final Architecture Diagram

Figure 1 (below) shows the final architecture diagram for DontCrysis's system.

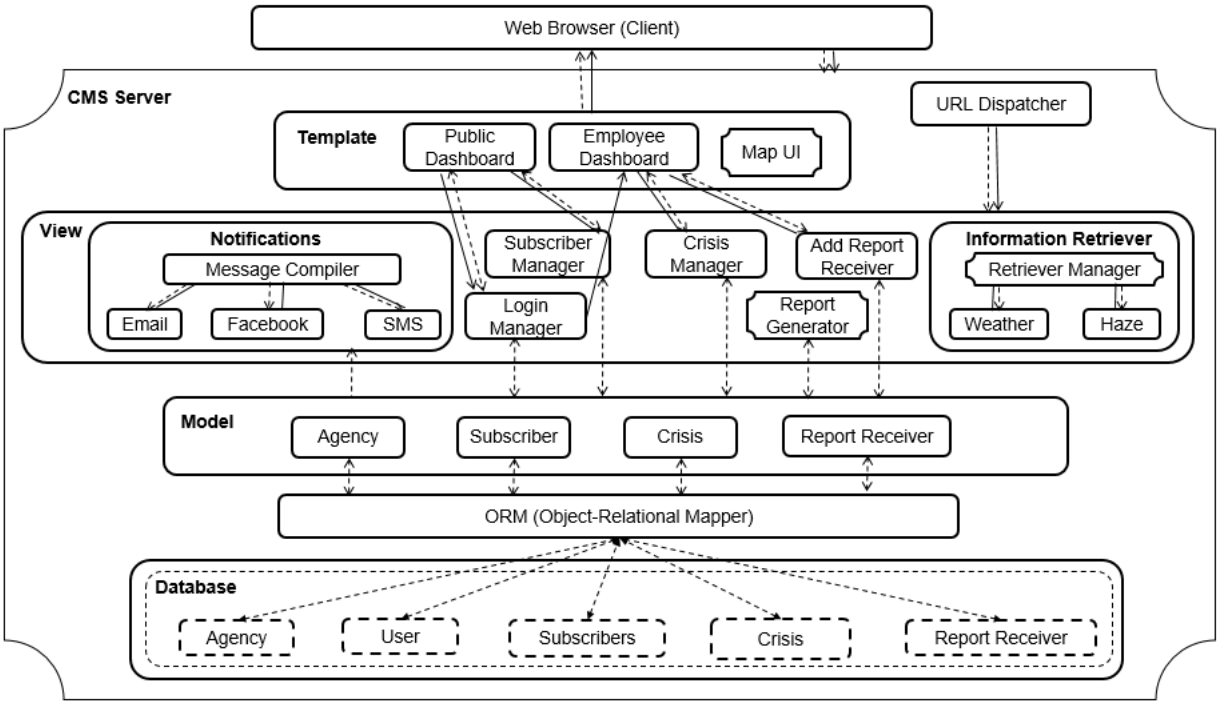


Figure 1 - System Architecture

2. Identify System Components

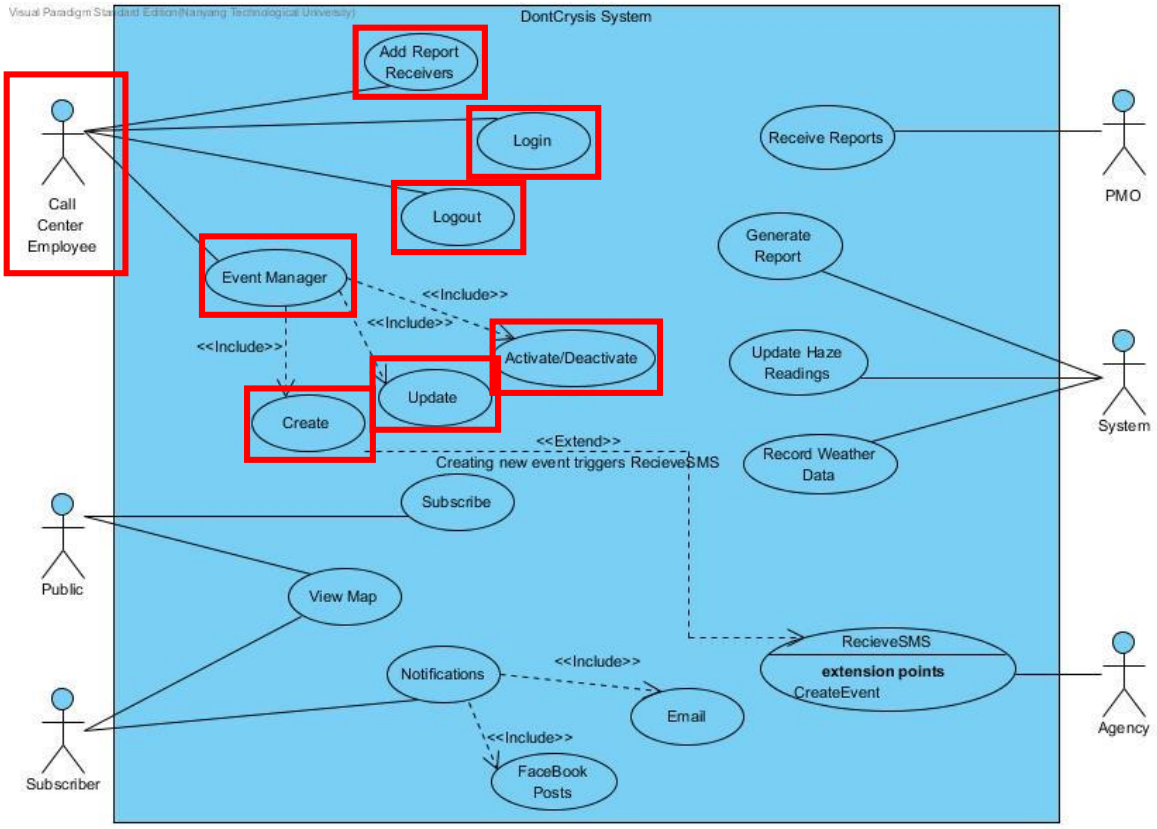


Figure 2 - Use Case (Call Center)

The first set of use cases are related to the call center employees/officials. To support this set of use cases, we require

- *Employee Dashboard:* The GUI which the call center employees use to perform the delegated tasks
- *Public Dashboard:* The GUI used by the call center employees to login and proceed to the employee dashboard
- *Managers*(Crisis Manager, Add Report Receiver, Login Manager): control and data flow (event information)
- *Notifications*(SMS, Facebook): to post about the crisis on Facebook page and send out SMS to concerned agencies for the reported crisis
- *Database:* to store event and report receivers information

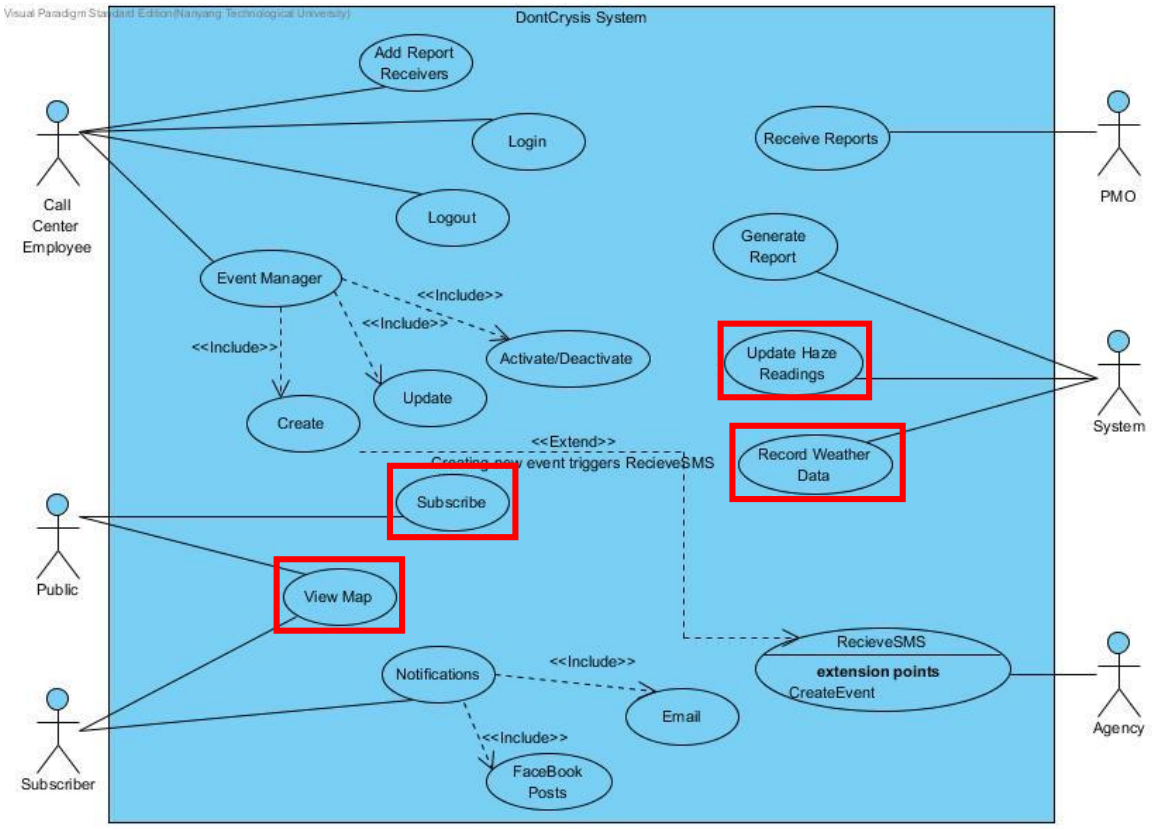


Figure 3 - Use Case (Public)

The second set of use cases are related to public and the system itself (update the information to be viewed to the public). These users are able to view information about events on the map and subscribe to notifications. For supporting them, we require

- *Public Dashboard*: To access the subscribe option if they wish to subscribe
- *Map UI*: To display the map in a proper format with necessary information
- *Information Retriever*: To retrieve the information about Haze and Weather using their respective APIs and then use it to update on the map through Google API

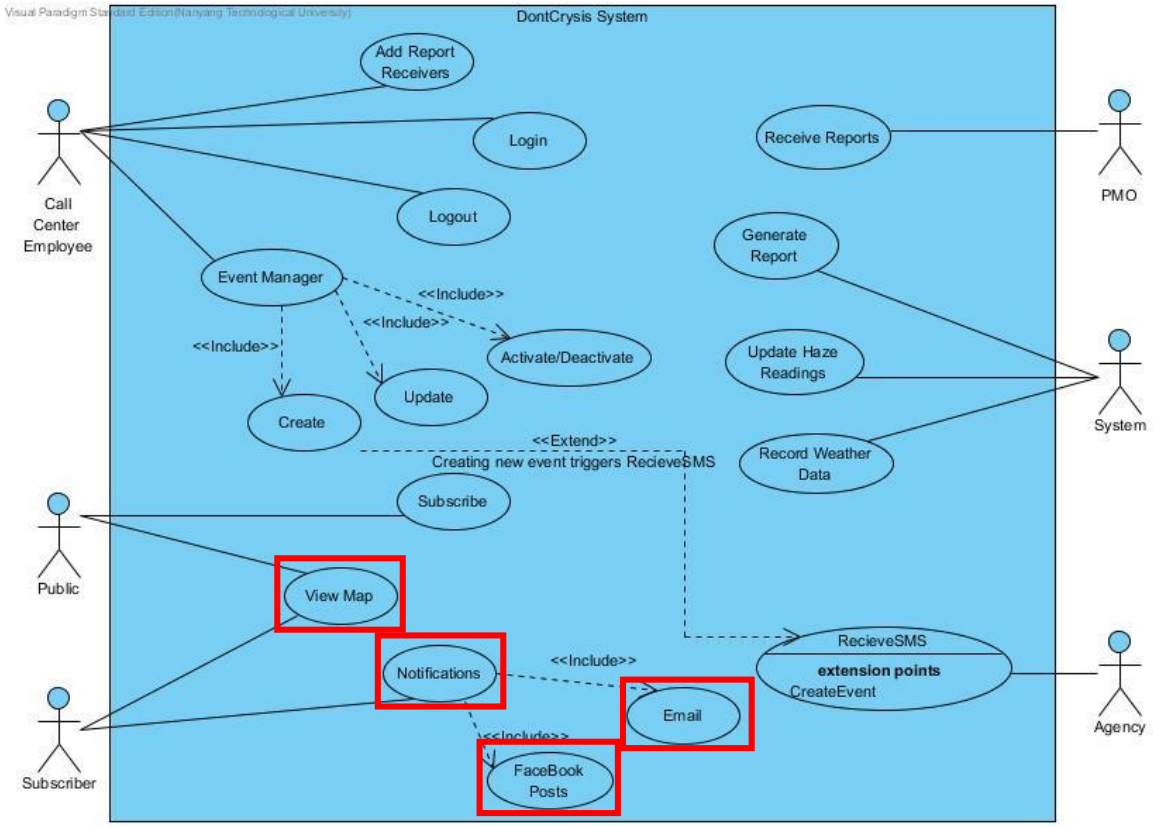


Figure 4 - Use Case (Subscriber)

The third set of use cases are related to subscribers. These users are able to view information about events on the map and subscribe to notifications. For supporting them, we require

- *Map UI*: To display the map in a proper format with necessary information
- *Notifications (Email)*: To send out email notifications to subscribers when a crisis is reported
- *Database*: To access information about the crisis and retrieve email IDs of subscribers

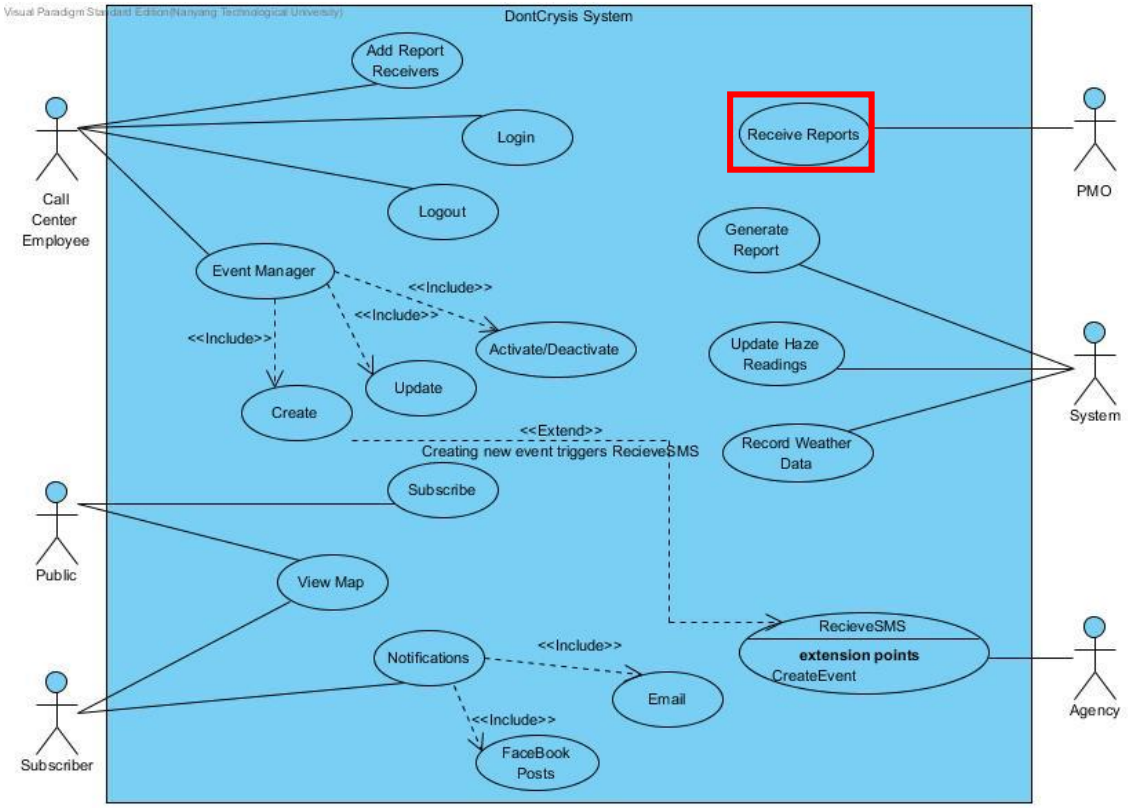


Figure 5 - Use Case (PMO)

The fourth set of use cases are related to PMO who receive reports after every 30 minutes. To support it, we require following components

- *Report Generator*: To generate reports to be sent to PMO
- *Notifications (Email)*: To send out generated reports via email to PMO
- *Database*: To access information about the crisis and retrieve email IDs of report receivers

3 Grouping & Connecting Basic Components

The components identified above can be grouped together (as shown in Fig 6). The services provided by the system like email notifications, Facebook posts, login, crisis reporting, etc. can be grouped together as one unit under View. The user views can be grouped together as templates. The View, Template and Database together forms the system server and web browser is the system client. The client sends in an Http request to the server which is processed by the view and then a HTML response object is returned to the browser. (This is facilitated by Python Django Framework, explained in the next section).

The grouping helps identify functional units, thus creating independent blocks, which support *reusability* and *maintainability*. The use of Django framework makes the system highly *scalable* and *secure.*

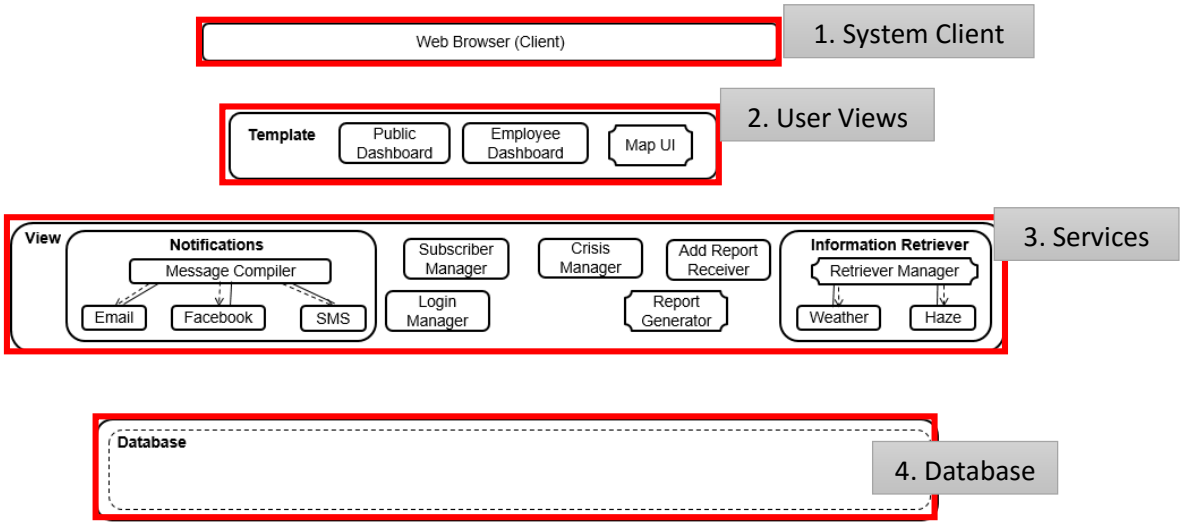


Figure 6 - Grouped System Components

Python Django Architecture (General)

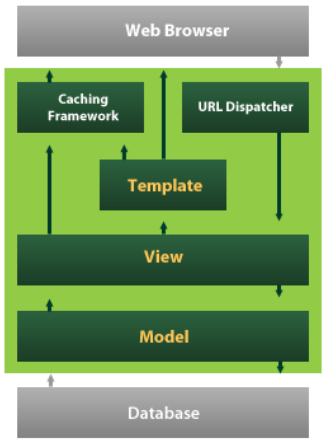


Figure 7 - Django Architecture

1. The URL Dispatcher maps the requested URL (requested by client, in DontCrysis client is Web Browser) to a view function and calls it. If caching is enabled (Not

Enabled in DontCrysis), the view function can check to see if cached version of the page exists and bypass all further steps returning the cahced version instead.

2. The view function peforms the requested action, which typically involves reading or writing to the database. It includes other tasks, as well.
3. The model defines the data in Python and interacts with it. In DontCrysis it is contained in MySQL relational database.
4. Templates return HTML pages. The Django template language provides a simple-to-learn syntax providing all the power needed for presentation logic.
5. After performing any requested tasks, the view return an HTTP response object (after passing the data through a template) to the web browser.

DontCrysis, Python Django Implementation

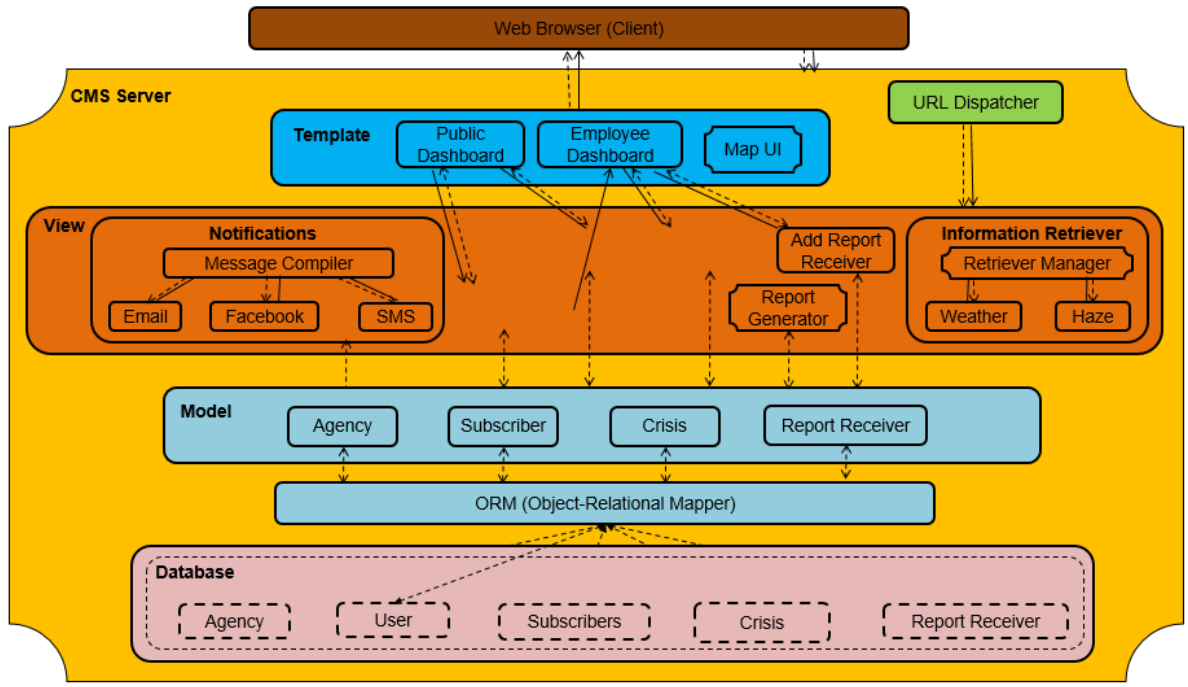


Figure 8 - Django Implementation: System Architecture (MTV Framework)

Figure 8 shows the implementation of Django architecture in DontCrysis system. MVC pattern is closely followed, however since the *Controller* part is handled by the framework itself, it is best to be interpreted as MTV (Model Template View) framework.

MVC Interpretation

- ***Model***: *It is the* data-access portion and is handled by the database layer, which is described by the Model component and ORM (Object-Relational Mapper) as shown in fig 8.
- ***View***: It is portion that selects which data to display and how to display it. It is handled by views and templates as shown in fig 8.
- ***Control****:* It is the portion that delegates to a view depending on user input, is handled by the framework itself by following the URLconf and calling the appropriate Python function for the given URL. It also reads a settings file so that it knows what to load and set up. URLconf is specified inside a python script file named *urls.py.* In the above diagram it is depicted by *URL Dispatcher.*

MTV Interpretation

- ***Model***: It refers to the data access layer. This layer contains anything and everything about the data (as shown clearly in fig 8), how to access it, how to validate it, which behaviors it has, and the relationships between the data.
- ***Template***: It represents the presentation layer. This layer contains presentation-related decisions: how something should be displayed on a Web page or other type of document. In fig 8, public dashboard, employee dashboard and MapUI are the templates, HTML pages which are rendered by the view and returned ad HTTP objects to the client i.e. web browser.
- ***View****:* It represents business logic layer. This layer contains the logic that access the model and defers to the appropriate template(s). It acts as a bridge between models and templates.

MTV framework in Django and DontCrysis implementation can be interpreted as other MVC web-development framework like Rails by considering Django views to be 'controllers' and Django templates to be 'Views'. In DontCrysis's interpretation of MVC, the "View" describes the data that gets presented to the user/client and it's not necessarily just *how* the data looks, but *which* data is presented. In contrast, other web based MVC frameworks like Ruby

on Rails suggest that the controller's job includes deciding which data gets presented to the user, whereas the view is strictly *how* the data looks, not *which* data is presented.

Software Quality Attributes Supported

1. Scalability

The system in implemented using the Django framework that is based on stateless shared-nothing web technology. It was ensured that the web nodes (running Python-Django code) are independent from our persistence layer that includes database, caching and session storage. Since the Django web nodes have no stored state, they scale horizontally - just fire up more of then when you need them. Being able to do this is the essence of good scalability. Even, current systems such as Disqus and Instagram prove the scalability of Django which have millions of users using them.

2. Portability

Since the system is coded in Python using the Django framework, it is highly portable. Django and Python are supported cross platform and in addition support almost every database. All that is needed to change is the settings.py file which can also be done using some conf file changes. The system can run on various OS whether it be Unix-like or Windows and can supports all the databases.

3. Integrity

In our system, we are using inbuilt Django User Authentication package that is very secure. For user login, by default, Django uses the PBKDF2 algorithm with a SHA256 hash, a password stretching mechanism recommended by NIST. This should be sufficient for most users: it's quite secure, requiring massive amounts of computing time to break. Besides user authentication, we use CSRF (cross site request forgery) protection every time the user enters the details in any forms. CSRF (cross site request forgery) protection is easy to turn on globally and guarantees that forms (POST requests) are sent from your own site.

Moreover, the presentation layer was coded in Django HTML Templates that ensures XSS (cross-site scripting) protection whereby Django template system by default escapes variables, unless they are explicitly marked as safe. In addition, Django uses

built-in ORM, thus there is no risk of SQL injection (raw queries are possible, but by no means something that a beginner would need to use)

4. Flexibility

The system, due to its MVT architecture, is loosely-coupled and also the python Django culture is very idealistic. Often times it does not have shortcuts you find in other languages, because the culture is very against having multiple ways of doing things. In addition, python Django framework has a heavy emphasis on backwards compatibility. This improves readability of the code, something that python Django excels in. Therefore, the goal of simplicity is achieved which makes it easy to add new features easily and maintain codebase.