# System Documentation

## Salalah Methanol Company

## Inventory Management System

July 12, 2018

**Devan K S**

# Contents

# 1 Introduction

The SMC inventory management system allows the user to create and manage records of information related to users and the devices they have been assigned to by the IT team. The system features various validation features to remove errors in the information. It contains graphs to show information in a clearer manner. This is a system documentation to help out any future developers of the project. Many steps followed has alternate methods too. Any pethood can be used according to the users wish. A user needs to have minimum knowledge in Python and Django framework.[1]

# 2 Setup Pipeline

A computer with internet connectivity is required for setting up the development environment.

Administrative rights might be required for some steps and some firewalls might block some essential sites required updating and installing packages. Due to this reason we can easily set up a development environment online for our purpose. If you want to setup a local development environment, please check **Appendix A** This pipeline would consist of "GitHub" and "Heroku" (If testing, "TravisCI" or "CircleCI" too[2]). Use the latest browser for all the steps requiring one.

*The services mentioned in the documentation are only slightly touched upon. It is out of scope of this document to explain every feature of the services in the pipeline and only the bare essential steps are listed.*

*(Latest versions of Edge and Chrome has been checked for the below steps and can be used)*

1 Set up GitHub

- 1.1 Sign Up/Login on GitHub
- 1.2 Set up repository
  - 1.2.1 Install Git based on the OS[3]
  - 1.2.2 Extract the code for the system into a folder
  - 1.2.3 Set up GitHub to push the folder to Origin[4]

[1] Django can be learned by doing the project in **https://docs.djangoproject.com/en/2.0/intro/tutorial01/** .
[2] These are automated testing online application which can easily be joined to the pipeline. This is beyond the scope of the document, but an example ".travis.yml" for TravisCI and a python test has been added for ease of building a better testing environment. The "test.py" file presently doesn't test anything.
[3] Available from *desktop.github.com* and *git-scm.com* for windows. The former is suggested as it has a GUI.

2 Set up Heroku

2.1 Sign Up/Login on Heroku

2.2 Make a new application in Heroku

2.2.1 From your dashboard create a new app in any stage(Preferably development or staging)

2.2.2 Choose GitHub as the deployment method and connect the account and the repository created in step 1.2

2.2.3 Activate automatic deployment on a branch[5] of the repository for easier development

2.2.4 Check if automatic deployment works by pushing from GitHub app by adding a random text file. Heroku should configure the app according to the various files already in the repository.[6]

2.3 Set up the newly created application

2.3.1 From the settings of the newly created app, set up "Config Vars" (Environment variables)

2.3.2 Set "DJANGO_DEBUG" as "True" if the app is sued to develop new features. Else set it as an empty string for using the app in a production environment.

2.3.3 Set "HEROKU_ALLOWED_HOST" as the website of the app when the "Open app" option is used in Heroku. It will be "*<app_name>*.herokuapp.com".

2.3.4 There should be a "DATABASE_URL" variable set. If not, the PostgreSQL database add-on didn't get installed. Read **Appendix B**. If it is available continue on to the next step.

2.3.5 Select more in Heroku and Select "Run Console".

2.3.6 Run the command "python manage.py migrate && python manage.py createsuperuser".

2.4 The site should be accessible now from the "Open App" option in Heroku.

[4] Go to **https://help.github.com/articles/adding-an-existing-project-to-github-using-the-command-line/** link.
[5] There are branches in GitHub which can be used for better development, splitting production and development environments. This is beyond the scope of the document but can be easily be learned from sites.
[6] Files used by Heroku explicitly are "Procfile"," requirements.txt", "runtime.txt" and "wsgi.py"

# 3 Code Breakdown

As the pipeline has been set up the developer can start developing. For the project multiple famous and stable APIs and libraries have been used.

The system has been built using Django Framework (Version 2.0.5) with Python (Version 3.6.5). Major code of the application is built using the Django Admin User interface. The interface has been used and changed for further requirements of the application like Graphs.

## *3.1 Directory Tree*[7]

```
+---inventoryManagement
|    +---adminFiles
|    +---migrations
|    +---modelFiles
|    +---modelsList
|    \---templates
|         \---admin
|              \---import_export
+---SMCManagement
|    +---extraSettings
+---static
|    +---css
|    +---favicon_images
|    +---images
|    +---js
```

## *3.2 "inventoryManagement" Directory*

This directory contains all the code required for the app according to MVT model. Most of the original files has been made into modules to better control development process[8]. Therefore, there isn't any information to change in the original files created when a Django project is created[9]. The files are made into functions which are put in to specific folders. "tests.py" isn't extensively created, and will be limited to a file. If more tests are created it can be made into modules and put in a separate folder. The "apps.py" file is to set a verbose name for the app in the admin interface. "urls.py" and "views.py" had been altered in an attempt to provide custom 404 and 505 error pages[10].

[7] In a PDF document, most of the folders are linked to their respective headings in the document.
[8] Files are "models.py" and "admin.py"
[9] There exists a *"__init__.py"* file in almost every directory. This file exists for making each folder into packages so that the files in the folder can be imported in other scripts. Initializing code can also be put in the file.
[10] This doesn't work and presently provides the default error pages of Django. The error is easily fixable.

### 3.2.1 "modelFiles" Directory

This directory contains all the necessary custom functions and features for creating the models in the "modelsList" directory. The files are:-

1. *"CustomCharField.py"*: This file contains customized char fields to be used in the models for better standardized input to database. It contains two classes. The "UpperCaseCharField" and "TitleCaseCharField", as their names suggest, change any input received into their respective case. This is done inheriting from the "models.CharField" and initialize with all of its properties. Then a pre_save function is defined (which is automatically called when an object is saved) which takes the input value and converts it to the required case using a built in function.

2. *"CustomValidators"*: This file contains "RegexValidators" which are used to validate inputs received from the form available in the admin view. An explanation regarding regular expressions is beyond the scope of this document. Three different validation objects are made with messages for the user and codes to detect errors in views.

### 3.2.2 "migrations" Directory

The migrations directory contains one initial migration done which contains all the changes to be made in the database. This file is run when *"migrate"* is executed.

**THESE FILES SHOULD NOT BE ALTERED.** If altered, the reversibility of the *"migrate"* command is lost. For any changes to migration, change the model files accordingly and run *"makemigrations"* command to make new migrations in the folder. These can be set into the database by running the *"migrate"* command. If alteration is required, make sure there is an appropriate command to reverse the change in the same migration file for better coding practice.

### 3.2.3 "modelsList" Directory

This directory contains the models themselves. They contain a simplified abstraction of the model in a database. Any changes made to these files will be reflected in the database. The changes ARE NOT automatic and must be started using the *"python manage.py makemigrations"* [11]and *"python manage.py migrate"*[12] commands. "__str__" function is declared so that the object gives the attribute value when used in places where the object is returned. "verbose_name" can be changed for altering the text displayed regarding the models views. The files in this directory are:

1. *"deviceListModel"*: The model "deviceList" has only one attribute, "deviceName". This attribute and model was created so that during input of records in recordModel the user has to ensure the device is name is typed correctly. A future alteration might bring the same property to departments. This is important as the device and department names are grouped in the graph functions and any mistakes can result in wrong grouping.

2. *"recordModel"*: The model "record" has many attributes which has multiple validators and conditions. It also initializes "HistoricalRecords ()" which creates an extra table in the database to keep better track of all changes happening to a record in the recordModel. It also contains a regular expression based validation happening to deviceTag before saving the object (*validate_deviceTag_pre_save ()*). This function is then connected to a signal "pre_save". This is added as an extra measure as "import" function of the "import_export" library doesn't validate using the validators declared. Possible alterations in the future will include a similar function for other important attributes.

3. *"recordSummaryModel"*: This file contains a proxy model, "recordSummary", based on "record" model. This model is created to add an extra link to a custom "changeList" template from the home page of the admin page.

[11] This command will be referred to as *"makemigrations"* command from hereon.
[12] This command will be referred to as *"migrate"* command from hereon.

### 3.2.4 "adminFiles" Directory

This directory contains the split up functions which originally existed inside of "admin.py" file. Now all the functions are imported in the file from this directory. The files are:-

1. *"ModuleCachingPaginator.py"*: This a replacement for the default Paginator used by the Django admin interface. This was necessary as the default Paginator added an extra "count *" query to every webpage with the Paginator module wasting a query's time. This module does the query in the beginning and the sets it as a cache in the user's computer helping speed up subsequent load times.

2. *"ModuleDeviceListAdmin.py"*: This is where the admin class for the "deviceList" model is made and this class is imported and registered with the admin in the main "admin.py" file.[13] "list_display" allows you to show any other attribute in the model when showing the table in the view. In this case, there isn't any other available attributes in this model.

3. *"ModuleRecordAdmin.py"*: This is where the admin class for the "record" model is made. This class is imported and registered with the admin in the main "admin.py" file. The "list_display", "list_filter" and "search_fields" can be modified by adding or removing attributes already existing in the model. [13]

4. *"ModuleExportActionModelAdmin.py"*: This creates an inheritable admin class created from the library "import-export" joining its import mixing and export action model admin.

5. *"ModuleRecordResource"*: This file sets up any required settings for the "import-export" library. The file initializes model "record" with a resource. The "id" attribute is ignored during import and export. "deviceTag" is considered to be attribute uniquely identifying each record for the input, allowing the module to understand when records similar to the ones in the database are being imported and change the detail already in the database accordingly.

6. *"ModuleSummaryAdmin":* This module creates an additional admin page using "recordSummary" proxy model created. This file points to a different "changelist" template which has been modified to provide graphs. The query has also been edited to show a department and device based grouping. The attributes in values function can be changed to offer different grouping and annotate is used in a queryset to make an attribute not a part of the original model.[14]

[13] "Show_full_result_count" is disabled to stop another extra "count *" query which is run to get a total count for the view. The Paginator is also changed to a custom made one for the same reason.

[14] Changing this file and its corresponding changelist template is very hard and must be attempted only if you have full knowledge of the system. It is better to create a new proxy model and follow a similar procedure.

### 3.2.5 "templates" Directory

This directory contains the custom made templates which differed from the original Django admin interface. This is especially used to set up the graph page as Django admin doesn't provide a default way to set up a slightly different template than replacing the whole thing. It also contains different templates which might be used in the app. In this particular scenario all templates required exist inside admin folder inside this folder.

The following information are details regarding changes made to default templates or new ones made.

1. *"base.html"*: This template has been modified to include favicons and other created static files. It adds a background logo. Added an empty preload block.

2. *"base_site.html"*: This template has been modified to include the loading animated company logo container inside the preload block defined in "base.html"[15].

3. *"change_list.html"*: A "graph-tools" block is added. Some extra styles are applied and required JavaScript and css files for the custom "dropDownFilter" are imported. The filter is also put in cache for 500 seconds on first load to avoid extra queries.[16]

4. *"filter.html"*: This template changes the original filter page drastically by removing the original list of possible filter options to a hidden dropdown list which is accessible from each filters own search box.

5. *"record_summary_change_list.html"*: This template is a custom made template extended from "change_list.html". This uses the "graph-tools" block and adds styles, the library "chartJS" and a JavaScript file containing code to create the chart and also added sorting ability to the table. The department value takes the user to record list with the department name filter on. The device name takes the user to record list with both department and device filters on. The URLs are encoded too. Pagination is removed. The total is changed to use commas instead of just displaying large numbers using "intcomma" of "humanize" library.

6. *"import_export/import.html"*: This is changed so that output is better arranged and the traceback, in case of errors, isn't displayed on load. Please note that the traceback can be easily accessed by using developer tools of any modern browser. So, **this isn't a permanent solution** and a method of turning off traceback of the "import_export" library when "debug" = False must be devised in the future.[17]

[15] Yes, blocks in the parent class can be overwritten by a template extending from it.
[16] During the creation of the document, due to an error in "Django-Debug-toolbar", the filter cache will not work properly. As the toolbar is only active during debugging, **Production system isn't effected by the error.**
[17] A possible solution can be to make script which remove the element entirely. This will face issues when used in a browser blocking the execution of JavaScript. So, a style for hiding the element is necessary anyway.

## *3.3 "SMCManagement" Directory*

The "settings.py" file has been split into multiple modules and put into the "/extraSettings/" folder. The file contains variable like "SECRET_KEY","DEBUG" and "ALLOWED_HOSTS" which has default values and also takes in values from the environment variables. These environment variable names are important when setting the app up for production as the variables must be given values. **MAKE SURE THE "SECRET_KEY" IS SET ONLY BY USING AN ENVIRONMENT VARIABLE.** The file sets the "BASE_DIR" variable.

### "extraSettings" Directory

This directory contains all the modules of the "settings.py" file. Most of the files contain the exact information as their names and an in depth detailing of the files is out of scope of this document. So only a summary of what some files do are mentioned below.

1. "databaseSettings.py": The file sets up a local mysqlite3 database if ever the developers sets up a local development environment. It also uses the environment variable "DATABASE_URL" if it is set to replace the local database with the database URL provided.

2. "debug_toolbar_settings": This contains the settings required for the Django-debug-toolbar to work. As there are many panels available, it can be added or removed accordingly by referring to the documentation for it. Unlike the default settings, the Django debug toolbar shows up whenever "debug"=True. While, the documentation asks for URLs to be modified so that it shows only when a specific URL token is there.

3. "logging_settings": Turns on console logging. The settings file doesn't import the file when "DEBUG" is false.

4. "middleware": **Make sure that the order of middleware is carefully checked**. Each middleware has a specific position for it work and editing the file must be done with utmost care.

5. "productionLevelChanges": These changes are made to make the app production ready. If these changes are to be switched off for some reason, it is better to comment out the import statement for this in the original settings file.

6. "staticFilesSettings": This file sets specific variables required for static files. If an extra folder with static files exist somewhere it must be added to the "STATICFILES_DIRS". The "STATIC_ROOT" variable decides where the command *"python manage.py collectstatic"*, hereafter referred to as *"collectstatic"* command, will collect the files. "STATIC_URL" is used when the files are needed.

7. "templatesSettings": This activate cached template loaders which may be deactivated when the app isn't in production.

## *3.4 "static" Directory*

This directory consists of every static object to be used in the Django app. These objects includes stylesheets, JavaScripts and images. Some files have been joined to reduce number of HTTP requests made, while others have been "minified" by online resources to reduce the size. This is done automatically done by "Whitenoise" [18] too. **Do note, if a file, which also has a ".min." version, changes make sure the changed file is "minified" and updated in the ".min." version.** Most files are small and simple enough that they don't warrant extra explanation[19]. Following are the ones which require some explanation:-

1. "js/SortChart.js": This file is just the combination of "js/ChartFile.js" and "js/tableSortFile.js". This file also has a "minified" version which is actually the version used for the app. **Due to this, any change in "ChartFile.js" or "tableSortFile.js" must be made in their respective "minified" versions and the combined JavaScript file and its "minified" version**.[20]

2. "js/tableSortFile.js": This contains a function which will be called when the header of a column is clicked in RecordSummary page. A better sorting algorithm can be implemented if necessary.

3. "js/ChartFile.js": This file contains functions which get run when the user clicks on the "render graph" button in the recordSummary admin page. This file can be further modified easily to allow any graph type offered by the "ChartJS" library. The explanation of the code can be found in **Appendix C**.

[18] "Whitenoise" even offers gzip compressed files.
[19] None of the Stylesheets are going to be detailed as they are basic concepts of CSS
[20] For any change there will be at least 3 other files to be changed. This is cumbersome, but effective in reducing number of requests and the size of file. Reducing the number of requests will not be necessary when HTTP/2 is widely available.

# 4 Appendix

## *4.1 Appendix A*

Setting up a local development environment is easy if an account with administrative capabilities is available. The most important requirements for a local development environment for this project is:

1. Python 3.6[21]
2. Pip (Installed automatically on windows, when Python is installed)
3. SQLite3
4. VirtualEnv[22]

The steps to set up the local development environment is listed below. There may be extra dependencies to be installed in any of the steps below. These must be installed for the steps to work:

1. After the essential requirements are installed, make sure these applications are executable from command line. If they are not, add them to the "Environment Variables". SQlite3 **HAS** to be added manually.
2. After the apps are installed, use VirtualEnv to create an environment for the project anywhere. After this, activate the virtual env by running the file "/Scripts/Activate"[23].
3. Then "pip install –r requirments.txt" can be run when in the home folder of the application to install all required python packages for the project.
4. Create an empty database in the home folder of the app using "sqlite3 localdb.sqlite3".
5. In the "SMCManagement/settings.py" file comment out the import command for Production level changes.[24]
6. Now, after "migrate" command is run, the development server can be started by executing the "python manage.py runserver" command.

[21] Check **"runtime.txt"** in the home folder of the application for exact version. This exact version is preferable.
[22] Check **http://docs.python-guide.org/en/latest/dev/virtualenvs/#lower-level-virtualenv**
[23] The position of this script can defer according to operating system.
[24] This is ABSOLUTELY necessary when working on a local environment as this file forces HTTPS connections.

## *4.2 Appendix B*

In most cases, restarting the Dyno would fix the issue of a PostgreSQL database not getting attached to the Heroku App. If it still doesn't work you could manually add the database, following are the steps for that:

1. In the "Overview" tab of the app, click "Configure Add-ons".
2. Under "Add-ons", click "Quickly add add-ons from Elements" and search for "PostgreSQL".
3. Just click the "Heroku Postgres :: Database" to add it to the app.

## *4.3 Appendix C*

*ChartJS Code Explanation*

The rainbow () takes in two arguments and returns a RGBA code for a color. What the function does is it splits the whole spectrum of colors into the argument specified by "numOfSteps". After which it returns the RGBA value of the color using the variable "step" similar to an index.

The renderCanvas () gets executed when the "Render Graph" button is clicked on the "Record summary" admin view. The function takes the table and splits the information different arrays containing the list of departments, devices and their totals. These values are used to initialize the chart using the variable "myChart". The "type" in this definition can be changed for different graphs.[25] The "Labels" attribute can be changed to something like building or room, but that value must appear in the table, which means "list_display" must be changed accordingly. And to make the graph mean something useful the query in the file "**adminFiles/ModuleSummaryAdmin**" must be changed accordingly.

[25] Refer to ChartJS documentation for better understanding of this and to know which graphs are supported.