GrantSeer Documentation 1.0

Yang Song
yxs176@cse.psu.edu

Pennsylvania State University

1 Introduction

This document aims to guide you design reusable apps and loosely coupled system specifically based on new GrantSeer. GrantSeer originally developed by Dr. Puck Treeratpituk and written in Ruby on Rail. However, to improve the readability of the whole project and make it more maintainable, we decide to rewrite the system with Python and Django.

Who need to read this documentation

1. Anyone who wants to improve the current GrantSeer project.
2. Anyone who wants to create new web apps that share the same database `csx_exp`.
3. Anyone who wants to know the very basic features about Django 1.6.

Who shoud not read this documentation

1. Anyone who would like to know how to create a Django web app from zero.
2. Anyone who works on a different database.

The complete reference and tutorial of Django could be found at https://docs.djangoproject.com/en/1.6/

Versions

Python 2.7

Django 1.6

2 GrantSeer Architecture

As we want to decouple the whole system, new GrantSeer is split into two services: `GrantSeerFrontendService` and `SeerBackendService`. In service design pattern, `Frontend Service` basically is responsible for interacting with users, generating request to `Backend Service` and rendering the data transferred from `Backend Service` for users. `Backend Service` is responsible for interacting with `MySQL` database, grouping and sorting those data and what not. In Django, `Backend Serivce` should be regarded as a data persistent layer of web app, even it could do more stuff.

Figure 1: GrantSeer Architecture

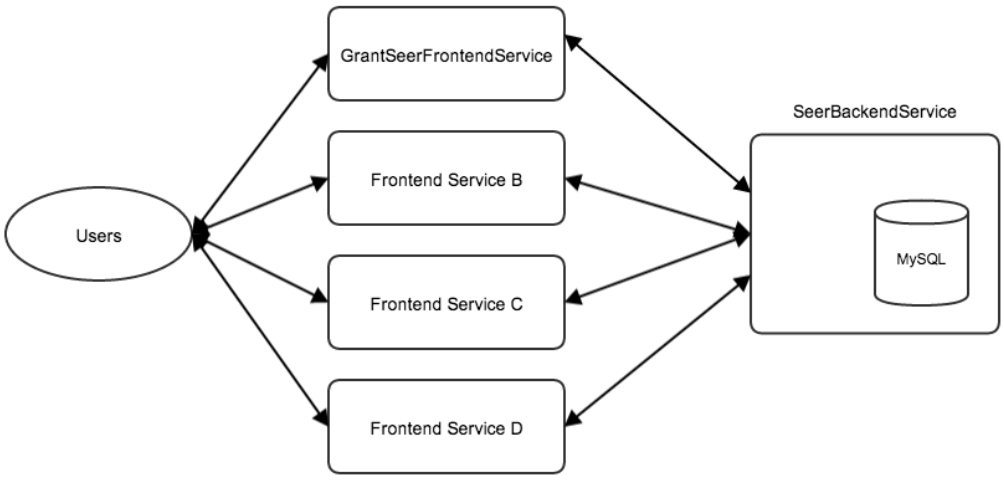

Figure 1 depicts architecture of GrantSeer. Currently only one frontend service exists. Developers are expected to add more frontend services that share the same database with `GrantSeer`. There are two reasons for this design.

1. Distribute the workload. User requests only go to frontend services and different frontend services will handle different requests. `SeerBackendService` only handles requests that come from frontend services.

2. Security. Only the ports that be able to reach frontend services are open. Malicious users are not be able to see `SeerBackendService`, so it's impossible to compromise backend service.

3. Fail-safe. Even one or more frontend services are down, the rest of frontend services will be working normally.

3 Creating Frontend Service Project

Your new frontend service should tightly-coupled with any other services in the current system, which means:

1. Your frontend service should only send request to `SeerBackendService` and receive response from it.

2. Your frontend service are NOT supposed to interact with other frontend services.

Since your frontend service has to comply with these two rules, you have to create a new web apps from zero without filling database related configurations in the `settings.py` file.

Details about how to create a empty project and new app, please refer Django official documentation[1], which will walk you through the basic things smoothly.

4 Creating New App in SeerBackendSerivce

This documentation will focus on some details about creating reusable apps in `SeerBackendService`. In this section, `SeerBackendService` folder always means folder `SeerBackendService/SeerBackendService`.

[1] https://docs.djangoproject.com/en/dev/intro/tutorial01/

URL Mapping

To make all apps in `SeerBackendService` maintainable, URL mapping plays significant role. In `SeerBackendService`, at least when we are designing this system, we create two `urls.py` files. One is under `GrantSeer` folder; another one is under `SeerBackendService` folder.

The `urls.py` file under `SeerBackendService` folder servers:

1. Django admin pages
2. index page. This page will take `HTTP GET` request and tells you if `SeerBackendService` is running or not.
3. constructs urls for each apps. We will discuss it in the following paragraph.

The `urls.py` file under `GrantSeer` folder only specifies its own urls that can be navigated to designated html pages. For example, `piname/$` url will navigates to the method `GrantSeer.views.pi_name`. But with the help of the `urls.py` under `SeerBackendService` folder, this url becomes `grantseer/piname/`. For each app, you should provide a index page that at least could provide information about the current status of the app.

In general, when you create a new app in `SeerBackendSerivce`, in `urls.py` under `SeerBackendService` folder, you should add a new url that includes all the urls of your new app. It is just like creating a namespace for each app so that you will not worry about duplicated urls when many people create new apps under the same project.

Models

Django strictly follows MVC(Model-View-Controller) pattern. You will find a `models.py` file under `GrantSeer` folder. Basically, this auto-generated file wraps the underline database table and you are not supposed to change the content of this file unless the database structure has been modified.

Currently there is only one app in this project, so we still put `models.py` under `GrantSeer` folder. But if you want to create a new app and reuse this file, you strongly encourage you to move `models.py` to `SeerBackendService` folder as this will indicate that this file is supposed to share with other apps.

Data Wrapping

When you splitting one component into two, data transferring is always on agenda. To the best of my knowledge, no consent about how your serialized data generated by backend service should be look like. In this project, we use JSON to wrap all the data.

When you look at `GrantSeer` web app source code, you may be curious about why we create may `DataWrapper` classes and `data_wrapper` methods in `views.py` file.

The only reason is that the data generated by `SeerBackendService` contains lots of fields, including both attributes in database tables and other fields, such as number of pages, error messages, etc. For table attributes, Django provides a easy to to convert each rows you fetched from database to JSON. What you have to do is to call `serializers.serizlize('json', query_result, fields=fields_tuple)`. However, what you get from that is only the JSON data of your query results. To add more information to that data, we have to new `Data_Wrapper` classes to hold those data and call `json.dumps(var(Custom_Data_Wrapper(...)))`. Then you are ready to send this JSON data back to frontend service.

However, if you could provide a more elegant way to wrap the data, please go ahead to modify the code and, if possible, to make everyone's life easier, provide APIs that everyone can call.