Public-contracts Documentation

Release 0.1

jorgecarleitao

May 09, 2017

Contents

Contents:

CHAPTER 1

Installation

This document explains how you can install the API for *accessing* the database. To deploy the website, see *Install development environment*.

We assume you know a little of Python and know how to install Python packages in your computer using pip.

Getting the source

The source can be either downloaded or cloned from the GitHub repository using:

```
git clone https://github.com/jorgecarleitao/public-contracts.git
```

Like most Python code, the source doesn't need to be installed; you just have to put it somewhere in your computer.

You need Python 3.

Dependencies

For using the code, you need to install three python packages:

Django

We use Django ORM to abstract ourselves of the idea of database and use Python classes to work with the data:

```
pip install Django
```

Postgres

Our remote database is in postgres. To Python communicate with it, we need a binding:

```
pip install psycopg2
```

treebeard

The categories in the database are organized in a *tree structure*. We use django-treebeard to efficiently storage them in our database.

Install using:

```
pip install django-treebeard
```

Running the example

Once you have the dependencies installed, enter in its directory and run:

```
python -m contracts.tools.example
```

If everything went well, it outputs two numbers:

1. the total number of contracts in the database, that you can corroborate with the official number.
2. the sum of the values of all contracts.

If some problem occur, please add an [issue](https://github.com/jorgecarleitao/public-contracts/issues) so we can help you and improve these instructions.

From here, you can see section *Asking questions to the database* for a tutorial, and section *API Reference* for the complete documentation.

CHAPTER 2

Asking questions to the database

This tutorial assumes you already *installed* the source.

We assume you are in directory "public-contracts", as the installation part ended there, and that you start a Python session from there (e.g. enter "python" in the terminal).

Setup

To use Django, we have to setup it first. In the module `set_up` of package `main/tools`, we provide a function to that, which we now use:

```
>>> from main.tools.set_up import set_up_django_environment
>>> set_up_django_environment('main.settings_for_script')
```

This sets up a minimal Django environment using the settings `main/settings_for_script.py`, a minimal configuration to use the database.

Accessing the database

In this API, objects are Django models, which we need to import:

```
>>> from contracts import models
```

Look at this contract in the official database. It has the number "791452". Lets pick it up:

```
>>> contract = models.Contract.objects.get(base_id=791452)
```

Now, lets start with its price:

```
>>> print(contract.price)
```

and its description:

```
>>> print(contract.description)
```

Lets now pick the entity that made this contract. In this case there is only one, but in general there can be more: a contract has a ManyToMany relationship with entities because each contract can have several entities (a joint contract), but also each entity can have several contracts.

In fact, each contract has two ManyToMany: the entities that paid, and the entities that were paid.

Lets say we want the entity that paid this contract. In that case, we pick the set of all entities that paid, and select the first one:

```
>>> entity = contract.contractors.all()[0]
>>> print(entity)
```

Let's now pick the contracts that this entity made. To that, we use the "contracts_made" of the entity:

```
>>> entity_contracts = entity.contracts_made.all()
```

Lets count the number of these contracts:

```
>>> print(entity_contracts.count())
```

You can check that this number matches the number in the official database. (there can be small error when there were contracts added today; our database is synchronized in the end of the day).

Now, how much is the price of all those contracts?

To answer that, we have to aggregate the prices. The syntax in Django is the following:

```
>>> from django.db.models import Sum
>>> total_price = entity_contracts.aggregate(our_sum=Sum('price'))['our_sum']
>>> print(total_price)
```

Again, you can check on the official website.

As a final example, we are going to use a filter. Lets say you want all the above contracts, but restricted to prices higher than 10.000€. For this, we need to "filter" these contracts:

```
>>> expensive_contracts = entity_contracts.filter(price__gt=10000*100)
>>> print(expensive_contracts.count())
```

The "price__gt" means price (g)reater (t)han, and we multiply by 100 because `prices are in euro cents`.

For now, that's it. You now have the minimal knowledge to ask your own questions. In section *here* you can find references of all models.

Notes:

- The syntax we use here (e.g. "contracts_made") **is** Django API.
- If you don't know Django, you can look at the existing source code of the website and start from there.
- You can find the documentation of the API *here*.

If some problem occur, please add an issue so we can help you and improve these instructions.

CHAPTER 3

API Reference

This part of the documentation focus on the API for accessing and using the database. It documents what objects the database contains, and how you can interact with them.

Publics uses Django ORM to build, maintain and query a Postgres database. This has two immediate consequences:

1. If you don't know python/Django but you know SQL, you can access it remotely (see *Database*).
2. If you don't know SQL, but you know Python and/or Django, you can take advantage of this API (see *Asking questions to the database*)

Contract

This document provides the API references for the contracts in the database.

API

class `models.`**`Contract`**

A contract is a relationship between *entities* enrolled in the database. Each contract has a set of contractors (who paid), and contracted (who were paid), and relevant information about the contract.

All the fields of this model are retrieved from Base. Except for `base_id` and `added_date`, all entries can be null if they don't exist in Base. They are:

`description`
The description of the contract.

`price`
The price of the contract, in cents (to be an integer).

`category`
A Foreign key to the contract *Category*.

`contractors`
A ManyToMany relationship with *entities*. Related name "contracts_made".

contracted
A ManyToMany relationship with *entities*.

added_date
The date it was added to Base database.

signing_date
The date it was signed.

close_date
The date is was closed. It is normally null.

base_id
The primary key of the contract on the Base database. It is "unique" and can used to create the link to Base (see *get_base_url()*)

contract_type
A Foreign key to one of the types of contracts.

procedure_type
A Foreign key to one of the types of procedures.

contract_description
A text about the object of the contract (i.e. what was bought or sold).

country

district

council
Country, district, council.

This model has a getter:

get_base_url()
Returns the url of this entity in Base.

Entity

This document provides the API references for the entities in the database.

API

class models.**Entity**
An entity is any company or institution that is enrolled in the database that are related trough *contracts*.

All the fields of this model are directly retrieved from Base. They are:

name
The name of the entity.

base_id
The primary key of the entity on the Base database. It is "unique".

nif
The fiscal identification number of the entity.

country
The country it is registered. It may be "null" when there is no such information.

It has the following getters:

total_earned()
Returns the total earned, in € cents, a value stored in *EntityData*.

total_expended()
Returns the total expended, in € cents, a value stored in *EntityData*.

get_base_url()
Returns the url of this entity in Base.

get_absolute_url()
Returns the url of this entity on this website.

And the following setters:

compute_data()
Computes two aggregations and stores them in its EntityData:

1. the total value in € of the contracts where it is a contractor
2. the total value in € of the contracts where it is contracted

This method is used when the crawler updates new contracts.

class models.**EntityData**
Data of an entity that is not retrieved from Base, i.e, it is computed with existing data. It is a kind of cache, but more persistent. This may become a proper cache in future.

It has a OneToOne relationship with *Entity*.

As the following attributes:

total_earned
The money, in cents, the entity earned so far.

total_expended
The money, in cents, the entity expended so far.

Category

This document provides the API references for the categories of contracts in the database. See ../tools/cpvs_importer for how these categories are built.

API References

class models.**Category**
A category is a formal way to categorize public contracts within European Union. It is a tag assigned to a contract.

A category as an OneToMany relationship to *Contract*: each contract has one category, each category can have more than one contract. This relationship is thus defined in the contract model.

A category has the following attributes:

code
The CPVS code of the category.

description_en

description_pt
The official descriptions of the category in english and portuguese, respectively.

depth
The depth of the attribute on the tree.

And has the following methods:

get_children()
Returns all children categories, excluding itself.

get_ancestors()
Returns all ancestor categories, excluding itself.

get_absolute_url()
Returns the url of this category in the website.

contracts_count()
Counts the number of all contracts that belong to this category or any of its children.

contracts_price()
Sums the prices of all contracts that belong to this category or any of its children.

Legislature

This document provides the API references for the legislatures in the database.

API

class models.**Legislature**
A legislature is a time-interval between two elections.

All the fields of this model are retrieved from parliament website using a crawler.

number
The official number of the legislature.

date_start
The date corresponding to the beginning of the legislature.

date_end
The date corresponding to the end of the legislature. It can be null when is an ongoing legislature.

Deputy

This document provides the API references for the deputies in the database.

API

class models.**Deputy**
A deputy is a person that at some point was part of the parliament.

All the fields of this model are retrieved from parliament website using a crawler.

name
The name of the person.

birthday
Birthday of the person. May be null if it is not in the official database.

party

A foreign key to the party the deputy belongs. This is a cached version, the correct version is always obtained from the mandate the deputy is.

is_active
A bool telling if the deputy is active or not. This is a cached version, the correct value is always obtained from the last mandate the deputy is.

get_absolute_url()
Returns the url of this entity in parliament website.

update()
Updates the *party* and *is_active* using the deputies' last mandate information. This only has to be called when the deputy has a new mandate.

Mandate

This document provides the API references for the mandates in the database.

API

class models.**Mandate**
A mandate is a time-interval corresponding to a mandate in the parliament of a deputy. A mandate require always a district and a party.

All the fields of this model are retrieved from parliament website using a crawler.

deputy
The *models.Deputy* of the mandate.

legislature
The *models.Legislature* of the mandate.

party
The parlamentary group this mandate is respective to. A ForeignKey to *models.Party*.

district
The district this mandate is respective to. This is a ForeignKey to models.District.

The next two fields are required because some mandates end before the legislature ends.

date_start
The date corresponding to the beginning of the mandate.

date_end
The date corresponding to the end of the mandate. It can be null when is an ongoing legislature.

Party

This document provides the API references for the parties in the database.

API

class models.**Party**

A party, formally known as a Parlamentary Group, is required to have a mandate in the parliament. Here is just a category of the mandate.

All fields of this model are retrieved from parliament website using a crawler.

abbrev

The abbreviated name of the party.

get_absolute_url()

The url for its view on the website.

CHAPTER 4

Other tools

Crawler for Contracts and Tenders

This document explains how Base provides its data and how the crawler works.

Important: Please, take precautions on using the crawler as it can generate Denial of Service (DoS) to Base database. We provide remote access to our database to avoid that.

Important: Crawling Base from scratch takes more than 2 days as of Jan. 2014.

Base database

Base uses the following urls to expose its data

1. *Entity*: http://www.base.gov.pt/base2/rest/entidades/{[}base_id]
2. *Contract*: http://www.base.gov.pt/base2/rest/contratos/{[}base_id]
3. Tender: http://www.base.gov.pt/base2/rest/anuncios/{[}base_id]
4. List of Country: http://www.base.gov.pt/base2/rest/lista/paises
5. List of District: http://www.base.gov.pt/base2/rest/lista/distritos?pais={[}country_base_id]; (portugal_base_id=187)
6. List of Council: http://www.base.gov.pt/base2/rest/lista/concelhos?distrito={[}district_base_id];
7. List of ContractType: http://www.base.gov.pt/base2/rest/lista/tipocontratos
8. List of ProcedureType: http://www.base.gov.pt/base2/rest/lista/tipoprocedimentos

Each url returns json with information about the particular object. For this reason, we have an abstract crawler for retrieving this information and map it to this API.

What the crawler does

The crawler accesses Base urls using the same procedure for entities, contracts and tenders. It does the following:

1. retrieves the list `c1_ids=[i*10k, (i+1)*10k]` of ids from links 4., 5. or 6.;
2. retrieves all ids in range `[c_ids[0], c_ids[-1]]` from our db, `c2_ids`
3. Adds, using links 1.,2. or 3. all ids in `c1_ids` and not in `c2_ids`.
4. Removes, using links 1.,2. or 3. all ids in `c2_ids` and not in `c1_ids`.
5. Go to 1 with `i += 1` until it covers all contracts.

The initial value of `i` is 0 when the database is populated from scratch, and is such that only one cycle 1-5 is performed when searching for new items.

API references

This section introduces the different crawlers we use to crawl Base.

class `contracts.crawler.`**ContractsStaticDataCrawler**
A subclass `JSONCrawler` for static data. This crawler only needs to be run once and is used to populate the database the first time.

retrieve_and_save_all()
Retrieves and saves all static data of contracts.

Given the size of Base database, and since it is constantly being updated, contracts, entities and tenders, use the following approach:

class `contracts.crawler.`**DynamicCrawler**
An abstract subclass of `JSONCrawler` that implements the crawling procedure described in the previous section.

object_name = None
A string with the name of the object used to name the `.json` files; to be overwritten.

object_url = None
The url used to retrieve data from BASE; to be overwritten.

object_model = None
The model to be constructed from the retrieved data; to be overwritten.

static clean_data(*data*)
Cleans `data`, returning a `cleaned_data` dictionary with keys being fields of the `object_model` and values being extracted from `data`.

To be overwritten by subclasses.

save_instance(*cleaned_data*)
Saves or updates an instance of type `object_model` using the dictionary `cleaned_data`.

This method can be overwritten for changing how the instance is saved.

Returns a tuple `(instance, created)` where `created` is `True` if the instance was created (and not just updated).

update_instance(*base_id*)
Uses `get_json()`, *clean_data()* and *save_instance()* to create or update an instance identified by `base_id`.

Returns the output of *save_instance()*.

get_instances_count()
Returns the total number of existing instances in BASE db.

get_base_ids(*row1*, *row2*)
Returns a list of instances from BASE of length `row2 - row1`.

update_batch(*row1*, *row2*)
Updates a batch of rows, step 2.-4. of the previous section.

update(*start=0*, *end=None*, *items_per_batch=1000*)
The method retrieves count of all items in BASE (1 hit), and synchronizes items from *start* until *min(end, count)* in batches of *items_per_batch*.

If *end=None* (default), it retrieves until the last item.

if *start < 0*, the start is counted from the end.

Use e.g. *start=-2000* for a quick retrieve of new items;

Use *start=0* (default) to synchronize all items in database (it takes time!)

class contracts.crawler.**EntitiesCrawler**
A subclass of *DynamicCrawler* to populate *Entity* table.

Overwrites *clean_data()* to clean data to *Entity*.

Uses:

- `object_directory`: `'../../data/entities'`
- `object_name`: `'entity'`;
- `object_url`: `'http://www.base.gov.pt/base2/rest/entidades/%d'`
- `object_model`: *Entity*.

class contracts.crawler.**ContractsCrawler**
A subclass of *DynamicCrawler* to populate *Contract* table.

Overwrites *clean_data()* to clean data to *Contract* and *save_instance()* to also save `ManytoMany` relationships of the *Contract*.

Uses:

- `object_directory`: `'../../data/contracts'`
- `object_name`: `'contract'`;
- `object_url`: `'http://www.base.gov.pt/base2/rest/contratos/%d'`
- `object_model`: *Contract*.

class contracts.crawler.**TenderCrawler**
A subclass of *DynamicCrawler* to populate `Tender` table.

Overwrites *clean_data()* to clean data to `Tender` and *save_instance()* to also save `ManytoMany` relationships of the `Tender`.

Uses:

- `object_directory`: `'../../data/tenders'`
- `object_name`: `'tender'`;
- `object_url`: `'http://www.base.gov.pt/base2/rest/anuncios/%d'`
- `object_model`: `Tender`.

Crawler for Categories

Europe Union has a categorisation system for public contracts, CPVS, that translates a string of 8 digits into a category to be used in public contracts.

More than categories, this system is a tree with broader categories like "agriculture", and more specific ones like "potatos".

They provide the fixture as an XML file, which we import:

`contracts.categories_crawler.`**`build_categories`**`()`

Constructs the category tree of *`categories`*.

Gets the most general categories and saves then, repeating this recursively to more specific categories until it reaches the leaves of the tree.

The categories are retrieved from the internet.

Database

This document explains what is our database, how you can access it remotely, and how you can obtain a dump of it.

Database

Our database contains all the information the official databases contains. Additionally, our database is bound to a database ORM and web framework - Django - that facilitates its usage.

Either by using Django or not, you have full remote access (read only) to it.

Remote connection

You can access our database using:

- database: publics
- username: publics_read_only
- password: read-only
- host: 185.20.49.8
- port: 5432

Create dumps

In the terminal, run:

```
pg_dump -h 185.20.49.8 -p 5432 -U publics_read_only -d publics > dump.sql
```

This creates a file "dump.sql" (it can take a while) that you can load to a database in your own computer.

CHAPTER 5

Install development environment

This part of the documentation explains how you can jump to the development of publicos.pt and deploy the website on your computer. To only interact with the database, e.g. to do statistics, you only need to *install the API dependencies*.

We assume here that you know Python and a minimum of Django.

Dependencies for the website

Besides the *dependencies of the API*, the website uses the following packages:

BeautifulSoup 4

For crawling websites, we use a Python package to handle HTML elements. To install it, use:

```
pip install beautifulsoup4
```

django-debug-toolbar

To develop, we use `django-debug-toolbar`, an utility to debug Django websites:

```
pip install django-debug-toolbar
```

Running the website

Once you have the dependencies installed, you can run the website from the root directory using:

```
python manage.py runserver
```

and enter in the url `http://127.0.0.1:8000`.

If anything went wrong or you have any question, please drop by our mailing list so we can help you.

Running tests

We use standard Django unit test cases. To run tests, use:

```
python manage.py test <package, module, or function>
```

For instance, for running the test suite of contracts app, run:

```
python manage.py test contracts.tests
```

Running the crawler

To run the *Crawler for Contracts and Tenders* to populate the database, you require an additional package:

```
pip install requests
```

CHAPTER 6

Organization of the code

Django apps

The code is organized as a standard Django website composed of 4 apps:

- main: delivers `robots.txt`, the main page, the about page, etc;
- contracts
- deputies
- law

The rest of this section, `apps` refer to all aps except `main`. `main` is also a Django app, but does not share the common logic of the other apps.

All apps are Django-standard: they have `models.py`, `views.py`, `urls.py`, `templates`, `static`, `tests`.

Each app has a module called `<app>/crawler.py` that contains the crawler it uses to download the data from official sources. Each app has a `<app>/tasks .py` with django-rq jobs for running the app's crawler.

Besides a crawler, each app has a package `<app>/analysis`. This package contains a list of existing analysis. An analysis is just an expensive operation that is performed once a day (after data synchronization) and is cached for 24 hours.

Since `contracts` is a large app, its backend is sub-divided:

- views and urls modules are divided according to whom they refer to
- templates are divided into folders, according to the view they refer to.

Scheduling

We have a periodic job that runs in django-rq to synchronize our database with the official sources and update caches. It uses settings from `main/settings_for_schedule.py`.

CHAPTER 7

Contribute

This document explains how you can contribute to this project. See why in http://publicos.pt/contribui

Install the project for development environment

Deploy Python project

1. Create and start a new virtualenv:

```
virtualenv ~/.virtualenvs/publicos
source ~/.virtualenvs/publicos/bin/activate
```

2. Install dependencies:

```
pip install -r website_requirements.txt
```

Deploy database locally (Optional)

If you develop systematically, we recommend to install the database locally so your queries don't have to hit the production database.

1. Install and start postgres
2. Create a dump of the production database in your machine:

```
pg_dump -h 5.153.9.51 -p 5432 -U publics_read_only -d publics > dump.sql
```

This may take some minutes as it downloads the complete database to your computer.

2. Create a new user (say *publicos* and no pass) and a database (say *publicos_db*)
3. Fill the newly created database with the dump:

```
psql -U publicos -d publicos_db -f dump.sql
```

This also takes some time.

4. Create a file *main/settings_dev.py* and add the credentials of your database:

```
DATABASES = {
'default': {
    'ENGINE': 'django.db.backends.postgresql_psycopg2',
    'NAME': 'publicos',
    'USER': 'publicos',
    }
}
```

Enable searching (Optional)

If you plan to test or use search capabilities in your development environment, you need to enable searching, done via Sphinx:

1. Install Sphinx
2. Index your database:

```
python manage.py index_sphinx
```

3. Start Sphinx daemon:

```
python manage.py start_sphinx
```

(To stop it, call *python manage.py stop_sphinx*.)

Start your server

Start the server running:

```
python manage.py runserver
```

and open a browser in *127.0.0.1:8000*.

Ticket system

This project uses a ticket system for documenting issues.

The system's idea is to keep track of the issues the project has and by whom they are being addressed.

Given the size of the project, we prefer to use tickets for problems that require some time to be solved.

Adding tickets

We write tickets with clear, concise and specific information about issue we are addressing.

Working on tickets

Working on the ticket means the contributor is committing himself to modify to the source to fulfill the ticket expectations.

We give the chance to contributors - specially to new contributors - to go thought the full process: if the person who submitted the ticket is willing to work on it, he should have priority on doing it, even if this means having the ticket open for a longer time.

Closing tickets

When a ticket is fixed, normally by a commit or set of commits, it is closed as fixed. A ticket can be re-opened if there is no consensus that it was solved.

The Source Code

Code styling

1. Follow PEP 8
2. Capitalize the first letter of classes.
3. Import modules in the following order:
   (a) builtin modules
   (b) Django modules
   (c) other third party modules
   (d) project modules
   (e) app modules
4. Don't use:

```
from X import *
```

5. Comment and document directly the source code only when necessary to understand what it means within its context. The big picture is documented here.

Hint: Making the code clearer and better documented is a good way of start contributing to this project since you read code and try to make it more clear when you don't understand of it.

Documentation

A modification on the code that changes semantics of the project has to be accompanied by the respective change in documentation to be incorporated.

Hint: Improving the documentation is a good way to start contributing to the project, since you learn about the project while improving it.

Committing

When you feel that your changes provide value to the existing code, commit it.

Commit message

Use 72 characters limit to the first line of the commit message.

A commit is self contained: the first message explains what it does, the rest of the lines explain how and what changed specifically. The actual code changes of the commit support what the commit message claims.

When the commit fixes a given ticket, it must contain that information on the commit message. E.g.

Fixed #12 – Added support for i18n.

Pull requests

When you have a commit or set of commits that you fell are worth being incorporated (e.g. because they close a specific ticket), make a pull request to announce that you have value that can be added to the project.

requesting a pull

We prefer the GitHub way: push your local commits to your GitHub fork, and create a pull request from there.

The message of the pull request should be a message that explains that commit or set of commits.

Pull request review

The idea of the pull request is that you are notifying other contributors that you have a set of commits that are worth adding to the project.

As such, it is worth to have the pull request reviewed by other contributors before entering the project's source. The idea is that other persons can check what you did.

CHAPTER 8

Publics

This place documents the backend of our website.

This backend provides an interface to access to three distinct portuguese public databases:

- Public procurements (Base)
- MPs and parliament procedures (parliament)
- Law (law)

We build and maintain an open source website and API for querying these databases. Specifically, this project consists in three components:

- A database in postgres and driven by Django ORM, remotely accessible.
- An API for querying the database using Django and Python.
- A website for visualizing the database and sharing statistical features of it.

How can you use it?

To navigate in the database and discover some of its features, you can visit the website.

To use the API, e.g. to *ask your own questions to the database*, you must first *install it*.

To contribute to this API and/or website, see section *Install development environment*.

Index

N

P

R

S

T

U