

YAAS Project : Report

Timothée RABOEUF - 74764

28 oct. 2018

Contents

1	Introduction	2
1.1	Implemented requirements	2
1.2	Packages used	3
2	Development strategies	4
2.1	Confirmation form (UC3)	4
2.2	Automatic bid resolution	4
2.3	Concurrency management	5
2.4	REST API	5
2.5	Functional tests	6
2.6	Language switching management	6
2.7	Data generation	7
2.8	Deployment	7

Chapter 1

Introduction

This YAAS application is deployed on both <http://traboeuf.pythonanywhere.com> and <https://yaas.salicorne.ovh>. The sources are available at github.com/Salicorne/YAAS and github.com/it-teaching-abo-akademi/dt00bt67-server-side-web-services-project-201

1.1 Implemented requirements

The following list of requirements has been implemented :

- UC1: create user
- UC2: edit user
- UC3: create auction
- UC4: edit auction description
- UC6: bid
- UC5: Browse & Search
- UC7: ban auction
- UC8: resolve auction
- UC9: language switching
- UC10: concurrency
- UC11: currency exchange
- WS1: Browse & Search API
- WS2: Bid api
- OP2: soft deadlines for bidding
- TR1: data generation program

1.2 Packages used

This YAAS application relies on the following packages :

- django (2.1.1)
- django-crispy-forms (1.7.2)
- django-filter (2.0.0)
- djangorestframework (3.8.2)
- markdown (3.0.1)
- requests (2.20.0)

Chapter 2

Development strategies

2.1 Confirmation form (UC3)

UC3 states that the user should be asked for a confirmation before creating a new auction. I implemented it using the rendering of a confirmation form, containing all auction informations in hidden field. The first auction creation form is validated server-side : if all informations are correct, the confirmation form is shown with these informations. Only when this confirmation form is submitted the new auction is created and saved in database : thus if the user cancels the process or never submits the form the auction is not saved. This behaviour is described in the file `Auction/views.py`, in the functions `AuctionEditView::get`, `AuctionEditView::post` and `auctionConfirm`.

2.2 Automatic bid resolution

For the bids resolution, the problematic is to allow resolution without any web request being made. We can't use classic Django views, and need to find another method. To begin I had a look at the Django module `django_cron` but all it does is exposing a django command allowing to run jobs through `manage.py`, but still relies on an external tool (such as Unix crontab) to run this command. The module `django-extensions` has a jobs scheduling system, but it does not run jobs automatically either. I ended up manually launching a new thread, that will check all active (ie not banned and not resolved) auctions every minute and resolve them if needed. This thread is flagged as **daemon** : *the entire Python program exits when only daemon threads are left*.¹ This allows the program to run without having to manage this thread, and commands ran via `manage.py` to be non-blocking (as this thread will be launched during their execution).

This auctions resolution thread is launched in the Auction app initialization. This is achieved thanks to the `AuctionConfig` class, subclassing

¹<https://docs.python.org/3/library/threading.html#threading.Thread.daemon>

`AppConfig` in `Auction/apps.py`. This config class is declared in the `__init.py__` file of the Auction module. Beware : when using `python manage.py runserver`, Django starts two processes : one for the development web server and another process to automatically reload the application on code change. The Auction app is loaded in both processes, causing the auctions resolution thread to be launched twice. To avoid this behaviour, the development server should be launched with `python manage.py runserver --noreload` and manually restarted on any code change.

2.3 Concurrency management

This YAAS application relies on optimistic concurrency management : each auction has a `bid_version` attribute, and this version is checked when bidding. If the version is not up-to-date, the bidding process will be aborted and the user will be asked to refresh the page so as to see the up-to-date version. An auction version will be outdated when the description is updated, or when a bid is made by another user. It is important to keep in mind that this optimistic concurrency will not be 100% efficient if the application runs on several threads : if two users make a bid at the exact same time, there could be a CPU jump between the processing of the two requests, between the version validation and the actual bidding. We would have to implement a database locking mechanism in that case, but this is not supported by Django for SQLite.

2.4 REST API

The REST API is implemented using the *Django REST Framework*. It follows main REST principles : it relies on HTTP verbs (GET, POST), and is *stateless* : there is no state kept on the server, and each request is independant. The code is shared between the regular web app and the API : for a single action there is a function for the actual action, and two other englobing functions used to call this main function from the regular views and from the API. The naming convention is as follows :

- `get_function` : The actual code logic
- `api_function` : Used to call the function from the API
- `function` : Used to call the function from the web app.

For the endpoints needing authentication, HTTP *Basic authentication*² is used.

The following endpoints are available :

²<https://www.w3.org/Protocols/HTTP/1.0/spec.html#BasicAA>

- GET `/api/auctions` : to browse all available auctions.
- GET `/api/auctions/<int:id>` : to view the details of a specific auction.
- POST `/api/auctions/search?q=param` : to search auctions by title.
- POST `/api/bid/<int:id>` : to put a bid on a specific auction. The POST data must be a JSON object including the `bid_version` (which can be retrieved using GET `/api/auctions/<int:id>`) and the new price.

A concrete example to bid on the auction #331 for 98 euros with the account user, password user, knowing the current `bid_version` for this auction is 3 would be :

```
POST /api/bid/331 HTTP/1.1
Host: yaas.salicorne.ovh
Authorization: Basic dXNlcjp1c2Vy
Content-Type: application/json

{
    "version": 3,
    "price": 98
}
```

2.5 Functional tests

Functional tests have been created using the Django test module. Tests are written for UC3 (auction creation), UC6 (bid) and UC10 (concurrency), as well as some basic API calls (browse and get). These tests use both the regular web views and the REST API. 13 tests have been implemented, covering auction creation behaviour for registered and anonymous users, bidding process and its various constraints using both the API and the regular web views, and a full concurrency scenario.

2.6 Language switching management

Language switching is handled at the session level, in the view `Utils::change_language`. The requirement *OP3: store language preference* has not been implemented, but it could have been easily achieved by subclassing the default django User to add an optional *language* field, and by setting this language when the user logs in.

The localization module `l10n` has been disabled, as it had an impact on the default float delimiter (, in french instead of .) in the decimal fields, but this modification was not reported in the serialization process.

2.7 Data generation

The data generation program can be run using `GET /generatedata` as a registered or anonymous user. This program erases all bids, auctions and users, and populates the database with 51 users (`user/user`, `user1/user1`, `user2/user2` etc), 51 auctions and some bids.

2.8 Deployment

This YAAS application has been deployed on `pythonanywhere` as stated by the requirements, with the domain `traboeuf.pythonanywhere.com`. However, due to the lack of control of the execution environment and the various limitations of this website the application is also deployed on a production-ready Linux server, with the domain `yaas.salicorne.ovh`. This deployment is based on two containers : a Gunicorn application, running the Django code, and a nginx server serving static files. both containers are behind a Traefik reverse-proxy, responsible for requests routing and Let's Encrypt certificates management.