YAAS Project : Report

Timothée RABOEUF - 74764

28 oct. 2018

Contents

Chapter 1

Introduction

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
- 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 Session management

2.2 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 submited 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`.

2.3 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 `daemonic` : *the entire Python program exits when only daemon threads are left.* [1] This allows the program to run without worrying about this thread, and commands ran via `manage.py` to be non-blocking (as this

[1] https://docs.python.org/3/library/threading.html

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 `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.

2.4 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 concurency 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 mecanism in that case, but this is not supported by Django for SQLite.

2.5 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.

2.6 Functional tests

2.7 Language switching management

2.8 Data generation