TimelapseApp Documentation

Stefan Foulis

Sep 29, 2018

Contents

TimelapseApp will be a web application built to create timelapse videos. It is connected to cameras which can send images.

Warning: Nearly none of this actually exists yet. This are just my list of wild ideas.

CHAPTER 1

backend

- django with a GraphQL API
- django-celery for background tasks
- django-channels (in combination with relay subscriptions)
- moviepy to generate movies
- authentication using JWT

CHAPTER 2

ui

- node server
- react & relay (talks to graphql api of backend)
- google material design or bootstrap4

CHAPTER 3

cameras

- **RaspberryPi (Zero?)**
  - with a camera module
  - accessing a GoPro over the WebAPI
  - controlling a DSLR over usb
- Can run in online and offline mode
- delivers images to backend over wifi/4G/LAN

CHAPTER 4

Things to explore

The primary reason for this project is not actually not the awesome timelapse application. The main motivator for me is the explore solutions to challenges I've encoutered in other projects on a green field.

The main themes are:

- **Insight and feedback**: Easily see what is happening and why. For operations, developers and the end user.
- **Great client-server communication**: GraphQL + Relay all the way down.
- **Prevent race conditions**: Avoid running tasks that can't run at the same time from clashing.

For all these challenges I'd like to find coding conventions, existing packages and build new packages where necessary.

4.1 Logging and progress

Define a coding style that treats logging as a first class citizen.

eliot looks like a great candidate for structured logging. It allows adding context to the logs and allows signifying why it is running by passing along a log identifier. So a process may be started in a web request and continued in multiple celery tasks and because a unique log id was passed along the complete hierarchical log stream for a certain thing can be extracted again at the central logging system.

Built upon eliot we'd define a convention on how to express progress and heartbeat for any task (but especially long running tasks) through logging. That information can then be used in the UI to show progress indicators for users and for admins as a general overview of the system and what is going on in it.

django-channels could be used to send this information to clients. ELK could be used to store the logs. Kibana, or possibly a custom UI that understands eliot logs, could be used to explore the logs.

4.2 Locking, State and Race Conditions

Define a way to code that avoids race conditions, provides clear indicators of what state the objects in the database are in and provides clear paths to change the state. APIs to do this should be easy and treated as first class citizens.

django-fsm already provides a superb way of defining the different states an object can be in and how to transition between them. It also allows asking "what transitions are possible from the current state?". Besides making state clearer for devs this also gives us the opportunity to reduce complexity in the clients in some cases. Instead of the client having to know about all the possible rules for state transitions, it could build parts of the UI semi-automatically by asking the "what transitions are possible from the current state?" question. Additionally it would be really useful if we could find a way to answer the "why can't I transition to state x now?" or "what do I have to do to transition to state x?".

We'd define a some basic states that every object has, like "initialising", "ready", "deleting", "deleted". Then we can combine the finite state machine with locking. We'd have a standard way to aquire and release named locks for a given object. A easy way for sub-objects to inherit certain locks from a primary object. By default only one lock per object could be acquired at a time. But it would be possible to define exceptions of locks that can be acquired at the same time with others.

We'd also have a standard way of defining what should happen when a lock can't be aquired (try later, fail, ...) and a clear way for long running processes to continuously check whether they still have the lock and gracefully stop if they loose it. This could also make it possible to cancel processes over the UI if we change our mind. Coding conventions should encourage writing idempotent functions and easy ways to chunk long running functions into smaller parts and also save the progress. For example for a celery task that sends out 1000 emails, it should handle sending them out in small batches and record that progress. If the task is killed, dies or is cancelled the progress should be clear and it should be possible to resume it.

4.3 GraphQL, Relay and React

Through the use of GraphQL, Relay and React, as well as Relay Subscriptions with django-channels, we'd provide a stack that treats instant feedback and progress information as a first class citizen.

Using the metadata available from the apis mentioned above we can produce a really rich experience.

An unclear thing with GraphQL and Django (graphene-django) is how to handle validation and permission checking for mutations. We'd want to figure out a standard way of doing that (possibly using serializers from django rest framework).

4.4 Application Insight

For the developers and operations it is really useful to know what is going on in a system. Besides the benefits from logging, locks and progress mentioned above, it would be useful to have easy insight into task queues (celery). Some way of visually showing what is running right now, what is queued and what the backlog on every queue is. Also information like durations of each task than ran with min/max/average information over time. Allowing feedback like "this task is taking unusally long" and possibly even displaying that to the enduser.

We'd also log every request with url/statuscode/user/duration. The duration can later be used for stats by querying elasticsearch.

4.5 Fancy stuff

Image recognition:

- auto tag images by objects in the image (car, house, giraffe, moon, bird, person, ...)
- detect anomalies (like if an image probably is in the wrong stream)
- detect day / night

Auto add tags about weather based on position, time and historic weather data.

Auto add tags about sun/moon visibility based on position and direction of camera.