Vladislav Lysenkov

A prototype of the movie archive for research and publishing in structural biology

Bachelor's Thesis
Information Technology

May 2016

MAMK
University of Applied Sciences

DESCRIPTION

MAMK
University of Applied Sciences

Date of the bachelor's thesis

06.05.2015

Author(s)

Vladislav Lysenkov

Degree programme and option

Information Technology

Name of the bachelor's thesis

A prototype of the movie archive for research and publishing in structural biology

Abstract

The aim of this work was to pilot technologies that could lead to the foundation of an archive of animations visualizing dynamics and functional changes in molecular structures. The scientific context is important as an implication of specific premises and requirements that were to be satisfied.

The thesis involved the development of the web-based user interface for the display and management of videos and their further annotation with links to PDB and EMDB. This was done using technologies such the Django framework, the Popcorn.JS library and various APIs. The methods included the Agile SCRUM based iterative methodology, the user experience testing, and the version control by the means of SVN.

Stepwise, the project had grown into a functional prototype. The essential details, the process and the challenges are described. The user testing revealed the usability issues and general expectations.

In conclusion, this work had demonstrated the feasibility of the discussed movie archive and has the possibility of the further development.

Subject headings, (keywords)

Software development, Agile, Python, Django, JavaScript, User experience testing

Pages	**Language**	**URN**
47	English	

Remarks, notes on appendices

Tutor	**Bachelor's thesis assigned by**
Reijo Vuohelainen	European Bioinformatics Institute (EMBL-EBI) Workplace supervisor: Ardan Patwardhan

CONTENTS

1 INTRODUCTION

1.1 The EBI and its role in modern structural biology

The European Bioinformatics Institute, one of the outstations of the European Molecular Biology Laboratory stores, curates and distributes biological data. This data is comprehensive enough to be one of the main sources for significant biological research.

One of the goals of the organization is to provide openly accessed cloud services, tools and databases to support analysis and publication. The EBI consists of groups working on genomes and gene expression, protein sequences, molecular structures, chemical biology and others.

One of these groups, the PDBe or Protein Data Bank in Europe helps to collect, interpret and validate structural biology data, advancing expertise and developing and maintaining structure databases such as PDB and EMDB (Electron Microscopy Data Bank) and services on top of them such as PDBeFold, PDBeMotif or SIFTS. Most of them are to some extent integrated and connected with each other, as well as with other services from the EBI and associated institutions.

This integration might be one of the ultimate goals of bioinformatics as a science. It is very heterogeneous, yet to elicit the biological value from the data, it is always important to relate it to another source of information. "The key to bioinformatics is integration, integration, integration" (Chicurel 2002). The EBI thrives to fill the gaps and bring new solutions to the table. So does PDBe. Along with the tradition, this movie archive project demonstrates the feasibility of linking one kind of useful information to another.

1.2 Aims and objectives of the work, or the need for a movie archive

The idea of the movie archive was proposed by Helen Saibil in 2012, at an expert workshop "A 3D cellular context for the macromolecular world" that was held by PDBe. More precisely, it was stated that despite animations being "an excellent means to visualize dynamics and functionally important changes in molecular structures" (Patwardhan et al. 2014), journals where such animations are usually stored do not provide them in a sustainable and integrated way. Indeed, in the course of the development it was ascertained that supplementary materials in journals are far from consistency. Not only they have no metadata and are not referred to as standalone entries in any particular order, but the essentials like storage and presentation are lacking. Some of these movies are even confined in PDF documents (Clare et al. 2012).

Therefore it was decided to initiate a project that would meet this need and

a) cater for movies, maintaining them in a controlled manner for public access
b) use YouTube or another movie channel as a video backend whenever possible
c) capture annotations and references linking the movie to relevant EMDB and PDB entries, since animations are based on experimental structures from PDB and EMDB.

Thus, the aim of this project was to pilot technologies that could be used to develop such movie/animation archive, by the means of web-based user interface for uploading, annotating and validating the movies.

1.3 Structure of the work/study

This document is divided according to the stages of development.

The first chapter describes the preamble of the project: where it was originated, how does it fit in and why.

The second chapter describes objectives and challenges that had to be thought through and solved, as well as the initial steps and foundations that had to be taken and laid in order to proceed productively.

The third chapter describes the actual implementation, starting with the basis of the project, the Catalog.

The fourth chapter continues to describe the actual implementation, touching upon a proper display of movie annotations, an essential distinctive feature of the project.

The fifth chapter describes another important and the longest stage, the integration of project as one of the PDBe services.

The sixth chapter describes everything with the user perspective.

In the seventh and final chapter, the conclusion is made.

2 AN OVERVIEW OF TECHNOLOGIES, SETUP, AND METHODS

2.1 The architecture of the movie archive

There are four essential parts that constitute the movie archive project.

First, the part called the Catalog represents the medium for management and representation of certain objects and their metadata, in our case – of the movies visualizing molecular dynamics. The meaning behind these objects is semi-important: one needs to understand the relevance of given data to not go off the rails and to understand the needs of an end user. This is further discussed in the *Testing and Completion* chapter. Other than that, the task is technical. The Catalog must be further divided into the means of assembling, arranging and accessing the index of its entries and the means of interacting with suitable video hosting services where movies can be collected and stored.

Due to the nature of the project, the data (i.e. movies) is supposed to be stored at a reliable source. Thus the movie archive should be able to provide access to the requested data with high availability and properly answer the privacy issues that would inevitably arise at current state of the typical publishing workflow. These characteristics can be set as very basic requirements, although the actual implementation uncovered that there is more to be desired. Hence, a developer-friendly API and mutual compatibility with external libraries represent advanced requirements.

Second is the interface for annotating movies showing molecular dynamics – in other words, stressing the context of any particular movie. As shown in Figure 1, the user should be able to interact with the video seeing the different states timely highlighted.

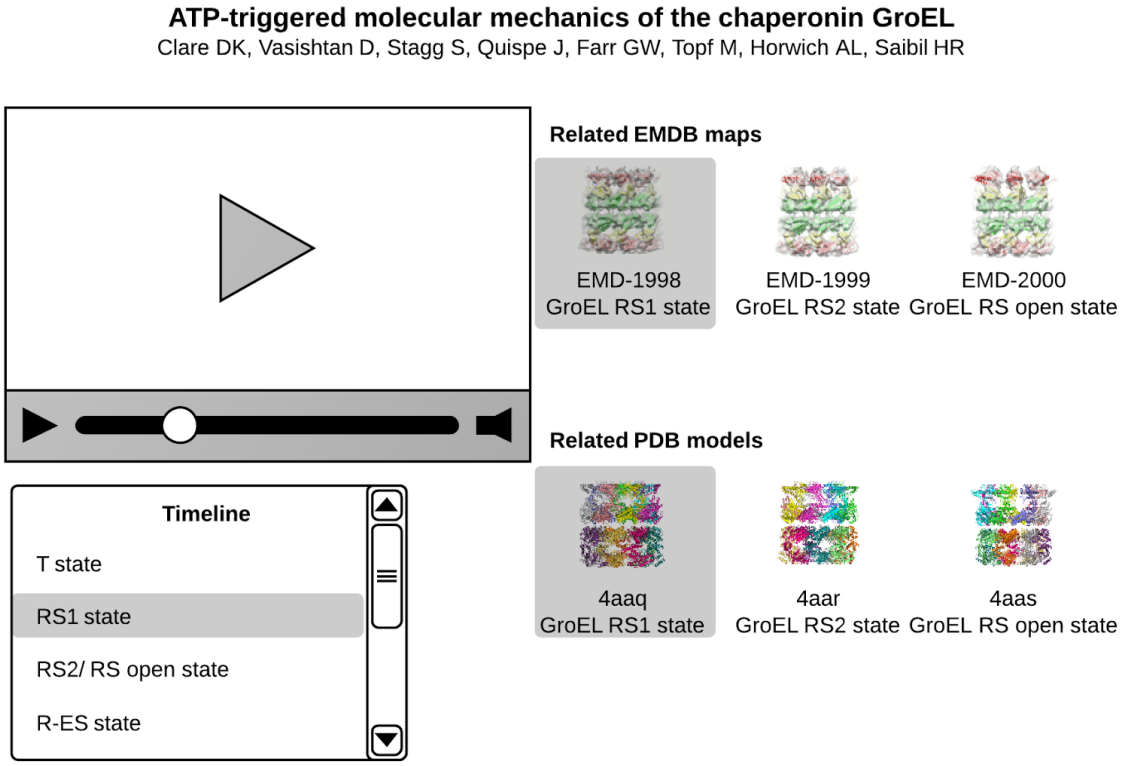

Figure 1. The initial mock-up

Third, the proposed web service is currently a "child" of another project, Electron Microscopy Pilot Image Archive (EMPIAR). The latter, being an operating archive for raw 2D image data related to the EMDB structures, has a profound Deposition & Annotation workflow and an authorization system. The wise thing to do was to use and adapt these systems rather than implement them from the scratch. Expectedly, there might arise a further need for a common authorization across PDBe projects.

Fourth and finally, the user interface is finalizing this work.

Altogether they form a system allowing to compose, modify, review and view movie entries, which in turn consist of video files, annotations to them, and range of references.

2.2 The Python language, the Django framework and their packages

Since the main and parent project is written in Django, inducing the development of any new project that don't have contradicting is pursued along the way. It is unknown whether any comparison with other solutions (like Ruby on Rails or Java under Grails) has been made at PDBe. It is evident though that Python, a general-purpose language used as a foundation of Django, is vastly popular in scientific applications. One could name a number of reasons behind it but Python is, above all, well supported by the community and therefore gives access to tons of specific libraries such as SciPy and NumPy.

The Django framework, as well as any other framework, is basically a collection of libraries. Django is often described as a "powerful framework" for web applications; however it does have limitations and is more of a "convenient framework". According to its history, it was devised as an answer to a growing demand for adding new features at several content-driven sites in a timely manner (Holovaty and Kaplan-Moss 2009). It does its job – having Python for its operation and loosely using the MVC (Model-View-Controller) design pattern, – the Django framework allows clear, quick assembly and integration of most of given projects.

In essence, Django makes the life of a developer easier, abstracting from repetitive tasks and distinguishing between the template and the context layers. It should be noted that while generally following the MVC pattern, Django provides its own naming for its parts. To eliminate confusion, it is helpful to elaborate on both. In terms of MVC,

- *Model* represents a module that serves as mediator between the database and other modules, describing the structures of database tables, providing mechanics of access and control, filtering and sorting and so on;

- *View* represents a module that performs data input and output by means of the user interface;

- *Controller* represents a module that processes the data. The *Controller* is calling either the *Model* or the *View* when it needs to address the database or the user correspondingly.

In terms of Django, however, *View* has a different interpretation, hence Django is rather an MTV framework:

- *Model*, again, represents a data access module;

- *Template* represents a module that is generally called *View* and is responsible for *how* the data appears. Usually it is a web page written in HTML with the addition of special tags stating where and in what format the given data should be put;

- *View* represents a module that is responsible for determining *which* data appears. Thus, the *View* processes the data just as the *Controller* does, but along with the Django framework itself and its service modules such as callback to the predefined URL.

In addition, Django allows concepts like ModelForm (a built-in shortcut for creating forms) or using processing logic inside *Models*. This behavior is made upon the flexibility of Python that sees each module as containing separate definitions and statements and importing them from other modules. See Figure 2 for further details.

Further in this document, only Django MTV definitions of *Models*, *Views* and *Templates* are used.

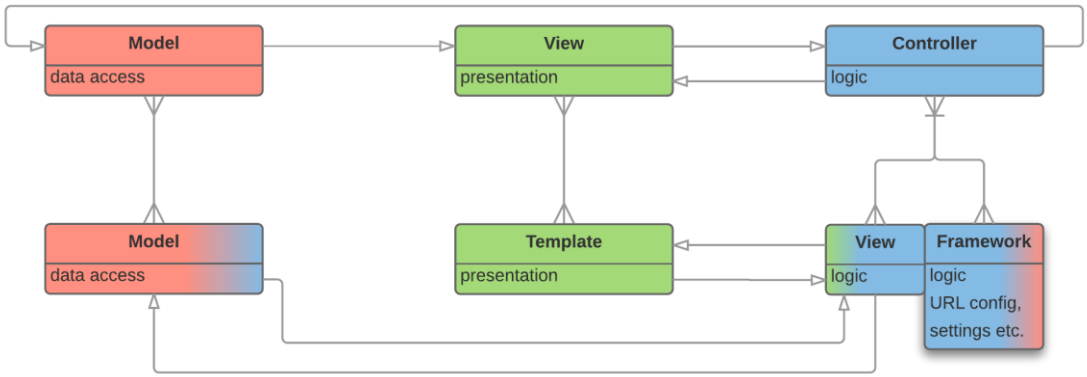

Figure 2. Diagram of the Model-View-Controller and Model-Template-View patterns.

The modularity of Python implies that virtually any Django project is naturally compartmentalized and likely to use external packages (libraries). For reference, in terms of Python a package is a group of modules, or a module that contains submodules under a common namespace, while a library (a module distribution) is package that is subjected to be published and imported en masse. This is done in accordance with Python and Django philosophies (The Zen of Python and DRY correspondingly) and in a general attempt to prevent the reinvention of the wheel.

For example, if one would like to do anything with URL, it would be sensible to import already implemented package *urllib* (*urllib2*, *urllib3* according to the exact needs) rather than to implement the functionality yourself.

It should be noted that two major revisions of Python (Python 2 and Python 3) do not have backward compatibility. Python 3 was designed with the intention to reduce the redundancy of the previous version, conforming to the main philosophy. The changes included, for instance,

- removed support for old-style classes,
- print function (instead of a statement),
- renamed functions,
- extended Integer division,
- altered dictionaries,
- Unicode for all strings.

Understandably, all this caused substantial difficulties in the translation from Python 2 to Python 3. As a result, a sufficient amount of libraries that were written in the era of Python 2 have not been ported to Python 3. This implies the use of Python 2 in many projects, including this one.

The Django framework itself, despite the laboriousness of the task, has started to officially support Python 3 from the version of Django 1.5. The community endorses developers to use compatibility library (*django.utils.six*) and write Python 3 code.

2.3 The development and testing environment

2.3.1 Setting up the workstations and the prerequisite modules

At the workplace, it was advantageous to operate different development environments. The term *development environment* in this context is not to be misinterpret for IDE (Integrated Development Environment), which is a type of software application and discussed in the *Version control* section. The term *development environment* stands for an instance with a specific installation and configuration of software products and their dependencies.

At first, I was given a Red Hat Enterprise Linux machine supported by EBI's Systems department. It didn't have root rights and the development was to an extent constrained – particularly, old versions of Python (2.6) and Django (1.4) were impedimental. For a supplement of the web server with up-to-date software, an Amazon EC2 instance was used over SSH connection.

Later I was given a MacBook with OS X El Capitan installed. This version of the OS has novel security feature called *System Integrity Protection (SIP)* that stood in the way, particularly forbidding to install *gdata* package for Python. It was impossible to do this manually either unless booting in the recovery mode and turning off *SIP*. However, since the laptop was supported by EBI's Systems, another solution had to be found. In the end I have placed two modules in the package (*apiclient* and *oauth2client*) in the project directory and the rest could be installed properly, allowing Python *distutils* system package manager to catch them up automatically.

It is worth mentioning that alongside with Django itself, the (abridged) list of imported packages for this project included:

- *httplib/httplib2, urllib/urllib2, urlparse, json* for making corresponding requests that would otherwise be on the sketchy side;

- *gdata 2.0.18* / *google-api-python-client 1.4.2* for interaction with the YouTube API (V2/V3);

- *PyVimeo 0.3.3* for interaction with the Vimeo API;

- *django-embed-video 1.1.0* for a facile embedding of uploaded YouTube and Vimeo videos. It was later proved to be an overkill for the task, but left for any future use;

- *mysql-python 1.2.5* that is an Python interface for MySQL database server.

- *django-sslserver 0.19, Pillow 2.9.0* (fork of *PIL*), *psutil, postgres, psycopg2, NumPy 1.9.2, python-dateutil 2.4.2, SciPy 0.16.0b2* for integration purposes.

At the time this work was underway, the PDBe group was discussing transition to a more up-to-date, stable and supported version of Django going by the number 1.8.x. This was proposed a sufficient time ago, but Systems were reluctant of securing the move because it would require a fair amount of time and effort not only for Systems, but for all groups with relevant services.

The development of the movie archive was done entirely with Django 1.8.2 version instead of 1.4.x that was released in early 2012 and running on most of the available servers and workstations. A specific setup (basically a virtual environment) at the development and productions servers was required and the parent project EMPIAR took the lead in piloting the update. It is closely described in the next section.

2.3.2 Web server infrastructure and the Anaconda environment

The IT infrastructure at the EBI is inherently complex as it comprises several datacenters in England. However, at the web production layer only VMs (Virtual Machines) need to be considered. Without going into details, there are naming schemes associated with VMs, utilizing 3-tier model of deployment – *development*, *staging* and *production* machines. They are used to provide access to the EBI data resources and services over the internet. It should be noted that this concerns only web application

server services. The VMs rely on storage provided over NFS and SAN where web applications are actually stored alongside with the data.

To be externally visible on the internet, or moreover, to be visible in a secure and scalable way, these VMs are running web application servers including Apache HTTP Server. Following the LAMP philosophy, they are not monolithic like the built-in Django web server (i.e. not containing all functions in one process) and are able to handle the requests when the resources are decentralized.

The Apache HTTP Server is a general-purpose http server. Its main purpose is to serve static content, but with the support of various modules it can serve dynamic content as well. The modules can enhance the functionality of the web server, for instance, aiming for support of different programming languages and IDEs or fixes, improvements and modifications. The modules are usually compiled with given environment and loaded into the Apache HTTP Server configuration file, but can be customized too. In our case, the *mod_wsgi* module is designed specifically for Python applications and in conjunction with Django is able to generate the content when required (i.e. dynamically).

This is how it works. The client sends HTTP requests to the server, where they are forwarded to the Apache HTTP Server and its *mod_wsgi* module consequently. The *mod_wsgi* together with Django serve as an application server that executes the logic to generate responses that would return to the client. In the meantime, the Django app connects to the database which may or may not be physically on the same server. The requests for static content are served directly from storage the web server has access to. Thus, the Apache HTTP Server acts as a medium between the client and the Django application. See Figure 3 for further details.

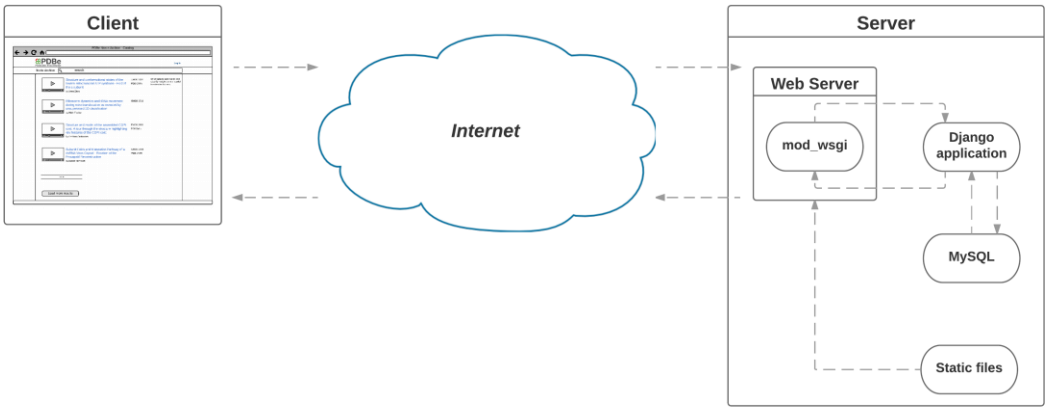

Figure 3. The diagram of the web server

The environment for running Django 1.8 in the VM, however, relies on an independent configuration. Since Systems were not going to make the general update, a separate and secure environment had to be introduced. The changes were done by means of Anaconda environment that was installed using the Conda package management system. Anaconda is actually built upon and includes Conda, along with Python itself and many more open-source Python, C, R, and Scala scientific packages with their dependencies. Conda creates environments simply hard-linking isolated installations of everything required (including all dependencies), making it possible to operate multiple environments by activating (changing the path to their binaries) ones when needed and even with lack of administrative privileges (root rights).

Down to the web server, the *mod_wsgi* module for Django 1.8 has to be replaced with the one modified and compiled with Anaconda Python 2.7 / Django 1.8 environment. From the technical perspective, the pages are generated by the server anew with each request. This may cause a significant overhead with large number of requests, which is why it is common practice to use load-balancing and caching for high-loaded web services. However since the EBI is rarely accessed by thousands hits simultaneously and scientific services may require non-caching technologies, this is not the case with the described web server.

2.4 Popcorn.JS – what it is and what it is for

2.4.1 The Popcorn library

In 2010, Mozilla jointly with Bocoup and CDOT developed an HTML5 video JavaScript library allowing embedding or adjoining web elements to the video. Back then, it was a notable example of using modern web technologies in interactive media, and was proposed as its future (Merkley 2012).

It is still alive, despite that the community is idle. The library has various plugins allowing connections to various sources from video services to Wikipedia. It is possible to use it to add free text, links and images alongside or over the video player.

The library was used in this project to implement a basic annotation interface for the movie archive.

2.4.2 Popcorn Maker

The Mozilla Popcorn Maker project was made on the top of the Popcorn.JS library. It was a web application that had been helping to create and share the media empowered with Popcorn.JS. It had its moments, but was closed not long before the movie archive project was started. It is no longer supported, because "it’s a matter of resources" (Mozilla Foundation 2015). However, since the Popcorn Maker project was open-source, the Wikimedia Foundation had planned to turn it into a collaborative video editor (Vibber 2015).

Picture 1. A screenshot of Popcorn Editor demo made by Wikimedia Foundation.

Popcorn Maker had undergone several major revisions and the last were powered by the Node.js JavaScript platform, a comprehensive solution for server environment. It would be sensible to try to reproduce it locally with integration in the existing system, but this is beyond the described project.

2.5 Version control

Version control, or revision control, is a necessary technique in large projects with multiple participants, because it allows to track and synchronize committed changes and generally benefits the team work.

The movie archive project is supposed to be only a part, or rather a child of the EMDB web service. Hence, continuing with team work, it is essential to use the same VCS (Version Control System), in our case – Subversion (SVN) for stated purposes. SVN is a free VCS that makes use of the central repository, extending it to working branches with their local copies.

In order to interact with SVN, it is possible to use a command line (e.g. from the server), a standalone software application such as RapidSVN, or a plugin for the IDE, such as Subversive plugin for the Eclipse IDE.

2.6 Management in general, Agile and SCRUM

At the workplace, it was customary to use the SCRUM methodology. Its purpose is the efficient team work, so the pronoun *we* should be more appropriate here. It is helpful in keeping the discipline and preventing the participants from tilting at windmills.

The methodology consists of three parts: the plan, the reports, and the retrospective. A cycle of development is called a Sprint, each of them having its own schedule but generally not more than four weeks. We had a Sprint planning in the start of each Sprint, where we had divided our goals for the sprint into relatively small tasks no longer than several days.

Then every day we made a short stand-up report of what was done on the previous day and what was intended to be done next. After the Sprint, we gave short presentations called Sprint reviews, followed by Sprint retrospectives where we discussed what was done and what weren't. The development of this project took three Sprints overall.

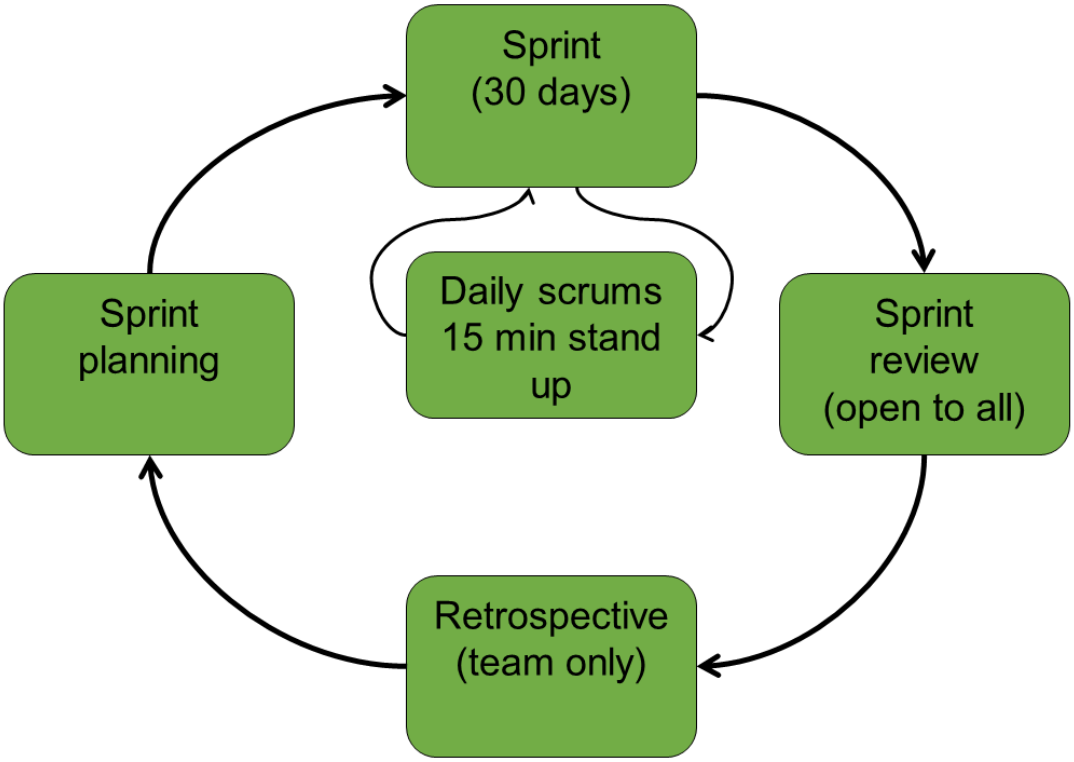

Figure 4. Overall process of the Sprint

3 THE CATALOG AND ITS CONTENT

3.1 The foundation of the Django project and the main app

The Django project is considered a Python package. It can contain several apps, which are essentially packages too and can in turn contain their modules. They are being imported in Django when the server is started. The modules hierarchically form the structure of the project. After the *django-admin startproject* and *startapp* commands have been executed, the structure is as follows:

```
– movie_archive/ – the root of the project
  – movie_archive/ – the root of the project package (source code)
  – __init__.py – these files are package–indicators for Python
  – manage.py – the management interface to the Django framework
  – settings.py
  – urls.py
  – app1/
      – __init__.py
      – admin.py
      – forms.py
      – models.py
      – tests.py
      – views.py
      – templates
          – index.html
  – app2/
  ...
  – upload/
  – templates/ – the root of the base templates
      – base.html
  – static/ – the root of the static content
  – media/ – mediafiles that are created or uploaded, e.g. movies
```

Code 1: The names of the unsigned modules should be self-explanatory

The actual hierarchy and nesting may seem redundant, but are important for three reasons: the Python 2 packages originate from filesystem, the Django apps should have a single location, and also it is just human-readable.

Verbatim, a catalog is a register, a simple way of organizing things. The fundamental job of a catalog is to contain, query, structure and categorize its content.

Conformably, the information concerning any particular object is stored as values of its attributes. All attributes are written in *models* and passed and processed in *views* according to their purpose. Therefore, the catalog represents all models and views needed for the movie archive to exist in this capacity.

An end user would work with an orderly array of the relevant data, queried from the centralized database. Interestingly, an end user itself would have their rights belong to this database, since there is a need to distinguish between them and to regulate the use of the objects. This is further described in the *Integration* chapter.

Consequently, an end user is given the possibility to see a limited amount of attributes and do a limited amount of actions upon them. For example, a list of open entries, with each one of them having its own description, thumbnail and title, or a list of the entries deposited by user, which is further described in the *Integration* chapter too.

In its *settings*, the Django project is set to connect to the MySQL database where the entries with their relations and values of their attributes are stored. The app constituting this project is historically called *Upload*, referring to the first function implemented during the project development – the upload of the movie to the YouTube.

3.2 The manipulation and the display of the content

This section describes selected views and models that are responsible for the CRUD operations. They are intentionally changed (truncated) from their final version to introduce the functionality gradually.

The above-mentioned CRUD, or *Create, Read, Update, Delete* concept represents the fundamental content management actions. In our case, this applies both to the database and to the interface – because using the Django framework implies the so-called *separation of presentation and content* design philosophy. [No reference here, this is a logical statement.] From the interface perspective, however, we are using the request

methods of the HTTP protocol, that is, we are using the REST (REpresentational State Transfer) design pattern that applies to web applications.

To start with, in order to *create* anything with user input, one needs to go through 5 basic steps. First, it is essential to have a *model* through which the Django would create a corresponding table to write this input to.

```
1  class Vid(models.Model):
2      title = models.CharField(max_length=200)
3      description = models.TextField()
```

Code 2: models.py sample

The second step of this abridged MTV interaction would be a *form*. The Django framework has a helper *ModelForm* class to simplify the creation of it.

```
1  class VidForm(forms.ModelForm):
2          class Meta:
3                  model = Entry
4                  fields = ['title', 'description']
```

Code 3: forms.py sample

That is, considering that the called *model* exists and has the corresponding fields. The final version of the project includes more complex customized forms, however at the stage of constructing of the Catalog this would be only bells and whistles. The *form* is required for the view to pass it to the template. It is common practice to address the *form* as either valid or not valid. But before that, we need to use POST request to obtain the data. This might already give an idea on how to implement the update on the data.

```
1  def create(request):
2          context = RequestContext(request)
3          if request.method == 'POST':
4                  form = VidForm(request.POST, request.FILES)
5                  if form.is_valid():
6                          form.save(commit=True)
7                          # The placeholder
8                          # form.save() for the placeholder
9                          return HttpResponseRedirect('/moviearchive')
10                 else:
11                         messages.error(request, 'Try again.')
```

```
12          else :
13                  form = VidForm()
14          args = {}
15          args.update( csrf( request ))
16          args['form'] = form
17          return render_to_response ('create_vid.html', args, context)
```

Code 4: views.py sample, create view

In this *Code 4*, the placeholder represents the additional logic or any changes in the model that are not directly influenced by the input data. For example, a call to the YouTube API function or another request. For an end user to access it, the *form* should be located at the specific URL. All possible URLs including the ones used only by the app's logic (i.e. not accessible in a form of a link in the UI) are specified in the urls module. It is illustrated in *Code 5*.

```
1 from django.conf.urls import patterns, include, url
2 urlpatterns = patterns (''',
3         url(r'^get/(?P<vid_id>\d+)/$', 'upload.views.view_vid'),
4         url(r'^create/$', 'upload.views.create'),
5 )
```

Code 5: urls.py sample, create and view-vid url patterns

After the *view* saves the *form*, the data in the database changes according to the input. The Update is done similarly, although it generates the page with the requested con-tent from the database first. The Read is straightforward and can be done in one line; the Delete makes use of the DELETE request method. All introduced CRUD opera-tions were used repeatedly over the course of development.

4 THE CATALOG AND ITS CONTENT

When it comes to professional video hosting services, there aren't many alternatives on the market. Certainly, there are dozens of video distribution websites, but much less can comply with our basic requirements, to say nothing of the advanced ones (see section 2.1). As an example, Popcorn.JS library provides media wrappers only for YouTube, Vimeo and SoundCloud services. It is possible to create a custom plugin or use the library separately from the media (further elaborated on in section 4.4), but it was decided to give the big players a try first.

4.1 Working with the YouTube API

The YouTube satisfies the basics requirements for video hosting services that were defined in the section *2.1*. The *Creator Studio*, a user interface for an end user, is clear and straightforward. It allows to upload videos, edit their metadata, add subtitles and see basic analytics and even provides plain video editor. Moreover, the YouTube has a function called *Annotations* which resembles the mock-up that was made prior to this investigation (see Figure 1, Figure 6).

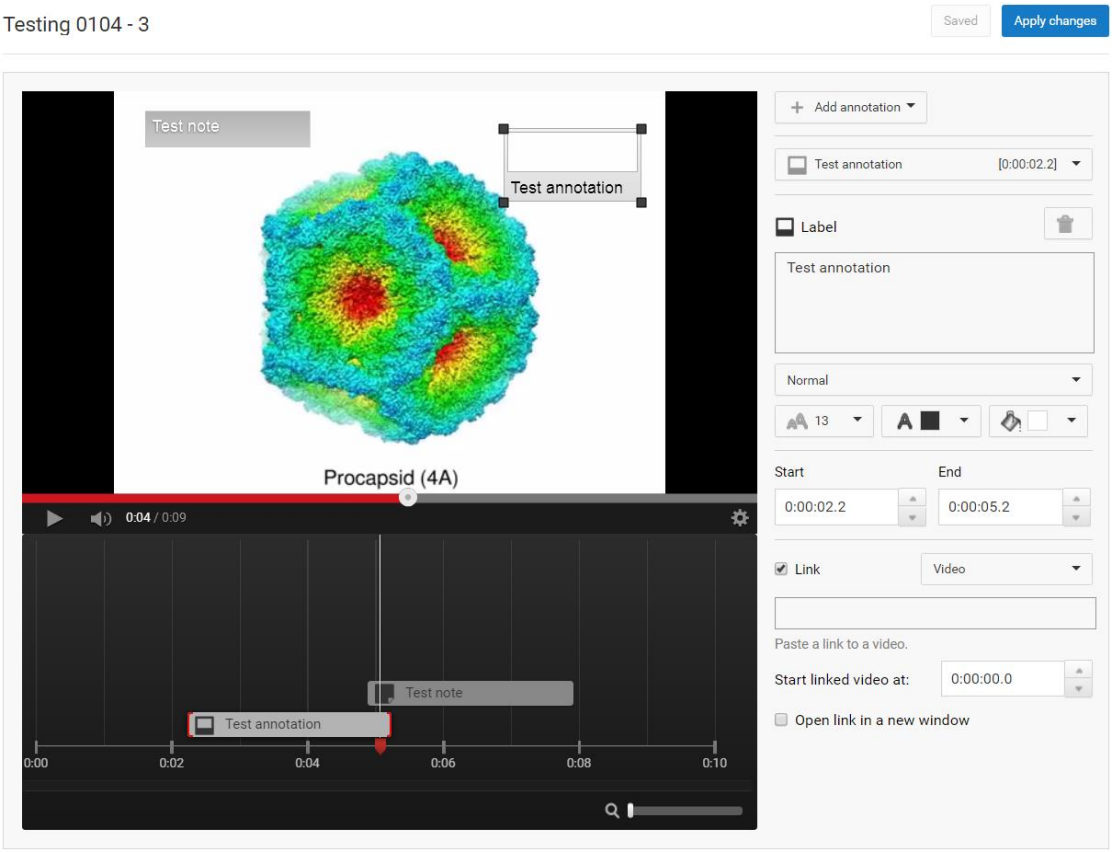

Figure 6. The YouTube Annotation editor.

These built-in *Annotations* could become a nice tool for the linkage between movies and corresponding PDB/EMDB entries, if only they weren't born limited. It is stated that "you can only add annotations to your videos on your computer" (YouTube help pages 2016) meaning that one needs to login to the YouTube website and use *Creator Studio* to proceed with this action. This contradicts with the proposed idea of the movie archive project: the movies should be managed from the EBI-maintained user interface, allowing

- manifold uploads without obtaining YouTube credentials for every user;
- additional features such as the above-mentioned linkage.

This could be performed only via the API (Application Programming Interface) of the YouTube video service. Unfortunately, the API is unsuitable for the YouTube *Annotations* despite that the community is asking for it (Google Project Hosting 2008).

Other than that, the YouTube API might seem even excessive, though well-documented. Since its temp of development is pushed by Google, it is likely that the current revision will become deprecated in a couple of years. This project uses the latest (V3) version at moment.

The API requires the user to authenticate prior to making any data requests. Consequently, we needed to obtain proper authorization credentials. This is done in five steps in Google Developers Console (*GDC*):

1. In the Project section of the *GDC*, create the project and see an Easter egg after choosing *test project* for the name.

2. In the API Manager section of the *GDC*, enable YouTube Data API v3. This is done in the *Overview* section with the unambiguous list of APIs. If this is not done, the next step will prompt back.

3. In this project, we are using OAuth 2.0 credentials. Therefore in the *Credentials* section of the *GDC*, create OAuth client ID with the application type being either *Web application* or *Other*. For the purpose of non-redundancy, the *Other* client is used. The system will generate a JSON that can be downloaded.

4. In the Python script or module with the desired functionality (e.g. uploading or updating the video) it is essential to query this JSON and set the relevant scope stating the level of access. This is the reason for having separate almost identical authentication functions in this project.

5. The initial execution of the desired functionality is most conveniently done from the IDE or Python interpreter. With OAuth client ID, the *GDC* prompts the developer to accept terms in the browser and then switch back to the application. Which in turn should generate another JSON with the access token for the relevant scope. In terms of Django, all JSON files fit well in the *static* directory.

The views of the movie archive project import and repeatedly call YouTube API functions. The calls with their arguments are prudently accommodated in place of the placeholder comment, as seen in *Code 6* that is an extension of *Code 4*. Global variables such as the one from class *EMDBGlobal* can be used in order to return the value into the view.

```
1  def create(request):
2          ...
3          if form.is_valid():
4                  uploaded_vid = form.save(commit=True)
5                  # send this file to youtube
6                  youtube = get_authenticated_service(uploaded_vid)
7                  try:
8                          initialize_upload(youtube, uploaded_vid)
9                  except HttpError:
10                         messages.error(request, 'HTTP_error')
11                 messages.success(request, 'Video_saved.')
12                 uploaded_vid.youtube_url = "https://www.youtube.com/watch?v
                       =" + EMDBGlobal.youtube_response_id
13                 form.save() # for the placeholder
14         ...
```

Code 6: views.py sample, create view, calling YouTube API functions

```
1  def initialize_upload(youtube, uploaded_vid):
2          body=dict(
3                  snippet=dict(
4                   title=uploaded_vid.title,
5                   description=uploaded_vid.descr,
6                   categoryId=uploaded_vid.category
7                   ),
8                   status=dict(privacyStatus=uploaded_vid.privacyStatus)
9          )
10          insert_request = youtube.videos().insert(
11                 part=",".join(body.keys()),
12                 body=body,
13                 media_body=MediaFileUpload(uploaded_vid.file, chunksize=-1,
                       resumable=True)
14         )
15         resumable_upload(insert_request)
```

Code 7: util-youtube.py sample, initialize-upload method

```
1  def resumable_upload( insert_request ):
2          ...
3          while response is None:
4                  try:
5                          status , response = in- sert_request .next_chunk()
6                          if 'id' in response:
7                                  EMDBGlobal.youtube_response_id = response['id
                                       ']
8                  except:
9                          messages. error ( request , ' Retriable error ' )
10                         retry += 1
11                         ...
```

Code 8: util-youtube.py sample, resumable-upload method

Consequently, the *initialize-upload* function takes authentication and saved arguments and calls the YouTube API insert method to upload the video in chunks. It is then passed to the *resumable-upload* method that checks the response status of the upload and saves the global variable with the successful result. See *Code 7* and *Code 8* for further details. To display the uploaded content, we need to pass acquired attributes to the template.

Inside the template the *Django template language* is used. It is fairly simple: whatever arguments are rendered in the *view* can be addressed at the *template* level. To let the *model* be as concise as possible, it was decided to import *django-embed-video* app that allows easy embedding of YouTube (and Vimeo) videos in the Django templates. The *Code 9* illustrates the whole concept. As a result, the movie is displayed in a standard

iframe element with player controls that is provided by the YouTube itself. However, further investigation reveals that utilization of the YouTube as a video backend for the movie archive raises a few is-sues. First and foremost, it can degrade quality of the uploaded video. It does a good job at being omnivorous, but is apparently focused on standard aspect ratios, color schemes and codecs. That being said, scientific videos such as discussed molecular dynamics animations are usually produced with professional software like UCSF Chimera with particular, sometimes peculiar configurations. These videos can easily be, for instance, vertical and generally not strictly following any widely accepted signal and resolution formats. Not surprisingly, when the YouTube attempts to adjust the uploaded video to accommodate it within allowed resolution mode, it is a rare occasion for that video to be available in full detail. Moreover, most videos after processing obtain black stripes. With a few CSS tricks, however, the responsive

video container can be achieved for an embedded YouTube video.

```
1  # views.py
2  def view_vid(request, vid_id=1):
3          return render_to_response('vid.html', {'vid': Vid.objects.get(id=
               vid_id)}, context_instance=RequestContext(request))
4
5  # vid.html
6  <div>
7  {% if vid %}
8          {% video vid.youtube_url query="rel=0" as my_video %}
9                  {% video my_video "medium" %}
10         {% endvideo %}
11 {% endif %}
12 </div>
```

Code 9: Displaying the embedded video using django-embed-video and a line from the corresponding view. The indentation in the HTML file is for readability

Apart from that, the YouTube API is changing fast enough to be a hindrance to slowly developed libraries like Popcorn.JS. For instance, the *official release* of the Popcorn.JS media wrapper for YouTube (*HTMLYouTubeVideoElement*) is relying on the YouTube API V2 and therefore deprecated. This is elaborated in section *4.3*.

Another matter to consider is YouTube's policy towards reloading of the same videos (duplicates). The service apparently compares hashes with previously uploaded videos, but the public API won't tell which video the current upload is being compared to (the *Creator Studio* will). Therefore the only feasible way to get this information is to hash files at the project side and maintain another database or table. This behavior was discovered during the debugging process but might as well arise if there is e.g. a mistake in the deposition process or a need to separate videos on the basis of privacy. Overall, it was then considered unsuitable to post such videos to the movie archive, but the YouTube option is left for a backup strategy.

4.2 Working with the Vimeo API

Since YouTube was proven to be insufficient, we had to come up with an alternative. Fortunately, Vimeo at least partially follows YouTube's example and satisfies the requirements too.

First thing to note is that Vimeo's user interface is notably more minimalistic than that of YouTube. Despite that, it allows all basic operations too, except for the video editor. Each can be executed via API; however, since Vimeo offers membership plans along with a free one, certain features are only available with paid account. Sadly, there is no single reference for these extensions in the API, instead they may just arise in the development process. For example, the ability for getting direct link for downloading the source of the video is an underdocumented function of the PRO API. Generally, the paid accounts get larger storage, detailed privacy and distribution settings, the video player customization and advanced statistics. After consideration, a PRO account has been purchased.

Similarly to YouTube, Vimeo requires a specific authentication workflow to be passed in order to start using its API. This is done in the following steps at the Vimeo Developer website:

- In the My Apps section, create an API app – which is not an actual application but rather its credentials. This generates *Client Identifier* (key) and *Client Secrets*. The app should get a status of approval for *Upload Access*. Vimeo says it takes up to 5 business days, however possibly due to the PRO status of the account it took only several hours.

- In the same section on the created app page, generate a token with the required scope. From the security perspective, a different token should be used for every other scope of use. Unlike YouTube, Vimeo does not enforce this policy, and a scope combining *public*, *private*, *create*, *edit*, *delete*, *upload* scopes can be set.

- The API uses OAuth 2.0. To make an API request with the token, the client key and the secret must be referred to as well. If anything else like the *bearer* (authorization type of the token) is needed, it will be obtained on the fly (with

help of some additional code) but will not be stored. For details on the method of authentication and its usage, see *Code 10* and *Code 11*.

```
1  def vimeo_make_auth():
2      mytoken='xxx'
3      mykey='yyy'
4      mysecret='zzz/aaa'
5      vimeovid = vimeo.VimeoClient(
6                                          token=mytoken,
7                                          key=mykey,
8                                          secret=mysecret)
9      return vimeovid
10      ...
```

Code 10: util-vimeo.py sample, authentication method. The credentials are hardcoded for clarification. A more clever thing to do is to import them

```
1  def vimeo_upload(args):
2      vimeovid = vimeo_make_auth()
3      video_uri = vimeovid.upload(
4                              # video_uri='video_uri',
5                              filename=args.file,
6                              upgrade_to_1080=False)
7                              vimeovid.patch(video_uri,
8                                              data={
9                                                'name': args.title,
10                                               'description': args.descr})
11     return video_uri
```

Code 11: util-vimeo.py sample, initialize-upload method

```
1  def create(request):
2          ...
3          if form.is_valid():
4                  uploaded_vid = form.save(commit=True)
5                  # send this file to vimeo
6                  vimeo_video_uri = vimeo_upload(uploaded_vid)
7                  vimeo_full_url = vimeo_request_url(vimeo_video_uri)
8                  uploaded_vid.vimeo_url = vimeo_full_url
9                  form.save() # for the placeholder
10         ...
```

Code 12: views.py sample, create view, calling Vimeo API functions

Like in the case of YouTube, the views of the project import and repeatedly call Vimeo API functions. The actual upload function is abstracted from the developer and rests in

the Vimeo library. Based on the credentials, it generates and utilizes an *upload ticket* that contains the metadata of the uploaded video. This is done via an authenticated *POST* request. The upload is made with a *PUT* request to the target URI. Once it is completed, the user can and will *PATCH* the result, updating e.g. the name the video or other parameters. See *Code 11* and *Code 12* for further details. On a side note, the videos can be uploaded repeatedly and will be regarded as separate entities.

To display the uploaded content, the *Django template language* along with *django-embed-video* app are used again. If one would try to share the embedding code from any Vimeo video using the *Share* button in the player, the result would be able to be effectively embedded without the black stripes. The *django-embed-video*, unfortunately, doesn't extract the correct aspect ratio or resolution for an *iframe* (container) element. It is an extraction indeed – and the values of the container should coincide with the aspect ratio of video resource itself. Vimeo allows to get the specification of any video in JSON by the means of a *GET* request (see *Code 13* for details). This JSON can contain e.g. a link to the video, its height and width, or the whole *iframe* code. The latter, unfortunately, uses default values of height and width as well. To display the correct width and height for every particular video, they need to be saved in the modified model along with e.g. the URL.

```
1 def vimeo_request_url(video_uri):
2         vimeovid = vimeo_make_auth()
3         headers = {"Authorization": "bearer " + vimeovid.token}
4         r = vimeovid.get("https://api.vimeo.com" + video_uri, headers=headers
              )
5         parsed = json.loads(r.text)
6         truncated = parsed['embed']['html'] # direct iframe code
7         ...
```

Code 13: util-vimeo.py sample, create view, calling Vimeo API functions

Other than that, there are two issues related to the display of the video. First, Vimeo degrades quality too, although often provides with more options, such as more gradual choice of resolution, the link to HTTP Live Streaming or the original video file. Theoretically, it can be employed in a third party player with Vimeo as a CDN (Content Delivery Network). However, it would require a substantial amount of effort to maintain and therefore left for the future development (see *Chapter 6*).

Last, but not least, Popcorn.JS does work well with the *official release* of the Popcorn.JS *media wrapper* for Vimeo (*HTMLVimeoVideoElement*). A full-fledged Popcorn.JS solution, however, requires more than just a wrapper. This is further elaborated in section *4.3.*

4.3 Popcorn.JS library and its appliances and implications

The idea of providing the user with annotations to a given scientific movie spreads out into two complementary elements: side annotations and overlay annotations. The first represent the relationship between a given movie and EMDB/PDB entries. The second and generally are free text (though can include links or even pictures) and appear on the top of the movie and should act as a tip to what is going on.

Unfortunately, few people use Popcorn.JS nowadays. The community is idle and powerless. The website of the library seemingly maintained by Mozilla, but not actually looked after – there are obsolete pieces of documentation, examples that no longer work and more importantly, and only the old revision (1.5.6) of the library itself available for download and on the CDN (Content Delivery Network). The open-sourced code on GitHub, however, is being updated relatively regularly – last commit to date (last check 28th of April 2016) was on 6th of June 2015.

It was suggested that those interested should build the required revision (1.5.11) themselves or use separate links for all necessary plugins. The Popcorn.JS documentation suggests using a "build tool" that is an available website that generates the old revision (1.5.7) once again. In the end the *Bower* package manager was used with a local copy of the GitHub repository. For an unknown reason it is not stated explicitly in the documentation, but inside the repository one can find the *Makefile* (basically a list of JavaScript modules) for the *Bower*. The *Bower* should be installed locally by the means of another package manager for JavaScript, *npm*. Other than that, it need correctly configured *Makefile* and *bower.json* (manifest file). This approach was tried successfully.

Another challenge in utilizing the library consists in its plugins. They allow to call a diverse range of functions. For example, it is possible to display stylized and animated

text, other media, extracts from Wikipedia, and/or Google maps in a timely manner along with the main media. However, not every plugin

a) is supported continuously

b) will suite specific needs

c) will integrate with other plugins flawlessly and in a known way

The Popcorn.JS documentation encourages developers to write their own plugins to support their cases. In our case, the task was to receive the data from the Django application and display both free text and EMDB/PDB annotations simultaneously. Accordingly, we needed to define the format of the extracted data and output this data in two different types of annotations.

The development had started with free text annotations, but closer to the end of the first Sprint it became clear that any attempts to extend their use to EMDB/PDB annotations would cause unnecessary delays. The data had been hardcoded (transferred literally) into the complex *div* HTML element that had a lower *z-index* CSS attribute than neighboring and parent elements. As a result, it was an overlay over the *video* HTML element. There were two major problems with it. First, its markup did not contain a special place for the style or the link, nor did it accept any code from Django. Second, due to the mixed nature of the resulting *div* element (one *div* on top of another), even if it would have a link, it would not be easy to make it clickable.

The second version of the movie annotation display was developed with mentioned challenges and issues in mind along with the deeper expertise in the domain. It is still an overlay, but now based on the modified *Popcorn Base Plugin* (Chirls 2012) collection, which in turn is an extension of the standard *Footnote* plugin. The *Footnote* plugin allows to pass text to the elements using arguments. The relationship between the Popcorn.JS and an HTML container is illustrated in *Code 13*. Most of the data can be easily inserted from Django using *Django template language*. For instance, the *pop.footnote* calls can be looped or supplied from a JSON file.

```
1  # index.html:
2  <script type='text/javascript' src='static/js/popcorn-complete.min.js'>
3  <div id="video"></div>
4  <div id="foo"></div>
5
6  # JavaScript
7  // create a Popcorn.JS instance
8  var wrapper = Popcorn.HTMLYouTubeVideoElement("#video");
9  // var wrapper = Popcorn.HTMLVimeoVideoElement("#video");
10 wrapper.src = "https://www.youtube.com/watch?v=1ZywgbBVUKE";
11 var pop = Popcorn(wrapper);
12 pop.footnote({
13             start: 1,
14             end: 5,
15             text: "Works with YouTube! <a href='http://www.mamk.fi/'>This is the
                   link</a>",
16             target: "foo"
17 });
18 pop.footnote({
19             start: 7,
20             end: 15,
21             text: "Yay!",
22             target: "foo"
23 });
24 // play video
25 pop.play();
```

Code 14: The example usage of the Footnote plugin for Popcorn.JS with media wrappers. The file popcorn-complete.min.js is an instance of the library

This qualifies for a common method for the two types of annotations as well, because the target can be different in each case. However, the display of the free text annotations in an overlay over YouTube and Vimeo *iframe* elements required further investigation. An HTML5 Video player was used as a working substitute, despite that it can only display open formats of videos such as *Ogg Theora* or *WebM*, or those which codecs were included in the operating system such as *MP4/H.264*. All the rest, proprietary formats such as *Apple QuickTime* would not be supported. The usage of professional HTML5 players such as *JW Player* would not help the issue (JW Player Media Format Reference).

The display of the side annotations with EMDB/PDB entries linked required the extraction of the corresponding data. Thankfully, there is a profound REST API specifically for this occasion. It provides JSON output with all details for major services, so what was need here is only to play with it to get the title, the description and the thumbnail of any particular EMDB/PDB entry.

5 THE INTEGRATION

The integration turned out to be the most intensive and the longest part of the project. It had to be done along with other functionality and took 1.5 Sprints (see section *2.6*) in total. It can be divided into the three parts.

First, it should be able to coexist with range of PDBe projects written in Django. The movie archive project itself is embodied in one Django app. What was described previously, however, is only the ground-laying of the project. Second, it was decided to utilize the existing authorization system, continuing to comply with the Django philosophy. Third, as was shown in the first round of user testing (see section *7.2*), the deposition workflow that was developed to demonstrate the feasibility required a substantial improvement. A new workflow had been designed and as a result, it was decided to modify the existing deposition workflow of the EMPIAR project.

5.1 General emdb_django project

The moving of the app from the movie archive-centered Django project to the new parent Django project requires several changes in configuration to be made. To start with, all other apps belonging to developers at PDBe need to be checked by Python at every run of the server. This implies that all packages that they are using should be installed on the local machine. Moreover, to deal with the development of more than one app at a time, developers work with their own *settings* module which is merged it in the SVN when it comes to the production.

By default, Django would direct any calls to the models to the single *default* database. In our case, most of the apps connect to their own database or an instance of the database. Handling multiple databases can be done either with specifying the particular database to use in the call or by setting up the *database routers* for the particular app's models. In this project, the usage of the latter was explicitly stated in the *settings* module. The routers are essentially simple methods to ensure the app will be querying the exactly predefined database. *Code 15* illustrates a control of a reading operation on models in the *Upload* (movie archive) app.

```
1 def db_for_read(self, model, **hints):
2         if model._meta.app_label == 'upload':
3             return 'movie_archive_dep'
4         return None
```

Code 15: routers.py sample, the method sends reading queries for the upload app to movie-archive-dep database

One other thing to consider is the arrangement of the templates. As seen in *Code 1*, for the sake of modularity the Django project can include layers of templates, usually at least app's templates and *base* templates (their directory lies in the root of the project package). Strictly speaking, the *base* templates need not and cannot be assigned to any particular app; yet among several developers, it is obvious that one keeps it clean and uses only own templates unless otherwise agreed. This project utilizes 2 (two) *base* templates. The subject of templates is covered in section *5.4*.

5.2 Using the EMPIAR User model and authentication system

The default User model in the Django framework is limited. It has only the field attributes such as *username*, *password*, *email*, *first-name*, and *last-name*, plus status and relationship attributes. In most cases and primarily for authentication purposes, it suits well. However, if there is a need in extended user profiles (such as an extended model for depositors with fields such as address and ORCID), it is advised to create a proxy model that would inherit from the original and override its functionality.

There have been discussions on whether there should be a common authentication system on the basis of an independent Django app that would be used by all future and some of the existing projects.

There are actually a number of ready-made solutions, reusable apps for handling registration and user profiling such as *django-userena* (Bread & Pepper 2013) for letting users operate their accounts. However, the movie archive project is utilizing the User model and the authentication system of another project, EMPIAR, to let existing users of that project operate with their credentials. In its turn, the EMPIAR project relies on the custom User model and the authentication that is built upon it, and so should the movie archive.

In order to create a *proxy model*, a custom class *UserProfile* was defined with a One-to-one relationship to the default User model. This user profile is represented by a *UserProfileForm* based on (but not inherited from) the model and the default *UserCreationForm*. Its fields are represented both by the built-in types that are defined inside the Django forms module like *forms.CharField* or *forms.EmailField* and custom fields made of other forms – which is allowed as forms are essentially classes in Python.

Other than that, a built-in user management system includes registration, password confirmation and reset, login/logout, as well as various user groups and permissions and the authorization (validation of the access rights of the authenticated user). All of them need to be specifically configured and called in the *views*. In the movie archive project, it is realized by the means of

a) adaptation of the EMPIAR's *views* related to basic functionality such as registration and login/logout;
b) wrapping the movie archive's *views* in the modified EMPIAR's *decorators* (authorization).

The above-mentioned decorators are used in some of the wrapped views that are covered in section *5.3*. In terms of Python, a decorator is an object (such as a function) intended to dynamically extend another object with the additional behavior. The concept belongs to the Python syntax from the version 2.4, allowing to prepend objects with the @ symbol for convenience. It is actually wrapping a given object and then returning a modified one. It is beneficial in terms of optimization and is often used in e.g. bounds checking (validation of an object within certain boundaries).

```
1  @never_cache
2  @is_depositor
3  def movie_deposition(request):
4          """
5          Requests object and returns display of the deposition list page
6          """
7          context = RequestContext(request)
8           ...
9
10 # The same result could be achieved with
11 def movie_deposition(request):
12          ...
13 movie_deposition = never_cache( is_depositor (movie_deposition))
```

Code 16: views.py sample, the decorators in action

The Django framework includes its own decorators that are sometimes used in this project. For example, the *@never-cache* decorator's name speaks for itself. The modified EMPIAR's decorators include checks of the user's status and permissions, such as the capability to create new depositions or make changes to existing ones. *Code 16* illustrates a case with both Django and own decorators.

5.3 The deposition workflow

At start, the deposition of new entry to the movie archive was fairly simple: a link to the Upload section, a few links to the editing subsections, a few concise forms (see *Figure 7* for illustration). The user experience testing that is elaborated in *Chapter 6*, however, had demonstrated that the whole workflow would benefit from a more ra-tional design. This doesn't concern merely a web page design that is elaborated in *Chapter 6*; rather, there was a need in an intuitive and thus an efficient process. Moreover, the release procedure had yet to be developed.

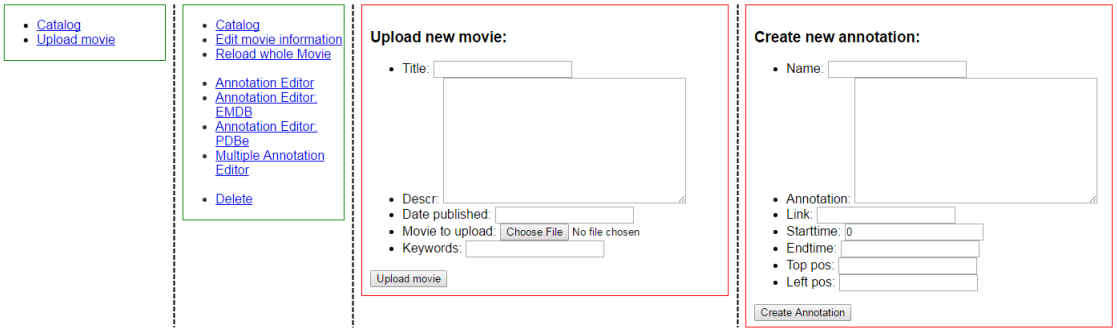

Figure 7. Menus and forms of the first user-tested revision of the project.

To tackle this issue, a deposition workflow had been conceived. Thereupon, the movie deposition had a life cycle and the user was to be guided at every step of it. The potential users of the movie archive had roughly been divided into two groups: the depositors and the annotators. The task of the depositors, who are usually owners or contributors to the movies, is to create a deposition, upload the movie and add metadata, and then submit it. The annotators would then review the deposited entry, make changes or send it back to the depositor, repeat, and finally release the entry.

The described workflow was generally similar to the correspondent workflow of the EMPIAR project which had already been a donor of the authentication system. Moreover, it would be at least remotely familiar to a potential user when implemented. Therefore it was decided to adopt and mimic the already developed solution that was the EMPIAR's workflow instead of writing a completely new one.

The structure of the EMPIAR project, however, is significantly more complex. Its purpose is in the storage of raw images – and as such, it should provide mechanisms for the upload of very large datasets (up to several terabytes) and their subsequent association with the metadata. This also implies a method of efficient communication between the depositor and the annotator. As a result, the data model is spread across each relevant group of users. It is done in a way of duplicating the entries with all their relationships to ensure the smooth transition of access rights. At this point, it was clear that the annotators' part of the workflow might not have bide its time in the end. Apart from that, a user can have access to their multiple depositions, but multiple users and multiple user groups can be assigned with access rights for the same deposition. The access rights differ in types – read, write, submit, owner.

As a result, the data *model* is fine-grained – the deposition entries have relationships with annotations of three types (free text, EMDB, PDB), with different types of authors and rights, and through the rights – with various possible depositors. The status attributes allow the movie entry to be either save or pended. The project operates with the selected EMPIAR *models* and *forms* when necessary (e.g. in authentication). The *forms* allow to populate them with information from the profile or other *models* such as authors. When the deposition has been completed, it is also possible to hold the entry instead of the submission – in a case when a depositor would like to wait for the publication or EMDB/PDB entry release.

5.4 The Front-end

This project is not about the front-end. It was intended to be usable by a professional community rather than responsive or generally good-looking. That being said, both sides had to be tackled to align with both the EBI guidelines and the actual user experience. The latter is described in *Chapter 6*, but it is necessary to provide the background and the approach to the issue.

As it was stated in *Chapter 4*, the Django framework allows to utilize its *template language* to render any arguments from *views*. However, it is not its only application. It is a programming construct, and as such, provides with simple logic such as conditional statements or loops or the concept of the template inheritance. The latter allows to have not one, but a set of templates for any particular project.

In this regard, the *base* templates define a *basis* of the site layout. They usually contain all standard HTML tags, styling and links to external resources such as JavaScript libraries. More importantly, they are builded in *blocks*, sections of the page that are placeholders for other templates. Insides of a *block* are then substituted by other templates using references to the particular *blocks*.

It is said that the apps' template *extends* the *base* template when its *block* with the content overrides the same *block* in the *base* template. Thus, the apps' template inher-

its the layout and styles of the *base*. It does not inherit any additional modules or template tags that can be *loaded* into the template. *Code 16* illustrates the concept.

```
# base.html
<!DOCTYPE html>
<html>

<head>
<title>
{% block title %}
{% endblock %}
</title>

{% block codeheader %}
{% endblock %}
</head>

<body>

{% block content %}
{% endblock %}

</body>
</html>
```

```
# index.html
{% extends "base.html" %}

{% block title %}
{{ argument.title }}
{% endblock %}

{% block codeheader %}
{% include "header.html" %}
{% endblock %}

{% block content %}
{% if many_args.count > 0 %}
{% for arg in many_args %}

<h1>{{ arg.title }}</h1>

{% endfor %}
{% else %}
<h1>Nothing to look at.</h1>
{% endif %}
{% endblock %}
```

Code 17: base.html, index.html samples, using the template tags.

As a result, the *base* template is the ideal space to comply with the EBI website guidelines and rules. These describe essential layout and design patterns that a developer is advised to follow, including the usage of grid systems for layout, the HTML5 standard compatibility, the EBI-specific colour palette and the usability techniques. The EBI also provides a boilerplate that is based on the *HTML5 Boilerplate* (H5BP 2015) and the *960px Grid System* (Smith 2014). This boilerplate is used as a base for the *base* templates for the project.

The movie archive does not utilize any complex visualization techniques. For the display of the depositions list, a *DataTables* plugin for jQuery library is used. The preview of deposition that is discussed in *Chapter 6* was challenging primarily due to a substantial amount of a routine work. The *base* template that was used in the Catalog and in the player page had contradicted with the *base* template for the deposition workflow. Apart from that, no problems with the generation of the layout were detected.

The general approach of appending the new elements to the UI was rather straightforward and can be described as “from the simple to the complex”. Their placement could have been changed during the development and in reaction to the feedback, as shown in Chapter 6.

6 USER EXPERIENCE TESTING

6.1 Aims, objectives and methods of testing

In order to produce tangible and applicable services, the PDBe group utilizes user experience (UX) testing as an integral part of the iterative development process. The feedback of the selected potential users helps developers to design features which are actually convenient and useful. Therefore, the outcomes and conclusions of testing might be included in the next Sprint (see section *2.6*).

For this project, an independent testing protocol had been devised and employed. The following illustrates a slightly simplified testing protocol:

1) Provide background information about the project – the idea with its justification, a general description of the implementation including details of technology and usage and relation to other PDBe’s services.
2) Explain the purpose of testing – that testing is an integral part of the development process and that the result will be used for further development.
3) Explain testing procedure – that testers are asked to navigate themselves in the interface and perform certain actions and observe how intuitive it is. The testers are informed that they are being observed as well. To avoid a possible experimenter bias, suggestions and tips would be fairly minimalistic.
4) The test is divided into three parts:
    a. 5 second test: show the catalog page for 5 seconds, remove it from sight and ask to describe the page and available functionality.
    b. Test case – watch the movie: ask to find a preselected movie, proceed to its page and play the movie. Then ask to explain what the movie is

about and what EMDB and PDB entries are involved and at approximately what time points. There are options in tabs to differentiate between YouTube and direct access – accordingly, ask to explain what the testers can see in terms of difference in functionality.

c. Test case – upload the movie: show the preselected video file and give some background on what it is about, which PDB and EMDB entries it should be linked to and at what time points; then ask to create a new entry in the archive (i.e. upload the video file and supply it with the given metadata). Consider:
    i. Are the testers able to find the login page and can they follow the directions for registration and login?
    ii. How do they explain what they see at the landing page of the deposition system?
    iii. Are they able to upload the video file, fill out the metadata including annotations and submit the entry?
    iv. What is their expectation of what will happen next?

5) Additional questions after the test:
    a. What improvements or new features would the testers like to see in the movie archive?
    b. What did they not like or find confusing with the movie archive?
    c. What kind of annotations should be captured for the movies? Which annotations that are currently captured are unnecessary?
    d. What type of controls would they like to use when filling out the annotations (e.g. for the time)?
6) Get background details needed to fill out the testing log.

Protocol 1. User eXperience (UX) testing protocol

A total of 6 (six) people were asked to follow the protocol and speak their mind. There were two rounds of testing, with users from PDBe and LMB (MRC Laboratory of Molecular Biology, Cambridge). Consequently, 6 (six) testing feedbacks were gathered and analyzed with a break of one Sprint (approximately a month) between testing rounds. The first round was carried out internally (at PDBe), while the second round included both internal and external participants. Two users from PDBe participated twice – in the initial testing and in the follow-up. All the rest participated only once.

6.2 The first round of testing and the analysis of gathered feedback

In the first round of testing, the participants interacted with the functional, but basic version of the movie archive project. The integration to the parent project had started by then; however the EMPIAR-like deposition workflow was not ready to be tested by a potential end-user. The elements of the interface and the logic of the deposition were structured the way it was relevant to the development process. The results of the testing show that this perspective was not always adequate to the user's expectations.

To summarize, both testers agreed on that

- in terms of styling, the general layout was uncluttered but could benefit from more outstanding and more intelligently grouped elements. For example, one suggestion was that EMDB/PDB annotations could assemble when there are too many of them, and another – that they should be grouped by time rather than type;
- in the player, it was expected to have the ability to pause the movie (which could be toggled on/off) when the EMDB/PDB annotation highlighted. This could become a guided walkthrough of the scientific context of a given movie;
- in the movie entry and annotation editing pages, there were difficulties in understanding the purpose and specifying of different inputs, especially of the position and the timing ones; the menus were not intuitive either;
- in the EMDB/PDB annotation editors, the whole concept of interaction with corresponding API was not obvious: from accession code format to filling the fields that are populated automatically.

In addition, they have both suggested

- to allow multiple annotations (free text, EMDB, PDB) to be completed on the same page and to see changes immediately (to have a more smooth workflow);
- to be able to select time from the video rather that textually and to have fine grained control;
- to have a clear "submit" button when the annotation is finished. At the same time, the whole movie entry should not be made public until it is "released".

The individual comments were regarded as well. Some of the notions had already been underway. However, since the project is rather short-term, we had to tackle certain priorities. Several most crucial suggestions had been discussed on the next Sprint planning and as a result, passed to the Product Backlog. In the final version of the project, the deposition workflow had been made anew, most of the found mistakes had been corrected and the player had not been altered in any way other than the addition of Vimeo video service.

6.3 The second round of testing and the analysis of gathered feedback

At the time of the second round of testing, the integration to the parent project had been almost completed and the participants interacted with the EMPIAR-like deposition workflow. Therefore it was essential to separate the first-time users from the second-time users who would notice the update. The protocol of testing, however, was general enough to not require changes.

The following summary illustrates the opinions of the users from the LMB site and one user from PDBe, who had not seen the interface before:

- The deposition workflow was substantially less perplexing and generally correlated with the expectations, except for the hold of the public release until all modifications are done and a corresponding request is made;
- There was little or no mention of the layout; however there were significantly more small unintuitive details. For example, in the movie entry and annotations editing pages overall, the left menu with green/red circle indicators was not always seen; the buttons 'Save' and 'Save + validation' were misleading for everyone;

- In the movie entry editing page, some fields such as ORCID (identifier for researchers) were consistently missed, inadequately filled or needed better introduction; buttons allowing to copy the existing information (e.g. from the user profile) were not frequently used;
- In the annotation editing pages, there were expectations to see a preview before the submission – because of the lack of comparison with the movie itself. This concerns not only the timing and the position, but the text and the gist of the annotation too;
- In the player, it was expected to have the ability to check and choose the time on the timeline, however the simple JS player wouldn't let to do so;

On the other hand, experienced users users were mostly focused on the acknowledgement of changes and the absence of such:

- They went to deposition straight away and passed through to annotations smoothly, copied the details using a button, quite naturally clicked 'Add More' in annotations – all that they could be to an extent familiar with as with the EMPIAR project (which is not their responsibility);
- Discussed on how the 'Save' button is different from 'Save + validation' and whether it's needed at all;
- Actually tested the interface in detail – for instance, discovered that fields that allow at least 10 characters in practice do not allow to input 'EMBL-EBI' as organization;
- Overall found the forms too stringent in comparison with previous version;
- Expected to see changes immediately (a preview) too;
- The drop-down lists for the position were considered a good option, but a suggestion was made to validate the position not only by its starting point but by the ending point as well;
- Another suggestion went to the timing settings – to be able to choose see 'start/end of video' in a checkbox instead of manually typing the number;
- Concerning the pausing of the movie that was reflected in the first round of testing, there was a complaint that without such functionality the eye switches between the EMDB/PDB annotations and the movie itself and thus one loses sense of the progress of the movie.

- A remark was made on that most of the time there would be only one EMDB and one PDB entry that correspond together. Hence in annotation editing pages there could be an option to freely add EMDB and PDB at the same timeframe – perhaps with a drop-down list and a button similar to 'Copy from user details' in the deposition form;

The second round of testing took place at the end of the last Sprint for this project. Hence, statements are better regarded as suggestions for the next generations of the project. That being said, a couple of suggestions were either aligned with the development course or relatively easy to implement (of which the buttons case would be a nice example). The several times mentioned preview has to do with the front-end development and was described in *Chapter 6*. Sadly, it was not available for testers in time due to the amount of work.

Overall, the test users actively shared their ideas, complaints and other thoughts, and even seemed to be generally pleased with the project. It gives some hope for the future.

7 CONCLUSION

The aim of this work was to demonstrate feasibility of a movie archive with annotations to link molecular animations to experimental structures from PDB and EMDB. Such a development could potentially take the burden from journals that are not presently able neither to store animations not to refer them to EMDB and PDB entries in a consistent way.

The implementation of the project took several steps. First, the scope of the techniques and technologies was known all along. Still, an evaluation of their particular capabilities had been done (Chapter 2). Consequently, a product backlog was devised in place of requirements specification, and was updated repeatedly as part of the iterative development process. Next, a number of revisions of the project had been developed, from the fundamental functionality to the integration with an existing code base (Chapters 3, 4, 5). Finally, the user experience testing in two rounds had been carried

out with its results being taken into consideration for the last and future revisions of the project (Chapter 6).

To name a few challenges, it had been discovered that good ideas might become abandoned out of priorities. This concerns, to start with, the Popcorn.JS interactive media library that is no longer well-supported and as such, raises difficulties where it shouldn't (Section 4.3). Apart from that, many proposed features in the product backlog had to be moved down because of the underestimation of the scale of the integration. In the end, the latter took more time than everything else (Section 5.3).

The outcomes, from the completed items and changes in the product backlog to the feedback from the user experience testing, resulted in multiple suggestions on the further development. For instance, from the technical POV, there is a lot to be improved in the Django architecture of the movie archive. On the other hand, the direction of the further integration into the PDBe services, as well as the collaboration with journals is undecided; the movie archive may become either an independent service or a supplement to the pillars it is relying upon such as PDB or EMDB.

On a final note, the nature of the project implies an open ending. Whether it will be supported by EBI/PDBe in the future or not, a state of functional prototype had been achieved and raised discussions of its application in the scientific community. The methods and technologies that had been piloted and documented are to an extent unique and stand a chance be adopted in future projects as well.

BIBLIOGRAPHY

Bread & Pepper company 2013. Userena Introduction — django-userena [version] documentation, documentation pages.
http://django-userena.readthedocs.io/en/latest/
Referred 25.04.2016

Brion Vibber 2015. Popcorn Maker is dead, long live Popcorn Editor, blog.
https://brionv.com/log/2015/10/02/popcorn-maker-is-dead-long-live-popcorn-editor/
Referred 15.04.2016

Chicurel Marina 2002. Bioinformatics: Bringing it all together, Nature.

Chirls Brian 2012. Popcorn Base Plugin, GitHub repository (MIT license).
https://github.com/brianchirls/popcorn-base
Referred 20.04.2016

Daniel K. Clare et al. 2012. ATP-Triggered Conformational Changes Delineate Substrate-Binding and -Folding Mechanics of the GroEL Chaperonin, Cell Press. Documents S2, S3, S4.

Django 1.8 Documentation, Design philosophies, Don’t repeat yourself (DRY).
https://docs.djangoproject.com/en/1.8/misc/design-philosophies/#don-t-repeat-yourself-dry
Referred 11.04.2016

EBI Website Guidelines.
http://www.ebi.ac.uk/web/guidelines
Referred 25.04.2016

EBI PDBe REST API documentation.
http://www.ebi.ac.uk/pdbe/pdbe-rest-api
Referred 20.04.2016

Google Project Hosting, Server-side issues and feature requests, 2008. New feed for accessing annotation data on Youtube videos.
https://code.google.com/p/gdata-issues/issues/detail?id=558
Referred 17.04.2016

Google, YouTube help pages 2016. Create and edit annotations, the video.
https://support.google.com/youtube/answer/92710
Referred 17.04.2016

Holovaty Adrian and Kaplan-Moss Jacob 2008. The Definitive Guide to Django: Web Development Done Right, Apress.

H5BP group 2015. html5-boilerplate, GitHub Table of Contents markdown.
https://github.com/h5bp/html5-boilerplate/blob/master/dist/doc/TOC.md
Referred 25.04.2016

JW Player Media Format Reference, Publisher Support pages.
https://support.jwplayer.com/customer/portal/articles/1403635-media-format-reference
Referred 20.04.2016

Merkley Ryan 2012. Online video -- annotated, remixed and popped, TED Talk.
https://www.ted.com/talks/ryan_merkley_online_video_annotated_remixed_and_popped
Referred 15.04.2016

Mozilla Foundation 2015. Product Update for Appmaker and Popcorn Maker, Mozilla Learning blog.
https://blog.webmaker.org/product-update-for-appmaker-and-popcorn-maker
Referred 15.04.2016

Patwardhan Ardan et al. 2014. A 3D cellular context for the macromolecular world, Nature Structural & Molecular Biology.

Python 2.7.11 documentation, 2016.
https://docs.python.org/2/
Referred 08.04.2016

Smith Nathan 2014. 960 Grid System, GitHub readme markdown.
https://github.com/nathansmith/960-grid-system/
Referred 25.04.2016

Tim Peters 2004. PEP 20 -- The Zen of Python.
https://www.python.org/dev/peps/pep-0020/
Referred 11.04.2016