A Generic Framework for Implementing Games With a Purpose

Carina Cocora

Bachelorarbeit

Betreuer: Prof. Dr. François Bry
Mentorin: Stefanie Schneider
Abgabe am: 15.02.2022
Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig verfasst habe und keine anderen als die angegebenen Hilfsmittel verwendet habe.

München, den 15.02.2022

Carina Cocora
Matrikelnummer: 11771870
ARTigo is an online platform offering several games with a purpose, their aim being to create user-generated artwork descriptions (through two-player games). The collected metadata is used to create and enhance a semantic search engine for these artworks. The backend of the current ARTigo gaming infrastructure offers a functional database as well as an API.

This thesis describes a novel implementation of the backend of the ARTigo gaming infrastructure using the Django Web framework. The project this thesis reports on includes both a restructuring of the database as well as a Django RESTful API that can be used for some of the existing ESP games and facilitates adding new games in the future.

The framework this work reports on represents one of the first steps in redesigning the ARTigo platform as well as a blueprint that eases adding or removing games from the platform.

This thesis also offers a perspective for extending both the players’ input and the artefacts players can describe. These extensions of the backend of the ARTigo platform provide a basis for adding new games with a purpose to the platform.
Zusammenfassung

ARTigo ist eine Online-Plattform, die verschiedene Spiele anbietet, deren Ziel es ist, von den Nutzern erstellte Beschreibungen von Kunstwerken (durch Zwei-Spieler-Spiele) zu erstellen. Die gesammelten Metadaten werden verwendet, um eine semantische Suchmaschine für diese Kunstwerke zu erstellen und zu verbessern. Das Backend der aktuellen ARTigo-Spielinfrastruktur bietet eine funktionale Datenbank sowie eine API.


Diese Arbeit bietet auch eine Perspektive für die Erweiterung sowohl der Spielereingaben als auch der Artefakte, die Spieler beschreiben können. Diese Erweiterungen des Backends der ARTigo-Plattform bieten eine Grundlage für das Hinzufügen neuer Spiele mit einem bestimmten Zweck zur Plattform.
First and foremost I want to thank Prof. Dr. François Bry and Stefanie Schneider for constantly providing guidance and feedback. I also want to thank all the people who put so much work into providing tutorials and publishing them on the internet. Finally, a thank you to everyone who supported me and reassured me when I had doubts.
# Contents

1 Introduction 1

2 Method 3

3 Related work 5
   3.1 Games with a purpose 5
   3.2 The ARTigo platform 6

4 Concept 11
   4.1 Defining the scope 11
   4.2 The former database and its drawbacks 13
      4.2.1 Describing the structure of the initial database 13
      4.2.2 Shortcomings of the initial database structure 14

5 Implementation 17
   5.1 Reimplementation 17
      5.1.1 Redesign of the former database 20
      5.1.2 Reimplementation of the backend 23
      5.1.3 Hypothetical game to be added 27
   5.2 Perspectives for future work 28

6 Evaluation 29
   6.1 Test design 29
   6.2 Test coverage 30

7 Conclusion and future work 31
   7.1 Games not included in the API 31
   7.2 Spell-proofing tags 32

Bibliography 45
Over the past decades, a significant amount of works have focussed on games with a purpose. This work will focus on the reimplementation of the backend of some of the ESP games on the ARTigo platform and propose a framework that can be used for other games with a purpose beyond the games currently available through the ARTigo gaming ecosystem.

Additionally, this thesis addresses some of the criticism that has been directed towards the design of the database structure and the backend design throughout the years and attempts to deliver an implementation taking some of these points into account.

The following chapter aims at giving an overview of the purpose of this thesis, the motivation behind this purpose as well as a summary of the approach employed in the course of this thesis project followed by a short overview of the results achieved. This will be followed by an overview of the topics researched so far in relation to games with a purpose generally as well as more specifically the ARTigo gaming ecosystem and the search engine behind it. In Chapter 4 the current database will be described as well as the shortcomings that it exhibits and where improvements can be made. The API that was implemented as well as the changes that were made to the underlying database are presented in Chapter 5. The evaluation of the code that constitutes the API that was implemented in the course of the project that this thesis reports on is presented in chapter 6. Finally, chapter 7 lists some ideas that could further extend the work that was presented in this thesis.
The reimplementation of the backend can be fragmented into two parts. The review and restructuring of the database and the requests to the database to either receive or save data.

The first step in the reimplementation of the backend for the game logic of some of the ESP games on the ARTigo platform was reconsidering the structure of the database. In the process of doing this, the database was restructured to allow for four of the existing games to function. Additionally, a potential extension of the database will be presented, that would allow for more diverse types of media to be added to it - both as resources in the game, to be tagged by users and multimedia to be inserted through user input. Additionally, given certain shortcomings of the original structure, which will be later discussed, the database was restructured to store more data, which can prove to be useful for the purpose of analyzing the different data types resulting from the games.

Elements of game logic that have to be handled through requests to the database include both those that modify the database in some manner by tags that are saved in a table according to some convention as well as those that do not but make use of existing tags. During a game, a tagging is saved to the database, once a tagging repeats itself, by means of another player having entered the same tag, this entry becomes a tag and is stored to the Tag table. This process is included in the first category. Due to the fact that the process of how a tagging is stored to the database or the taboo tags are selected to be shown to the player during a game is a matter of game logic and does not require for the database to be further extended, those processes will be handled as requests to and from the database.

The game logic of the original ESP game currently only requires tags to be saved to the database. However, games developed as an extension of the ESP game, designed with the purpose of gathering more specific tags make use of already existing tags. For those games, if tags result from them, a different type of request is necessary to save the tags to the appropriate tables in the database. Games with a taboo input or simply games that require a more specific input from users and especially games that make use of tags already existent in the database often include processes that do not make use of tables in the database specific to taboo input and beyond but rather access data from existing tables, where the taboo
input is selected from.

Extending the initial database to include all of the necessary tables and all of the required queries will be made using the django framework. This includes the entire process from adding the tables to the database to writing the necessary queries to the database such that the backend supports the game logic and facilitates the design of the URL scheme.

Eventually, after all required types of queries have been made for the game logic of all of the chosen games, a functional RESTful API should be available for the metadata resulting from these games. The data sent through the requests will be processed as JSON objects.
3.1 Games with a purpose

The term game with a purpose or GWAP for short refers to a category of games which involve humans performing one or more tasks as a by-product of playing a game. The tasks are difficult for a computer to perform but trivial for humans to perform. “Games with a purpose have a vast range of applications in areas as diverse as security, computer vision, Internet accessibility, adult content filtering, and Internet search.” [15]. This method is still successfully being applied for tasks which humans can still solve more efficiently than a computer, such as (most prominently) image recognition. [8] The data resulting from such a game can be used in a machine learning algorithm and therefore train a machine to do the task in the future.

The premise that the ESP game [16] was based on, was that images on the internet needed appropriate labels to make it easier for people to filter them and while computer vision techniques were not evolved enough at the time the ESP game was developed as a game that makes use of human skill to recognize and label images while making it entertaining and thus worthwhile for the humans to label the images accurately. The game itself implies that two players see the same image, are not able to communicate with each other, do not know anything about the other player and have to label the picture. The players have to label the images shown to them accurately under a time constraint of 2.5 minutes, they receive points if their labels match the labels of their co-player and bonus points if they agree on 15 images. It is not relevant if they enter the same label at the exact same time. Given this rule of the game, the players realize relatively soon that it would benefit them to guess what their co-player is thinking, which would incentivise each player to describe the picture in front of them accurately. A player can additionally choose to skip an image that is too difficult to label, in which case the co-player has to approve, once they have agreed on an image to be skipped, they can move on. To improve the accuracy in the description of images, a taboo input was introduced. The idea behind this is that players would have to work harder to describe an image through labels, since they are not allowed to use a
certain number of words that have been used very often for the image shown. [16]

The ESP game is an image labeling game that belongs to the category of games with a purpose and was developed in order to create descriptive data for images on the internet. In the course of playing the ESP game, players produce labels for images. The resulting labels were used to improve Web-based image searching. [8]

Several other problems that require much computing power and/or human skills can also be solved through a well conceived game with a purpose. Many other games with a purpose, where the purpose goes beyond improving image recognition algorithms were inspired by the ESP game throughout the years.

3.2 The ARTigo platform

ARTigo is an online game that has been developed at the Ludwig Maximilian University of Munich by the Play4Science group. [13] Figure 3.1 shows what the ARTigo homepage looks like. Each day, a new picture of an artwork is shown on the homepage along with the tags that have been entered most often for this picture and a caption. The ARTigo platform offers both a gaming environment consisting of eight games available in English, German and French as well as a search engine for artworks, powered by the annotations entered by different users in the course of playing the games. An example of a search using the keyword “ship” is shown in Figure 3.2. The ARTigo game itself was inspired by the ESP game designed by Luis von Ahn. [12]

Figure 3.1: The ARTigo homepage - Jules Elie Delaunay, The plague in Rome, 1869

Similarly, the purpose of the ARTigo game is labeling images with so-called tags. The images labeled through ARTigo are pictures of artworks that have been made available by different institutions. ARTigo was implemented as a game with solely a Single-Player mode with the second player being a simulation of a past game session since it cannot be guaranteed that exactly two players will play the game at the same time. For this reason and because a picture can only be used in a game once it has some labels, the administrators had to make sure that the pictures have some labels before the game can be played by
3.2. THE ARTIGO PLATFORM

regular players. Figure 3.3 shows the ARTigo game interface. A user can submit tags through the field below the image. The entered tags are shown as well as the number of tags entered by the other player and a timer indicating how many seconds are left of the current round as well as what round the user is currently playing. Tags entered by the user that match some tag already available for this picture receive 5 points and tags that match tags of the co-player receive 25 points. A game of ARTigo comprises 5 rounds, each round lasting one minute. A user does not have the option to skip a picture or move on to the next picture before the round is finished. After a game session, a user can once again look at the pictures of the artworks, this time the metadata is also shown for each separate picture. This feature was implemented on the premise that a user will learn more about an artwork after they have studied the picture beforehand during a game session.11

Figure 3.2: The interface of the artwork search engine

The study of art history consists primarily of studying artworks. Art historians rely on reproductions of the artworks as well as sufficient information on the artworks such as a title, the moment in time, when the artwork was created as well as the artist that created the artwork. This set of information, while relevant, does not offer sufficient information about the artwork itself. An alternative for making the pictures of artworks comparable and easy for art historians to categorize is annotating the pictures with labels that describe them accurately. The easiest method for coming up with the necessary labels is to use the time and knowledge of humans, more specifically laypersons through crowdsourcing. The annotations gathered for the artworks can be referred to as Tags. All of the above information form the Metadata referring to an artwork.13

Unlike a standalone image labeling game such as the ARTigo game, the ARTigo gaming ecosystem can be used to acquire more than simple annotations (referred to as Surface Semantics Tags). This is made possible because the output of some games can be used as input for other games generating more specific annotations (Deep Semantic Tags). The different games made available on the platform can be divided into the classes of Description, Dissemination, Diversification and Integration games. The ARTigo game, which is equivalent to the classic ESP game, belongs to the category of Description games. Each type of game collects a different type of a description of the artworks and improves the artwork search engine in their own way.17 All of the games that have been made available on the ARTigo platform are centered around the resulting Tags. While the ARTigo game is a
very good method of acquiring simpler Tags from users, the other games were thought of as an extension to the ARTigo game, meant to improve the quality of the resulting Tags and sometimes making use of the Tags resulting from ARTigo.

One game designed to collect more diverse tags (that does not however make use of pre-existing tags) is Karido. The main purpose behind Karido was to generate better labels for retrieving images. Karido is a game with a purpose in which players play the role of Guesser or Describer alternatively. Both players get to see a grid consisting of nine distinct but similar images of artworks from the database. The Describer has to describe the image that is highlighted on the grid that they see to the Guesser as well as possible. To this end, the Describer can send messages to the Guesser in the form of text. The Guesser can double click a picture if they consider that it matches the description. This image is then removed from the grid, the labels entered by the Describer are considered an appropriate description of the image and the Describer can choose the next image to describe. The game continues in this manner until there are no pictures remaining on the grid. It has been reported that Karido is however less enjoyable to play compared to ARTigo despite gathering more distinct annotations.

Scherz also compares picture databases made available through many museums to crowd based picture databases such as the one on the ARTigo platform as well as the metadata available in both cases. The metadata belonging to the picture database behind ARTigo, particularly the crowdsourced annotations contain many more content related terms as opposed to the metadata available for the picture databases of museums. In her dissertation, Scherz emphasizes the meaning of the ARTigo platform, the search engine created through crowdsourced annotations as well as the Tags themselves being a topic of scientific research.

The ARTigo game and more importantly the data obtained through the ARTigo project have also been studied from a computational linguistics perspective. The relevant metadata from this perspective are the tags that have been submitted by users. In her dissertation, Levushkina highlights some anomalies regarding the ARTigo tags such as the many synonyms available for the same picture and the high amount of misspelled tags. There-
3.2. THE ARTIGO PLATFORM

fore, semantic relations between the tags resulting from the ARTigo game such as synonymy, hyperonymy, hyponymy and meronymy have been analysed in order to improve search results.

The Tags respectively Taggings obtained through games on the ARTigo platform have also been the subject of various qualitative analyses. One analysis focused on answering questions such as whether the time pressure during some of the games influences the quality of the annotations gathered or whether a certain type of game is better at collecting more precise annotations, so-called annotations of a greater semantic depth than another type.

Some techniques for improving the quality of the annotations have also been studied. Most prominently, squaring and scripting of the tags. Those two approaches have yielded very good results when used on the ARTigo annotations both on their own as well as combined with one another. The use of these two approaches is also discussed beyond them being applied to the ARTigo data set. For the two approaches to be applied to the ARTigo annotations, the difference between surface and deep semantics tags needs to be highlighted. Surface semantics tags refer to those tags that describe what is visible in a picture as opposed to deep semantics tags that refer to what a picture of an artwork conveys. Squaring refers to the process of asking users to annotate data that has been generated through the ESP game itself. A good example for this technique being applied is the game Tag a Tag. Scripting on the other hand is a technique that refers to making the players act in a certain manner during the course of a game. Asking the players to only use annotations that describe a feeling conveyed by an artwork is a good example for this approach.

Ever since they have been in production, both the ARTigo game as well as the ARTigo platform and gaming ecosystem have been and remain subject to continuous research, primarily but not limited from an art historic and computer science perspective, which prompted many improvements to the platform and the games available on it over the years.
4.1 Defining the scope

An ESP game [16] is a game with a purpose, the purpose being the labeling of images. During an ESP game, two players come together and while seeing the same object, have to label it. This type of a game can be broken down into three fundamental parts: the object to label, most often an image shown to both players (or an image and a group of labels or something else), the input of the players, whether this is a string of characters, a sound, a video or another picture - input which has to fulfill certain criteria and what the players are restricted from using as input, what input types are forbidden to enter or what specific type of input is being expected from the player. The classic ESP game always includes a visible image - an object to be labeled during a game. When it comes to the ESP games on the ARTigo platform, a user can either submit a word (ARTigo, ARTigo Taboo), answer questions by either making a text input or making a choice from a list of words (Tag a Tag) or grouping two tags together to create more specific labels for the images (Combino).

The original ESP game, as it was designed by Luis von Ahn does not limit the type of tags a player can use in any way. Restricting the tags a player can use is referred to as “Scripting” the ESP game. The reason behind this is to prevent players from submitting tags that are very common. [17] Broadly, a taboo input is the type of input which the user should be restricted from entering during a game round. This input category can be of any type - an annotation (a string of characters), or even only allowing users to enter words from a category of words such as colours or even sentiments as mentioned by Wieser. In this latter case, everything else besides the words from the requested category would be considered a taboo input.

Starting from the database structure which only allowed for the original ARTigo game
to be played and metadata resulting from this game to be stored accordingly, one session of a game implies that two users (or simply one and a simulated second user) are paired up and submit their labels per resource for each one of the five game rounds. As soon as a user has entered a label, which has been previously entered by the other user as well, the label becomes a verified label and is stored separately. There is no taboo input for this type of game.

The setup for ARTigo Taboo, although similar to ARTigo, does include taboo input. The users cannot enter labels that match a label on the list of taboo tags. Similarly, Tag a Tag requires users to submit words as tags that answer a question. While in this case there is no explicit taboo input, users are only supposed to be answering the question and are therefore restricted from choosing the tags themselves. While ARTigo Taboo is the result of scripting the ESP game, this game is the result of both squaring and scripting the ESP game. The process of tagging of a tuple consisting of an annotation and a picture is referred to as squaring and achieving the semantic relationship between the different tags refers to the process of scripting. [17]

The game Combino on the other hand brings two tags together. During a round of Combino, the user sees an image and a sequence of strings, which have been used to describe the image beforehand, in a previous game round of ARTigo. The player has to submit a combination of tags, which are linked to each other semantically. [17] “The sets of annotations Combino collects are not really phrases but instead unordered sets.” [9]

At the moment, a game implies that at least a tagging or a tag of a picture, a tuple consisting of a picture and a tag or a combination of tags, which imply tuples of linked tags are created through user input per game round. An ESP game could however be extended to allow users to use more than just text to label an image or even input some other type of data for another type of multimedia (images, video, audio, text, documents, web pages, etc.) instead of only a picture.

Extending the potential types of input a user can generate in the course of a game would either mean that the existing database has to be extended with tables similar to the tables currently storing textual labels that have to be added to the existing database and a connection to the entire database has to be established from the newly added tables storing various other types of media.

Alternatively, it could also mean that the concept of a game mode could be introduced in order for the differentiation of “text”, “sound” or “video” to be made possible. In this other case, (for example by adding a Gamemode table to the database), it should be possible for players to choose how they want to play a game, whether it is by entering text, speaking a word out loud and uploading a recording of it, or uploading a video or picture from their devices, which shows similarities to a picture being shown to them or even a sound being played to them. Allowing players to annotate something other than a picture of an artwork but rather a piece of music or even a short film makes the differentiation between different types of resources stored directly or indirectly in the database necessary.

Storing different types of multimedia in a database has often been reported on and can be done in multiple ways. One way is by using so-called BLOBs (Binary Large Objects) to store the effective file in the database directly. Alternatively, including those types of files in the database can be done by using an external file directory and storing the data relevant to the multimedia file in database fields (such as the id of the user who submitted it, time it was created, size, etc.). The actual files are then stored externally to the database and a path to the location of the file in the directory is then stored in a field of the database for
4.2. The former database and its drawbacks

In this section, a brief description of the original database behind the ARTigo platform along with the data that can be stored will be given followed by some shortcomings of the former structure being highlighted. Since one of the objectives of this thesis project is to resolve some of the shortcomings of the database, the targets of the restructuring of the database will be enumerated, followed by an analysis of the data structures resulting from the different games that were chosen to be included in the interface. Finally, the structure of the redesigned database will be presented.

The original database for the ARTigo gaming environment saves all relevant information in a Postgres database and consists of 69 Tables in total. Through the Django migrations made for the redesign, 9 Tables, the so called Models are visible and can easily be explored through the django admin web page: Creator, Gameround, Gametype, Resource, Source, Tag, Title, Tagging and Gamesession. The UML association diagram in Figure 4.1 gives an overview of the tables or Django Models that resulted through the migrations before this thesis project was started as well as the multiplicities that exist between the different tables.

4.2.1 Describing the structure of the initial database

Every field in each table can be identified by an id stored in one of the fields. The id fields have not been added to the UML diagram for simplicity reasons.

The Gameround, Gametype and Gamesession tables store game-relevant data. The Tagging table stores all the tags entered by various players while Tag stores only the tags that have been entered by at least two players for the same resource. The Creator table stores the names of the artists who have created the art works depicted in the pictures being tagged, the Title table stores the titles of artworks and the Source table stores the institutions which have provided the pictures to be annotated while the Resource table stores all of the data regarding one picture depicting an art work.

An annotation becomes a tagging and is saved to the Tagging table as soon as a player types it in while playing a game. A tagging becomes a tag and is saved to the Tag table as soon as a second player submits the same tagging in relation to the same picture if the two players have been matched with one another and played the game together. A tagging can be traced back to a gameround (Gameround), which in turn can be traced back to a game session (Gamesession) and at last to a game type (Gametype). The Gametype table however only included the basic ARTigo game type, the image labeler after the migrations.
Figure 4.1: UML Diagram depicting the initial structure of the database, before the Redesign was finalized

were made and before the redesign of the database was finished.

A depiction of the Entity-Relationship (ER) diagram describing the structure of the initial database section referring to the gaming environment includes 16 tables for the original database and can be seen in Figure 4.2.

The Gameround table is connected to Gamesession, which in turn is connected to the Gametype table. The Gametype table includes the ARTigo game, ARTigo Taboo, the two versions of Karido (Turn and Time), Combino and ARTime. The Gameround is also referenced in Gameround_Resource, Gameround_Tag, Tagging, Person and sorting. Through the attributes gamerounds_id and opponenttags_id of the Gameround_Tag table, the synchronization of two players can be achieved.

The Person table refers to the registered users and includes an attribute for every potential user information that could be stored in this table.

Resources provided by different institutions are stored in the Artresource table, which in turn is connected to the Artresourcetitle and Artresourceteaser tables.

Tables where annotations are being stored as text in are the Tagging, Tag and ctag tables. The Combination and Combinedtag tables only store an id referring to a tag from one of the tables containing tags. The tag2tag table stores a tag_id and a correctedtag_id.

4.2.2 Shortcomings of the initial database structure

The following section of this chapter aims at highlighting some shortcomings of the structure of the initial database. It aims at highlighting what was missing from the former database and what could improve the structure in order to facilitate the addition of fur-
4.2. THE FORMER DATABASE AND ITS DRAWBACKS

The former database only allows pictures of artworks to be labeled at the moment. Users are also limited since they can only input text as a tag at this moment. This database also stores very little information about an artist and the title of an art work. The Creator table only stores the name of an artist and the Title table only the title of an artwork and the language of the original title. For the purpose of analyzing the annotations generated through the games on the ARTigo platform at a later point in time from an art historian’s perspective, further information about artists and titles can prove to be very useful.

Currently, the database does not offer the option of including any references to external web pages about artists or art works.

Basic information about a picture that will be tagged, including a path to the location where it is stored in the Resource table. The Creator, Title and Source tables, that are connected to Resource, store basic information about the artist, art work itself and the institution that provided an image depicting the art work. However, the Resource table stores some data, which can cause some confusion since the fields institution and source store very similar data.

Generally, there is very little information about the artists or art works themselves that can be stored in the tables of the database. Extending the available information on the objects to be labeled by players could help to better analyze the quality of tags collected through the different games at a later point in time more easily or answer specific questions (such as “Do the players label the style of an artwork accurately?”). This extension would also improve the possibility to make a selection from the database, improving the quality of the metadata.

It has been pointed out that similarities in the description of many different artworks, therefore a more thorough research of art history can be achieved by using various data
analysis methods. Optimizing the structure of the database to allow for more fields providing information about an artwork or an artist can only further improve the research possibilities.

The aim of the restructuring was to offset these shortcomings and come up with a new structure, where those problems have been corrected as well as to achieve the goals mentioned in the following section.
Implementation

The following part of the thesis focuses on the project conducted in the course of this thesis and how it has been implemented using the Django Web framework as well as, more specifically the Django REST framework for the design and implementation of the API. Additionally, the connection with the frontend of the application will also be briefly explained, even though the connection itself goes beyond the scope of this work.

The code developed in the course of this thesis project and presented in the following section can be run inside a docker container called api and can be accessed in the repository. This infrastructure including the docker containers that was provided for this project is the same infrastructure that was employed for the reimplementation and redesign of the entire ARTigo platform.

Furthermore, this chapter provides an overview of what the process of adding a hypothetical game would consist of, given the framework that has been implemented in the course of this project.

5.1 Reimplementation

The Django Web framework, the main framework used in the course of this project, is based on the Model-View-Template Architecture (MVT from here on). This software pattern contains three parts as the name suggests. These are the Model, representing the connection to the database behind the application, the View, the part that is visible to the user in the browser and the Template, a HTML mixed file. Therefore, the model handles the transfer of the data from the database to the views and the templates typically aid in displaying the data. [5]

One of the main reasons for using the Django Web framework is the Django Object Relational Mapper (referred to as ORM from here on) which accelerates the interaction with the underlying database. Additionally, Django offers a very easy to comprehend built-in admin interface. Furthermore, using this framework eases the testing process as well. The Django REST framework is easy to use with ORM data sources and eases processes such as serialization. [2] Serializers can also play an important role in defining what the various re-
CHAPTER 5. IMPLEMENTATION

Relationships in an API are. This characteristic can be extremely useful given the intricate data structures behind the ARTigo platform. Additionally, the Django REST framework also provides built-in renderer classes as well as the possibility to customize renderer classes. This feature makes it very useful for games such as the ARTigo game, since it makes returning various media types as a response not only possible but highly customizable as well.

The following chapter offers an overview of the Django Web Framework and the Django REST API Framework and how this was used for this project. The general setup of the project was done using the Django Web framework, the API was realized using the REST framework. Being that the implementation of the API realized in the course of this project was motivated by the general reimplementation of the ARTigo platform, this general implementation and where this project is situated in this context will also be briefly mentioned.

The connection between Django, the REST framework and the database as well as where this project is located in the context of the redesign of the ARTigo platform can be seen in figure 5.1, which gives an overview of the work presented in the following as well as situates the work that was performed in the course of this thesis project in the context of the overall reimplementation and redesign of the ARTigo platform and gaming ecosystem. As the figure illustrates, the Django framework offers a connection to the underlying database. The data serialization process takes place through the REST framework. The aim of this project is that the resulting API can be connected to the frontend of the redesigned gaming infrastructure on the ARTigo webpage, which has been re-implemented using the Vue.js framework. This thesis project does not cover the frontend development of the games but only the backend for four of the games available on the ARTigo platform.

Figure 5.1: An overview of how the database, Django and the Django REST framework work together to allow the application to work with Vue on the frontend side. (Source: This figure was inspired by the following Blog article https://medium.com/swlh/build-your-first-rest-api-with-django-rest-framework-e394e39a482c)

In Django, Models can be described as python objects that define the structure of stored
data. The fields of a model class are the equivalent of database fields in a regular SQL table. Django offers the possibility to define three types of relationships between models, more exactly for the data stored in the fields of models. These can be of the type one to one, one to many and many to many. The field names corresponding to where these types apply are OneToOneField for the one to one relationship, ForeignKey for one to many and ManyToManyField for the many to many relationship.

Additionally, there are many different field types and field options offered through Django, other than the field types specifying a relationship between two models.

Models therefore offer access to the database. The interaction with the Django models from the underlying database is performed through a database-abstraction API named Object-Relational Mapper or ORM. Once the models have been written in Python, the changes to the initial database need to be migrated. All of the necessary SQL commands that write tables to the database are taken over by the Django ORM.

A model can have an arbitrary number of fields, which can be of any type and can include various constraints for the value stored in the respective field. Each of the fields represents the column of a table in the database. These fields are the Model field references, as they are called in Django or field types, as they will be called from now on to avoid confusion. The fields can be specified by using class attributes in the models.py file. More details on the choice of database field types are presented in subsection 5.1.1.

Views are Python functions that contain the logic that is needed for the view to retrieve a web response after it has taken a web request. The web response can be anything that can be displayed on a web page. Whenever a view is supposed to display an HTML page, it will have to be connected to a URL, meaning that a URLconf will have to be created to support this. Django offers a highly flexible to use URL dispatcher. The URLs represent the endpoints for the API. Views in Django can be either function based or class based. Class based views consist of one or several methods, each one of the methods being the equivalent of a function based view. How this was implemented in the project this thesis reports on is outlined in subsection 5.1.2 of this chapter.

Using the The Django REST Framework typically includes the use of class-based views and provides an APIView class, which is a subclass of the View class in Django. For this reason, the API implemented in this project will make use of class based views as well. More specifically, APIViews will be used.

Serializers are one of the key components of the Django REST Framework. A view can make use of one or many serializers. The process of serialization occurs when models are converted into JSON objects. Through serialization, the models are being converted into a JSON representation, therefore anyone using the API will be able to parse the models even when not using Python. The user of the API will also be able to write data to the API, which then is converted into the JSON representation defined in the respective serializer and can be validated and added to the appropriate model. Eventually, when the backend of the application will be connected with the frontend, serializers ensure that objects are trans-
lated into data types that can be understood by the frontend programming language and framework used. The objects will be translated into JSON.

5.1.1 Redesign of the former database

The restructuring of the original database aims at fulfilling three fundamental goals:

Providing an appropriate way to save both annotations and other types of data resulting from the following games: ARTigo, ARTigo Taboo, Tag a Tag and Combino, while keeping in mind that other similar games will eventually be added to the platform. Bringing the database to a state that allows both for more than one media type (pictures) to be annotated by the players as well as allowing players to use various media types besides text to label a picture or another type of media. Extend the database such that more information can be stored for existing resources (previously only images of artworks) as well as make this change so that any amount of information can be stored for new (as well as existing) resources.

The first goal of the redesign process of the backend of the ARTigo platform also includes the task of making it easy to add new games, adjust the original games or even remove some of the existing games in the future.

Data related to users was not taken into consideration in this redesign process.

In order to fulfill its purpose of storing data resulting from games as well as data making the playing of games possible, the database needs to store three categories of data:

Data regarding the objects to be tagged. This category includes information about the resources such as about the artist and the title of the artwork that a resource represents. Data resulting from user input. This includes simple labels such as tags and combinations or variations thereof. Data regarding the structure of games such as the game type, game sessions and game rounds.

Upon analyzing the different games taken into consideration and especially the data structures resulting from them, it becomes clear that the logic of the ARTigo game and ARTigo Taboo is very similar and only bears a small difference to the logic of Tag a Tag. The only difference between ARTigo and ARTigo Taboo is that players are not allowed to enter certain tags from a list of tags that is being shown to the users. In Tag a Tag a user answers the question “What is the relationship between x and this image?”, x being a previously entered annotation for the picture. Similar to ARTigo Taboo, a list of suggestions are shown to the user for this answer. In this case however, users can use the suggestions as an answer. The answer of the user can also be treated as a tag for simplicity and especially since a tag stored in the database can be traced back to the game it was generated in. During the rounds of Combino, the user also sees a list of tag suggestions. Each player has to combine 2 tags and receives points for those combinations as soon as the other player combines the same 2 tags.

The process of the redesign of the database includes three distinct categories of changes. First of all, given the structure of the database after the migrations, there are a few tables to be added in order for the database scheme to be complete. Secondly, tables from the former scheme have to be modified to allow for more types of data to be added. And last, extending the current backend of the ARTigo platform to allow users to enter different types of media as input but at the same time to make it possible for users to be exposed to different types of media to be tagged from the side of the game. A third type of extension of the database is therefore necessary. In this case, it is an extension, the implementation of
5.1. REIMPLEMENTATION

the logic that would eventually be connected to these tables goes beyond the scope of this thesis project.

Given the choice of games, for which the backend has to be made functional in the course of this thesis project and the migrations that have been made from the former database, namely the scheme from which this project starts, some tables need to be added and some tables require alterations.

The game Combino is making use of already existing tags from the Tag table and creating a combination of tags through its game rounds. Therefore, adding the table Combination was necessary. This table contains the same fields as the Tagging table (Ids for the user and game round, time it was created and the obtained score), except for the fields tagging_id and group_id. The field tagging_id will contain references to the Tag objects that have been combined whereas group_id is the id of the tags that have been grouped together. This way, a combination of more than two tags can be implemented in the future as well.

Tags and Taggings resulting from the games Tag a Tag and ARTigo Taboo can be stored in the same tables as the tags resulting from the ARTigo game. No further changes are required to this extent.

![Figure 5.2: The UML diagram representing the final, redesigned database structure, including the changes necessary to fulfill all of the three goals for the database restructuring.](image)

To allow for more information on resources to be tagged to be stored in the database, the Creator and Title tables were extended to store more information. The Creator table now stores the dates between which the artist lived, the nationality of the artist as well as locations where the artist was active and techniques the artist used. The Title table now additionally stores the technique, style and movement which apply to the artwork. Due to these extensions, the tables Art_movement, Art_technique, Art_style and Location were also added, each containing the necessary information. The Creator and Title tables store a reference to the tables. Additionally, a field webpage was added to both the Creator and
Title tables. These fields reference the newly added WebPages table.

The table WebPages was added to allow for certain web pages to be added for different artists or artworks. The field url stores the path to the webpage whereas language stores the language of the respective web page. Therefore, multiple web pages can refer to the same artist or artwork.

The Source table, which was now removed, only stored a url to the institution providing the pictures. As mentioned in the section highlighting the shortcomings of the original database, the reference to the Source table from the Resource table as well as the field institution in the same table used to point to similar information. The table Institution was added and the reference to source in the Resource table was removed. This table will include the name of the institution providing the resource, a url to the institution website as well as the exact path to the image in the institution’s database in the field resource_url, if this is accessible this way.

For the purpose of adding further ESP games to the platform in the future and making the platform more flexible in terms of accepted input from users, the concept for a framework was developed. This framework permits the design of games that include more than just pictures and allows users to use multiple types of data as tags. How this would potentially affect the structure of the database will be presented in the following.

This extension of the database to include various types of media implies changes on both sides - storing the data used for different games and storing input from users. Currently, users can only tag one type of resource - pictures. A picture, just like any other type of media, this picture cannot be stored directly in a table. The Resource table was therefore extended with the field media_type, indicating the type of media stored in a row of the table. The Resource table however already includes a field called origin. Here, there is a partial path indicating where this file is stored in the file system used to store the pictures, which can be extended to include more than only pictures.

Users currently only have the possibility to use text as input. One way of extending this current setup, the field gamemode was added to the already existing Gamesession table. This table references the Gamemode table, which contains the name of the game mode, the type of media resulting from this game mode as well as an attribute storing whether the game mode is enabled or not. Given the current state of the framework, the only game mode that is enabled is the ‘picture’ mode, also referred to as the default mode from now on.

One way of storing various types of media as input is to store any type of input in the Tagging table. This would only require two more fields. The field media_type, which would store the type of resource and the field origin, which would store a partial path to the media type (similar to the origin field in the Resource table). Until the logic required to decode the different types of media into text has been implemented, the field tag in the Tagging table can simply remain empty. The breaking down of various types of media into text is however beyond the scope of this thesis. All of the media files being added to the database through this extension and as a result of enabling the concept of a game mode can be stored externally to the database in a file system.

Applying the changes explained in this section leads to the database structure that can be seen in figure 5.2. After creating the models, the resulting changes can be added to the database by making another set of migrations.
5.1. REIMPLEMENTATION

5.1.2 Reimplementation of the backend

The browser to backend interaction consists of HTTP requests that reach the backend through the endpoints accessed through the browser, namely the urls. Through the requests, the client may either be requesting existing data from the backend or make a request to introduce data created by the user to the database.

Creating this API using the REST Framework with Django comprises three steps after the database has been set up and the necessary migrations have been made. The conversion of the data from a Django model into JSON format in this case, called serialization, rendering the converted data to a view and finally defining a URL for the mapping to the view. The logic of the API is written in the views.py or more exactly the game_rest_views.py file, where the necessary classes corresponding to the different views are defined. For this API, the concept of class based views was used. More specifically, the APIView class provided by the Django REST Framework [2]. Using this approach, one view corresponds to a class. From the methods that can be used with APIView, only GET and POST methods were required for the chosen games.

Each view corresponding to a game contains a get and a post method, corresponding to the HTTP request methods. Since no process that has to be performed during any game for which this API was implemented modifies existing data from the database but either adds to it or simply needs to have some data retrieved, the only request types necessary for the ARTigo API are GET for retrieving objects and POST for sending data over to the server inside a request body in order to create a new instance of a model from the database. Every game view can be retrieved upon accessing the endpoint through the browser. For every view a url was created. The same url can be used for both get and post request methods.

The API comprises four games that are similar in the game logic to each other. The view classes forming the API for the ARTigo game subclass the Django REST Framework APIView class. The view classes forming the API are the ARTigoGameView, ARTigoTabooGameView, TagATagGameView and CombinoGameView classes. The ARTigoGameView, which was created for the backend logic behind the ARTigo game can be considered a blueprint or a point of departure for the API views for the other games as well. The view for the ARTigo game comprises two parts, the get and post request methods. All of the logic around sending tag and tagging instances to be validated and saved to the databases is located in the post request method. Meanwhile, the get method handles the retrieval of a random resource object from the database per game round as well as the creation of game rounds and game sessions. For every resource sent to the client, one game round instance is created, meanwhile every five game rounds require the creation of a game session. These two game related objects are created in the get request method to prevent the client from having access to sensitive game data such as the duration of a round for example.

For the views to function properly, serializers were necessary. Therefore, a serializer was created - first for every model reflecting a table in the database. However, as the complexity of the API grew, more serializers were necessary. A view can typically use one or multiple serializers, depending on how many types of data are involved. Views that represent an entire game, such as in this case make use of multiple serializers. For the ARTigo game itself, a simple serializer for every model was sufficient. The ResourceSerializer, GameroundSerializer, GamesessionSerializer and indirectly the GametypeSerializer were used for the get method, whereas the TaggingSerializer and TagSerializer were necessary for the post method.

The Django REST framework offers a so-called browsable API [2], which makes it possi-
ble to see the data retrieved through a get request in the browser by accessing the specified url. What is retrieved through the get request of the ARTigo game view can be seen in figure 5.3.

![ARTigo Game View in the browser](image)

**Figure 5.3: The ARTigo Game View in the browser**

Each one of the other games required some customized serializers. The serializers used either subclass the Django REST framework ModelSerializer class [2] or in some cases one of the serializers subclassing this class. The ARTigo Taboo, Tag a Tag and Combino games required a customized serializer for the object to be labeled, since this differs for every game. While ARTigo Taboo and Tag a Tag only differ from the ARTigo game in terms of what a user is required to label, Combino is also different from a user input point of view. This was solved by creating another serializer for Combino besides the serializer combining the list of tags along with a resource.

One aspect in which the other game views differ from the ARTigoGameview was the serializer used for the object to be labeled. Since the object to be tagged during ARTigo is always a picture, the ResourceSerializer was sufficient. For ARTigo Taboo, Tag a Tag and Combino, more complex serializers were required. The ARTigo Taboo game uses the TabooTagSerializer, which serializes a random resource and a list of taggings from the database and retrieves it through the get request method to the respective player. Similarly, the view for Tag a Tag requires the SuggestionsSerializer which includes a list of tags suggested along with a randomly selected resource and also makes use of the TaggingSerializer, which retrieves a random tag object to be labeled in connection to the retrieved resource. Combino in turn uses the CombinoTagsSerializer, which retrieves a randomly
5.1. REIMPLEMENTATION

selected resource along with a list of tags to be combined with each other by the player.

Serializers can be customized with fields that are not part of the model or can include logic for validating data that is sent to the endpoint through any field or can contain customized logic for the post request methods, these features were also used for the game API. Due to the fact that a significant part of the logic concerning the saving of a tag respectively tagging object in a post request was placed inside serializers, every game required a specific serializer for the post request method. Therefore, the TabooTaggingSerializer, TagATagTaggingSerializer and CombinationSerializer were created for ARTigo Taboo, Tag a Tag and Combino respectively.

The remaining games chosen to be included in the API exhibit some similarities to the ARTigo game itself, as was highlighted in section 3 of this thesis. For this reason, some elements so far described can be used for ARTigo but also for ARTigo Taboo, Tag a Tag and Combino. Since an object to be tagged is necessary to play ARTigo but also ARTigo Taboo, Tag a Tag and Combino, the manner in which this object is retrieved is one element that all four games have in common. Another such element is the entering of tags, which has to be done during ARTigo, ARTigo Taboo and Tag a Tag. One differentiation is required for Combino, since the user does not per se enter any novel data to the database but rather combines existing tags with each other, which results in a combination of tag objects.

Figure 5.4: The View of a Tagging object in the browser
One aspect that all games have in common is that players have to be synchronized to be able to play a game with one another. The games on the ARTigo platform are at the moment only implemented as single player games, where the co-player is just a player that has labeled the image the current player sees at a previous moment in time. This feature will also be maintained for the reimplemented version of the games. The easiest
manner in which this synchronization can be assured, is by querying the database for a
previously played game round, in which the same resource has been selected as the one
randomly returned during the game round that has just been started. Ideally, the game
round selected to be the virtual co-player would be a game round played by a different
player than the one playing the game round at the moment. However, this can only be
achieved in theory. Given this rationale, only the game rounds have to be coordinated and
a player can play against different players during the same game session.

Some views were created that can only display or save data related to one model. This
was mainly done for testing purposes before more complex views such as the game views
were created, especially for the purpose of testing the various serializers implemented in
the course of this project. These views include the TagView, TaggingView, GameResource-
View, CombinationView, GameroundView, GamesessionView and GametypeView. Figures
5.4, 5.5 and 5.6 show what the view for a Tagging respectively Tag and Resource object are
displayed as in the browser.

Finally, a timeout procedure was implemented for both get and post request methods
for every game view, such that the necessary time constraint for the games is in place. Once
the time that was set has passed, the client receives a timeout response from the server. The
timeout functionality for get request methods differs slightly from that for post methods
since saving a Tagging object to the database is tied to a Gameround object and therefore
the timeout is also tied to the time the game round was created. The timeout procedure for
the get request method is tied to the time the game session object was created.

For the games through which new Tag and Tagging objects can be added to the database
such as ARTigo, ARTigo Taboo and Tag a Tag, a normalization of the text was also imple-
mented in order to make the labels easier to analyze. Therefore, through the serializers
used for the post request methods of the three games mentioned above, the name of a Tag
object is saved in uppercase characters, irrespective of how the user entered the label.

An API Documentation for the above described API is provided in the Appendix of this
thesis and can also be accessed through the url path docs/, defined in the core/urls.py file
of this project. The code for the API is available in a GitLab repository. The link to this
repository can also be found in the Appendix.

5.1.3 Hypothetical game to be added

One of the main objectives of this thesis project was to come up with a solution to easily
add, change or remove games from the ARTigo platform. Due to this, the following section
will provide some insight into how adding a new, hypothetical game to the reimplemented
backend of the gaming environment given the framework achieved through this thesis
project can be performed.

Adding a new game to the gaming environment first requires coming up with a name and
adding this to the Gametype table in the database, along with the necessary properties,
such as the duration of a round, how many rounds this game consists of and whether it
should be enabled or not. Integrating this hypothetical game into the API would then
imply creating the necessary models, serializers and views. For each view that is being
added to the views file, an endpoint has to be created, such that the view can be accessed.
To this end, a path has to be added to the frontend/urls.py file.

Adding the game to the API implies creating a view and naming it after the game.
This view will make use of the relevant serializers to make the playing of the new game
possible, meaning that the necessary data needs to be sent and stored accordingly. Unless the game to be added requires methods that modify some already existent data, the get and post method structure present in every so far implemented APIView can be adopted for this game as well. As soon as the new game has been added to the API, tests need to be performed for the request methods implemented as a part of this game. In the event that new models were necessary for the new game, tests for these components also have to be added to the appropriate files.

5.2 Perspectives for future work

Even though the implementation of the API for the ESP games presented in the previous section can be used to make the games functional and provides a solution for a flexible backend for a platform offering multiple games with a purpose, there are many improvements or enhancements that can be applied to the backend implementation of the games on the ARTigo platform.

One feature that can enhance the quality of the labels collected through the games is for Resource objects to not be sent randomly in the course of a game anymore. Alternatively, a resource for which fewer (validated) labels exist can be returned at a future point in time. Resource objects could be filtered by tag count and especially validated tag count such that Resource objects with fewer labels can be prioritized.

A spell proofing mechanism that would lead to a corrected tag to be saved in addition to the tag entered by an user would also be an enhancement that would improve the quality of the collected data. More on the topic of spell-proofing labels will be presented in section 6.2 of this thesis.
This chapter provides insights into the testing process of the API described in the previous chapter as well as the extent to which tests were implemented for this project, namely the testing coverage. As with the environment that was provided for the implementation of the API developed in the course of this thesis, a testing environment was also provided.

6.1 Test design

The tests aiming at verifying the code written in the course of this project can be divided into four areas of the project. In order to test the RESTful class-based API, tests for these four categories were necessary, one for each one of the components making up the API. Therefore, tests for each one of the models and each method included in every class-based view were implemented. The urls representing the endpoints and serializers facilitating the data transfer and serialization were hereby indirectly tested as well. However, no additional tests were implemented for serializers or urls. Each one of those test categories corresponds to their own file in the project structure (test_models.py and test_views.py respectively).

Testing the models can be done using the command line as a first step by inserting data into the database and verifying that it was stored correctly. Using the Django framework to test models implies that even when the database has to be involved, the production database is not the one being used. Instead, a separate, empty database is created every time the tests are run and the tests will be performed using this virtual database. The database used for the tests is then deleted automatically as soon as the tests have been performed.

The two main types of tests that have been implemented for this project are unit tests and integration tests. For the most part of this project, testing after development was used. Therefore, all of the tests that were implemented for this project were written after most of the logic included in the views was already implemented and most models were finalized.

Testing the views was divided in testing the different types of requests implemented in the course of this project for the API. Testing views processing GET requests have first been
done in the browser as well as using the Postman application. View methods processing
POST as well as GET requests have first been tested using the Postman application.  

A testsuite for every model has been implemented with a so-called setUp method,
which creates a virtual object for the tests inside the test suite to be based on.

Depending on the model, different aspects were tested. For each one of the models, the
test suites include at least a test that verifies if an instance of the model can be created as
well as tests tailored to particular fields of the respective model that require verification.
Testing the models was done by using the Django testing framework.  

Multiple tests were made for every view representing a game. Every test suite corre-
sponding to a view first includes a setUp method as is the case with the model tests. Each
test suite for each view includes a test for the get and one for the post request. Given that
the views implemented in the course of this project were implemented by using the Django
REST framework, the tests for the views were implemented using the Django REST testing
framework.  

6.2 Test coverage

This part describes to what extent the framework was tested and how this was achieved.
Additionally, it also motivates the extent of the testing as well as go into which some parts
of the implementation did not require to be tested and for what reasons.

To measure the coverage that was achieved through the tests implemented for this
project coverage.py was used.  

Given the 63 tests that were implemented for the views and models an overall coverage
of 60 percent was achieved with 98 percent for the models, 81 percent for the views, 85
percent for the serializers and 60 percent for the urls.
Conclusion and future work

So far, the restructuring of the database and the implementation of the interface including 4 of the games available on the ARTigo website as well as the tests that have been implemented have been described.

Since the RESTful API implemented in the course of this project was never connected to the frontend of the application and therefore never tested by users with a functional user interface, it can be concluded that the results from the tests performed were not sufficient to draw any definitive conclusions at this point. Thus, the project this thesis reports about delivered only a proof of concept.

Additionally, the following section also goes into detail regarding the limitations of this thesis and the limitations of the implementation of the API will be discussed as well as certain aspects that go beyond the scope of this thesis but are nevertheless relevant for future work, especially given the reimplementation and redesign of the ARTigo platform as a whole.

Finally, perspectives for future work regarding the backend, more precisely a potential extension of the backend will be presented.

7.1 Games not included in the API

The interface implemented in the course of this thesis project included 4 out of the 8 games that are currently available on the ARTigo website. As outlined in section 3 of this thesis, the focus was on the games as similar as possible to the original ESP game in order to present the implementation of a generic framework for this particular type of games on the ARTigo platform and beyond.

The games Karido, ARTime, ARTigo Quiz and Crossover were not included in the API developed as part of this thesis. The queries that would have been necessary for the data required to make the remaining games available through the ARTigo gaming ecosystem functional as well as data resulting through these games were not included in the API.
CHAPTER 7. CONCLUSION AND FUTURE WORK

developed in the course of this project.

The main reason for not including these games in the API is that it does not align with the main goal of this thesis and implicitly of this thesis project, which was to find a way to implement a generic framework. Karido, ARTime and ARTigo Quiz are simply not similar enough to the other games to allow for this. Crossover on the other hand can easily be played given the framework implemented in the course of this project, due to the fact that it consists of 2 rounds of ARTigo, ARTigo Taboo and Tag a Tag each.

7.2 Spell-proofing tags

During the games, the user is required to enter text or make a type of input, that is then validated through the backend and eventually stored in the underlying database. During the ARTigo game for example, given the time constraint, spelling mistakes can often occur. Due to the fact that some of the other games use the ARTigo tags as part of the game logic and as a mechanism to generate other types of tags, having correctly spelled tags or a mechanism to automatically correct tags can prove to be very useful.

The tags resulting from the games on the ARTigo platform have often been analyzed. However, despite the relatively high number of wrongly spelled tags, no solution for this problem has been implemented so far and permanently included in the backend of the platform. The need for a grammar proofing mechanism of the annotations has so far been highlighted throughout the years.

Most prominently, Levushkina dedicated an entire chapter to the topic of grammar proofing labels and highlights the main reasons why the design of the ARTigo game itself could be causing the players to enter some labels with grammatical errors in them. Another reason for the erroneous tags could also be the fact that many users are not experts in the field of art history and are not familiar with how to spell certain specialty terms, as Levushkina also points out. The erroneous tags could ultimately either result in unverified tags or grammatically incorrect verified tags if two players enter a tag containing the same error. Potentially valuable tags could be lost in this manner.

The former ARTigo platform did not offer a grammar proofing mechanism for tags resulting through the ARTigo game or the ARTigo Taboo game (the two being the main games creating tags for images and not tags for an image and a tag or otherwise). So far, the reimplementation of the backend of the games described in the course of this thesis, does not include such a mechanism either. This feature is an objective for future work and goes beyond the scope of this thesis project.

In her work, Levushkina differentiates between spelling mistakes resulting from the user pressing the wrong key when typing a tag and cognitive errors resulting from the user not being aware of the right way to spell certain terms. As part of her dissertation, she analyzed the mistakes found in the ARTigo annotations, matched them with the entries of a dictionary, developed a mechanism of correcting the labels containing errors entered by players during sessions of the ARTigo game and presented an evaluation of this procedure yielding satisfactory results. The results were gathered during a period of approximately 6 months. During this time, the correction mechanism was actively used on the ARTigo webpage for all three available languages. The grammar proofing method presented by Levushkina does not involve an automatic correction of the labels entered by players but rather a player is shown a suggestion for a corrected label that they can easily accept, deny or simply ignore the suggested tag. If the player accepts the correction, their input is re-
placed by the suggestion in the validation process. In order for a corrected word to be suggested to the player, the edit distance was used. The Damerau-Levenshtein distance was used as a metric for this mechanism. This represents the minimum number of editing operations such as input, delete, replace and allowing for a swap as well between two strings. The edit distance was adapted based on the length of the word entered. During the course of a game round, if a word was recognized as false, the player would see a suggestion and if more than one corrected word was possible, the word most frequently entered for that particular picture was shown. The analysis showed that such a grammar proofing mechanism provides a good solution for spelling mistakes, even though sometimes correctly entered labels are falsely corrected. This mechanism is however not very efficient at identifying cognitive mistakes, as Levushkina points out, many wrongly spelled terms have already been validated in their erroneous form.

A mechanism that would correct the annotations entered by players on the ARTigo platform could be considered a worthwhile extension as part of the reimplementation of the ARTigo platform.
ARTigo API

api-token-auth

create

**POST** /api-token-auth/

Request Body

The request body should be a `application/json` encoded object, containing the following items.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>username</td>
<td>Valid username for authentication</td>
</tr>
<tr>
<td>password</td>
<td>Valid password for authentication</td>
</tr>
</tbody>
</table>

# Install the command line client
$ pip install coreapi-cli

# Load the schema document
$ coreapi get http://localhost:8000/docs/

# Interact with the API endpoint
$ coreapi action api-token-auth create -p username... -p password...

artigo_api

artigo_game > list

**GET** /artigo_api/artigo_game/

retrieves an game session as well as a random resource per game round

# Load the schema document
$ coreapi get http://localhost:8000/docs/

# Interact with the API endpoint
$ coreapi action artigo_api artigo_game list

artigo_game > create

**POST** /artigo_api/artigo_game/

allows users to post tags that are verified and saved accordingly to either the Tag or Tagging table

# Load the schema document
$ coreapi get http://localhost:8000/docs/

# Interact with the API endpoint
$ coreapi action artigo_api artigo_game create

artigo_taboo_game > list

**GET** /artigo_api/artigo_taboo_game/

retrieves an game session and a random resource per game round

# Load the schema document
$ coreapi get http://localhost:8000/docs/

# Interact with the API endpoint
$ coreapi action artigo_api artigo_taboo_game list
artigo_taboo_game > create

POST /artigo_api/artigo_taboo_game/

allows users to post tags that are verified and saved accordingly to either the Tag or Tagging table.

combination > list

GET /artigo_api/combination

Retrieves a random tag.

combination > create

POST /artigo_api/combination

allows users to enter a combination.

combino_game > list

GET /artigo_api/combino_game/

retrieves an game session and a random resource per game round as well as a list of tags to combine during a round.

combino_game > create

POST /artigo_api/combino_game/

allows users to post combinations of 2 tags that are verified and saved accordingly to the Combination table.

game_resource > list

GET /artigo_api/game_resource

retrieves a random resource object.

gameround > list

GET /artigo_api/gameround

retrieves a random game round.
gamesession > list
GET /artigo_api/gamesession
retrieves a random game session

gametype > list
GET /artigo_api/gametype
iterates through all gametype objects and chooses the correct one

get_collection > list
GET /artigo_api/get_collection

get_user > list
GET /artigo_api/get_user

get_user > create
POST /artigo_api/get_user

search > list
GET /artigo_api/search

tag > list
GET /artigo_api/tag
retrieves a random tag from the database
tag > create

**POST** /artigo_api/tag

Allows users to post tags that are verified and saved accordingly to the Tag table.

---

tagtag_game > list

**GET** /artigo_api/tagtag_game/

Retrieves a game session and a random resource as well as a tag and a list of suggested tags per game round.

---

tagtag_game > create

**POST** /artigo_api/tagtag_game/

Allows users to post tags that are verified and saved accordingly to either the Tag or Tagging table.

---

tagging > list

**GET** /artigo_api/tagging

Retrieves a random tag.

---

tagging > create

**POST** /artigo_api/tagging

Allows users to post taggings that are verified and saved accordingly to the Tagging table.
rest-auth

login > create

POST /rest-auth/login/

Check the credentials and return the REST Token if the credentials are valid and authenticated. Calls Django-Auth login method to register User ID in Django session framework.

Accept the following POST parameters: username, password Return the REST Framework Token Object's key.

Request Body

The request body should be a "application/json" encoded object, containing the following items.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>username</td>
<td></td>
</tr>
<tr>
<td>password</td>
<td>required</td>
</tr>
</tbody>
</table>

logout > list

GET /rest-auth/logout/

Calls Django logout method and delete the Token object assigned to the current User object.

Accepts/Returns nothing.

logout > create

POST /rest-auth/logout/

Calls Django logout method and delete the Token object assigned to the current User object.

Accepts/Returns nothing.
### password > change > create

**POST** /rest-auth/password/change/

Calls Django Auth SetPasswordForm.save method.

Accepts the following POST parameters: new_password1, new_password2. Returns the success/fail message.

#### Request Body

The request body should be a "application/json" encoded object, containing the following items.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>new_password1</td>
<td>required</td>
</tr>
<tr>
<td>new_password2</td>
<td>required</td>
</tr>
</tbody>
</table>

---

### password > reset > create

**POST** /rest-auth/password/reset/

Calls Django Auth PasswordResetForm save method.

Accepts the following POST parameters: email. Returns the success/fail message.

#### Request Body

The request body should be a "application/json" encoded object, containing the following items.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>email</td>
<td>required</td>
</tr>
</tbody>
</table>
### password > reset > confirm > create

**POST** /rest-auth/password/reset/confirm/

Password reset e-mail link is confirmed, therefore this resets the user's password.
Accepts the following POST parameters: token, uid, new_password1, new_password2. Returns the success/fail message.

**Request Body**
The request body should be a "application/json" encoded object, containing the following items.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>new_password1</td>
<td>required</td>
</tr>
<tr>
<td>new_password2</td>
<td>required</td>
</tr>
<tr>
<td>uid</td>
<td>required</td>
</tr>
<tr>
<td>token</td>
<td>required</td>
</tr>
</tbody>
</table>

### registration > create

**POST** /rest-auth/registration/

**Request Body**
The request body should be a "application/json" encoded object, containing the following items.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>username</td>
<td>required</td>
</tr>
<tr>
<td>email</td>
<td>required</td>
</tr>
<tr>
<td>password1</td>
<td>required</td>
</tr>
<tr>
<td>password2</td>
<td>required</td>
</tr>
</tbody>
</table>
registration > resend-email > create

**POST** /rest-auth/registration/resend-email/

**Request Body**
The request body should be a `application/json` encoded object, containing the following items.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>email</td>
<td>required</td>
</tr>
</tbody>
</table>

registration > verify-email > create

**POST** /rest-auth/registration/verify-email/

**Request Body**
The request body should be a `application/json` encoded object, containing the following items.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>key</td>
<td>required</td>
</tr>
</tbody>
</table>

user > read

**GET** /rest-auth/user/

Reads and updates UserModel fields. Accepts GET, PUT, PATCH methods.

Default accepted fields: username, first_name, last_name. Default display fields: pk, username, email, first_name, last_name. Read-only fields: pk, email.

Returns UserModel fields.
user > read

**GET** /rest-auth/user/

Reads and updates User Model fields. Accepts GET, PUT, PATCH methods.

Default accepted fields: username, first_name, last_name

Default display fields: pk, username, email, first_name, last_name

Read-only fields: pk, email

Returns User Model fields.

---

user > update

**PUT** /rest-auth/user/

Reads and updates User Model fields. Accepts GET, PUT, PATCH methods.

Default accepted fields: username, first_name, last_name

Default display fields: pk, username, email, first_name, last_name

Read-only fields: pk, email

Returns User Model fields.

**Request Body**

The request body should be a `application/json` encoded object, containing the following items.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>first_name</td>
<td>required</td>
</tr>
<tr>
<td>last_name</td>
<td>required</td>
</tr>
</tbody>
</table>

---

user > partial_update

**PATCH** /rest-auth/user/

Reads and updates User Model fields. Accepts GET, PUT, PATCH methods.

Default accepted fields: username, first_name, last_name

Default display fields: pk, username, email, first_name, last_name

Read-only fields: pk, email

Returns User Model fields.

**Request Body**

The request body should be a `application/json` encoded object, containing the following items.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>first_name</td>
<td></td>
</tr>
<tr>
<td>last_name</td>
<td></td>
</tr>
</tbody>
</table>


