Manual

The Unified Workbench for Knowledge Graph Management Platform

Version 1.0

The Unified Workbench for Knowledge Graph Management Platform
Version 1.0 Published February 29, 2020

National Institute of Advanced Industrial Science and Technology (AIST)
2-3-26 Aomi, Koto, Tokyo, Japan 135-0064

National Institute of Informatics (NII)
2-1-2 Hitotsubashi, Chiyoda, Tokyo, Japan 101-8430

Authors:
Research: Natthawut Kertkeidkachorn
Platform: Rungsiman Nararatwong
Principal investigator: Ryutaro Ichise

See *https://github.com/ichise-lab/uwkgm* for release details.

Table of Contents

Preface

Building the Knowledge

Every day the global 4.1 billion internet users and the Internet of Things are creating over 2.5 quintillion bytes of data[1] (B. Marr, 2018). With this enormous amount of input, much of humans' knowledge constantly dissipates into the world of disconnected data storage. As a result, we are missing many opportunities to learn from these hidden links. For many years, determined to unearth these treasures, both academic and industrial communities have built and bettered graphs that capture real-world entities and their relationships. They called them "knowledge graphs."

In knowledge graphs, each entity has its unique identity, as well as its connections to other entities. The graphs are self-descriptive – thanks to the invention of ontology – meaning that not only can you retrieve the entity (such as "Amazon") but its semantics or meaning (Amazon as a company, not a forest) as well. This powerful concept provides great flexibility for querying, which may help you to discover implicit information (e.g. Amazon purchased Box Office Mojo via its subsidiary, IMDb). As we progress through the implementation of the platform, you will find some actual queries that may help you to grasp the idea of the flexibility we mentioned.

Despite the great potential of knowledge graphs, the reason why the UWKGM platform exists is because such graphs are not easy to build and maintain. People still need to rely on both humans and machines to translate real-world data, either public or private,

[1] ITU statistics: https://www.itu.int/en/ITU-D/Statistics/Pages/stat/default.aspx

into a well-organized knowledge graph. In this manual, we will these issues and our methods to solve them.

UWKGM Platform in Your Projects

If any of your projects involve creating, maintaining, or utilizing knowledge graphs, the UWKGM platform is the basic essentials that you need to build and expand your systems. Our platform offers tools that will not only help you to manage your graphs but also provide you with powerful language processing capabilities for your AI. The following are some of our target audiences who may benefit most from the platform:

- *Organizations.* Aggregate your organization's complex data and build relations with massive knowledgebases such as DBpedia and Freebase to generate insights from a wider perspective.
- *Researchers.* Easily acquire, store, access, and expand your knowledge graph, as well as streamline research procedures with our research toolkit.
- *Developers and end-users.* As an open-source project, you can customize, scale, and extend the UWKGM platform to fit your needs, while also control who gets access to your data and tools.

Prerequisites

We write this manual for developers who want to deploy, adjust, modify, or contribute to the UWKGM platform. We assume that you have basic understanding of REST API, Python and both relational and non-relational databases. While we will provide basic explanation of the technologies on which we built our platform, we recommend that you familiarize yourself with the following topics:

- *SPARQL 1.1 Query Language for RDF*. Our platform follows the Resource Description Framework (RDF) standards and we use SPARQL to query and modify data the graph database. For references, we recommend W3C Recommendation page: *https://www.w3.org/TR/sparql11-query/.*

- *DBpedia.* We use DBpedia datasets to develop and test our platform. The DBpedia project extracts data from Wikipedia and created knowledge graphs based on RDF. Visit *https://wiki.dbpedia.org* to download their datasets.
- *Django and Django REST Framework.* Django is a Python-based web framework. The Django REST Framework extends Django with a toolkit for building Web APIs: *https://www.django-rest-framework.org*.
- *Virtuoso, MongoDB, and MySQL.* These are the database management systems that we have used so far in our development. While the platform is designed for future extensions with different DBMS, these three will work with the platform without additional connectors.

Other resources that may help with your deployment of the platform include:

- *Docker and Kubernetes.* We built a set of scripts that will help you to set up and deploy the platform on your preferred servers or in the cloud with a few-steps configuration. Learn more about Docker and Kubernetes to customize these scripts further to fit your requirements.

Content Organization

This manual is organized into three parts. The first part, UWKGM fundamentals, explains the idea and research behind UWKGM. It contains only one chapter and is intended for all readers. The second part, the UWKGM platform, dives into the detail of every part of the platform. It is particularly useful for developers who want to contribute to the project, as well as system administrators who want to make modifications to their deployments. Chapter 2 explains the underlying mechanism on which the platform operates. The chapter should be particularly useful for the developers. For system administrators, Chapter 3, which describes the platform's structure, should give them a comprehensive idea of how to operate, maintain, and make changes to the platform. The third part of the manual, Web API references, is intended for general users who use the Web API for their projects.

Source Codes

You can download all source codes in this manual from our GitHub repository at *https://github.com/ichise-lab/uwkgm.* Some of the codes in this manual leave out sections or details that are not the focus of the explanations. However, we provide a path to the script where each code section resides for further reference. We may include example use cases and their results if applicable. For example, the following code displays a greeting message:

```
/api/example/hello.py
>>> def hello(name: str) -> str:
...     return 'Hello %s!' % name
...
>>> hello('knowledge graphs')
Hello knowledge graphs!
```

The actual code may contain short explanations which this manual does not include.

How to Contact Us

Please contact Ichise Laboratory at National Institute of Informatics for inquires:

Ichise Laboratory
Principles of Informatics Research Division
National Institute of Informatics
2-1-2 Hitotsubashi, Chiyoda, Tokyo 101-8430, Japan
+81-3-4212-2000, +81-3-3556-1916 (fax)
uwkgm@nii.ac.jp

For demonstration, we provide a web user interface of our deployment of the platform at *https://uwkgm.nii.ac.jp.* The website is available in English and Japanese.

Acknowledgement

This work was partially supported by the New Energy and Industrial Technology Development Organization (NEDO).

Chapter 1

The UWKGM Research

Curating and managing knowledge graphs take considerable time and manpower. In our research, we were able to identify three major problems: adding new knowledge, erroneous knowledge injection, and inadequate knowledge. As a large amount of new knowledge is being created every day, it is simply too much for humans to handle. Therefore, automatic approaches that dynamically add new entities and relations to knowledge graphs were invented. Came with the methods are problems of injecting erroneous knowledge into knowledge graphs, some of which are due to human errors. Knowledge graphs with invalid data may lead to underperforming AI. Thus, verification and validation are essential to making the graphs reliable. Lastly, another common problem with knowledge graphs is that the knowledge is incomplete. To complete the missing knowledge, we need automatic reasoning, which is crucial to AI applications.

To help solve these problems, we present our framework, the Unified Workbench for Knowledge Graph Management (UWKGM). Although framework initially integrates and provides a set of technologies as a tool to manage knowledge graphs, our design allows it to support expansion that includes future advanced technologies.

Unified Workbench for Knowledge Graph Management

UWKGM consists of four main components to manage knowledge graphs: extraction, integration, verification, and completion. Knowledge extraction and integration constitutes the process of adding new knowledge; knowledge verification helps eliminate erroneous knowledge injection; lastly, knowledge completion helps solve the insufficient knowledge problem.

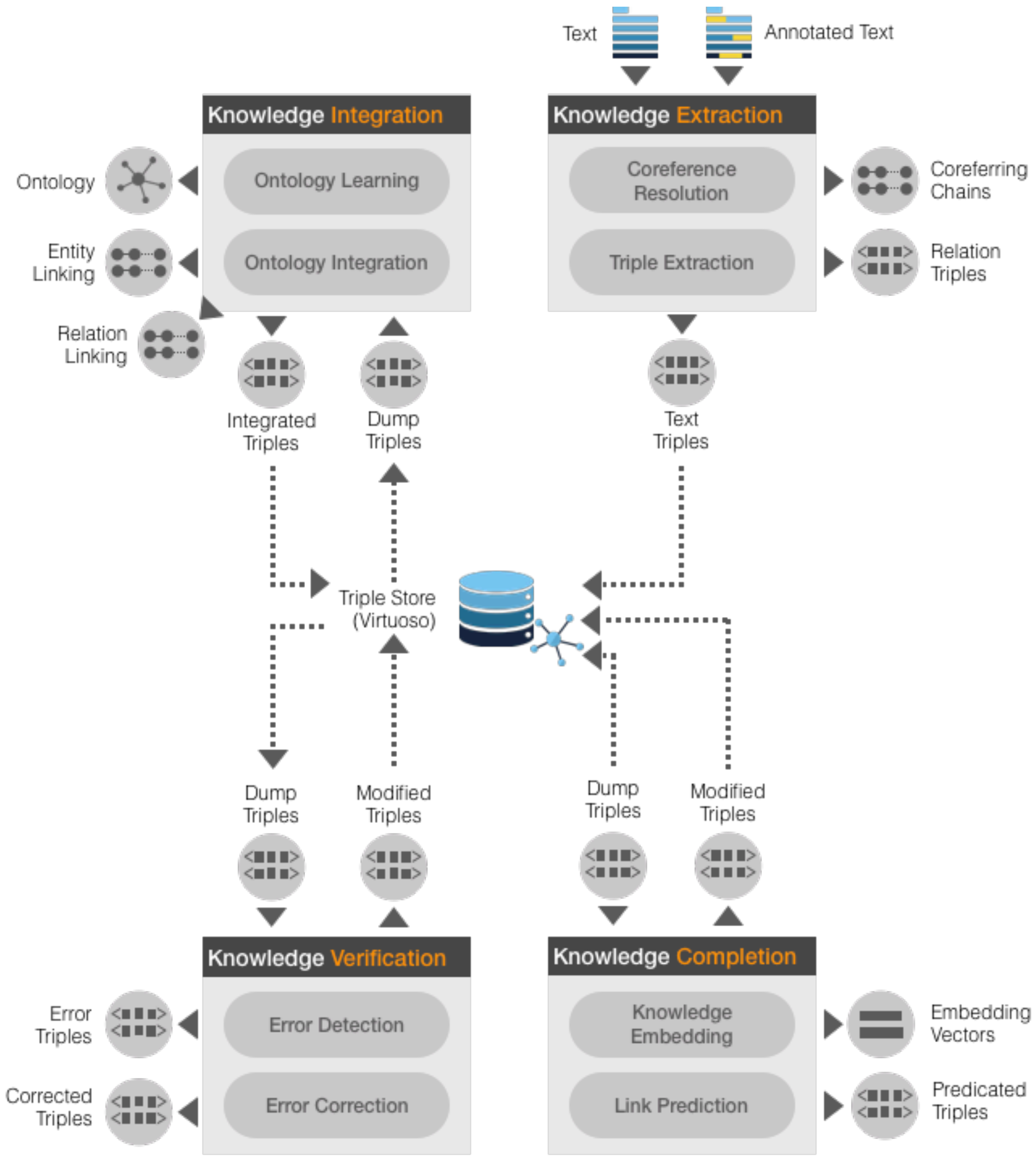


Figure 1.1 The UWKGM framework's architecture

Knowledge Extraction

This part of the framework retrieves relationships between entities from unstructured text. The relationships are represented as triples. We use Stanford's Open Information Extraction (OpenIE) and Coreference Resolution to extract triples from texts (Angeli et al., 2015). OpenIE extracts relation between two entities as a triplet (e.g., "Barack Obama was born in Hawaii" => ["Barack Obama", "born in", "Hawaii"]), and Coreference Resolution combines surface forms of an entity into a representation (e.g., "**I** voted for **Nader** because **he** was most aligned with **my** value," **she** said). The extraction component also allows domain experts to curate bootstrapping data to improve triple creation. For our configurations, see T2KG (Kertkeidkachorn and Ichise, 2018).

Knowledge Integration

The integration process serializes text triples to triples in Resource Description Framework (RDF) standard, before integrating them into knowledge graphs. The process consists of entity and predicate mappings. Entity mapping maps an entity in unstructured text to a Uniform Resource Identifier (URI). For example, ["Barack Obama", "born in", "Hawaii"] would be mapped to ["dbpedia: Barack_Obama", "born in", "dbpedia: Hawaii"]. Predicate mapping maps a predicate of a text triple to a predicate in knowledge graph, e.g., "born in" in the previous example would be mapped to "dbpedia: birthPlace".

Knowledge Verification

The verification component verifies and validates RDF triples before a knowledge graph with new entities and relations is being published. This component consists of two main modules, namely error detection and error correction. The detection module use constraints in ontology and pattern analysis to find errors in newly added triple (Lertvittayakumjorn et al., 2017; Rahoman and Ichise, 2016). The correction module is based on FIXRVE (Lertvittayakumjorn et al., 2017).

Knowledge Completion

The completion component learns embedding representations within a knowledge graph and infers the graph's missing knowledge. The component consists of two modules, which are knowledge embedding and link prediction. In knowledge embedding, entities and relations in knowledge graphs are represented as vectors in low-dimensional space. There are many AI applications that can take advantage of the representations, such as fact-checking. The link prediction module, based on TorusE (Ebisu and Ichise, 2018), predicts missing relationship among entities in a knowledge graph.

Chapter 2

Underlying Technologies

Before we dive into the core architecture of the UWKGM platform, this chapter will explore the underlying technologies which we used in our development. The platform is essentially a purpose-built knowledge graph management system operating on top of several database management systems and provide controlled access to its functionalities via Representational State Transfer (REST) protocol. Most parts of the platform are Python-based, and so we created a RESTful API service provider using a well-known high-level Python Web framework *Django*.

We will begin this chapter with our introduction to a few basic features of Django that we used for creating the API service. Next, we will move to one of the most popular Django extensions for REST API development, *Django REST Framework*. We have created an extension called *Structured Endpoints* on top of Django REST Framework to facilitate the platform's complex API structure. In the section following the framework, we will describe our idea and its implementation.

We should note that our explanation of Django and Django REST Framework in this chapter is not a tutorial but a brief introduction to familiarize you with how they work. Should you be interested to learn further, we recommend that you visit their websites for step-by-step tutorials and documentations.

Building a Web Service with Django

In general, Django is a Python-based web framework that follows the model-template-view (MTV) architectural pattern. A Django project is, by itself, a standalone Web service. Simply install Django in your Python environment using **pip install Django**, create a Django project using **django-admin startproject mysite** in your preferred directory, and run **python manage.py runserver** in the project directory and you will have a functioning development server ready to serve.

Before you continue further, make sure that you have version 3.0.3 of Django that is compatible with the one used by the UWKGM platform. Minor version upgrade should not affect how the platform functions and is highly recommended (we have recently upgraded from version 2.2.5 to 3.0.3 without any problem).

In the project root directory, the **startproject** command creates a subdirectory with identical name. This is the actual Python package for your project. Within it are project configuration files which tell Django how to operate the Web service. Our configurations mostly involve editing *mysite/settings.py* and *mysite/urls.py*. Most of the changes to *mysite/settings.py* involve data models (each model defines a table in MySQL database) and authentication/authorization system. The *urls.py* forwards HTTP requests to target views (as in MTV) based on the URL provided.

Django defines each MVT package in a project as an app. For example, running **python manage.py startapp accounts** will create an app (as a Python package) named *accounts*. Following Django's convention, you may define a *User* model in *accounts/models.py* and Django would create a database for *User* in MySQL database upon migration (using **python manage.py makemigration accounts** then **python manage.py migrate accounts**). You then create a view as a function in *accounts/views.py* to process an HTTP request and render a response displaying user information. Finally, you add *accounts/urls.py* then wire it to *settings/urls.py*, and all HTTP requests with URLs matching the rule would be forwarded to your target view.

We have just walked through a very brief introduction of how to create a website with a database backend using Django. For a detailed step-by-step tutorial on how to create a simple Django website with associated data models, visit *https://docs.djangoproject.com/en/3.0/intro/tutorial01/*. Now we will use Django REST Framework to turn a simple Web service into a RESTful API service.

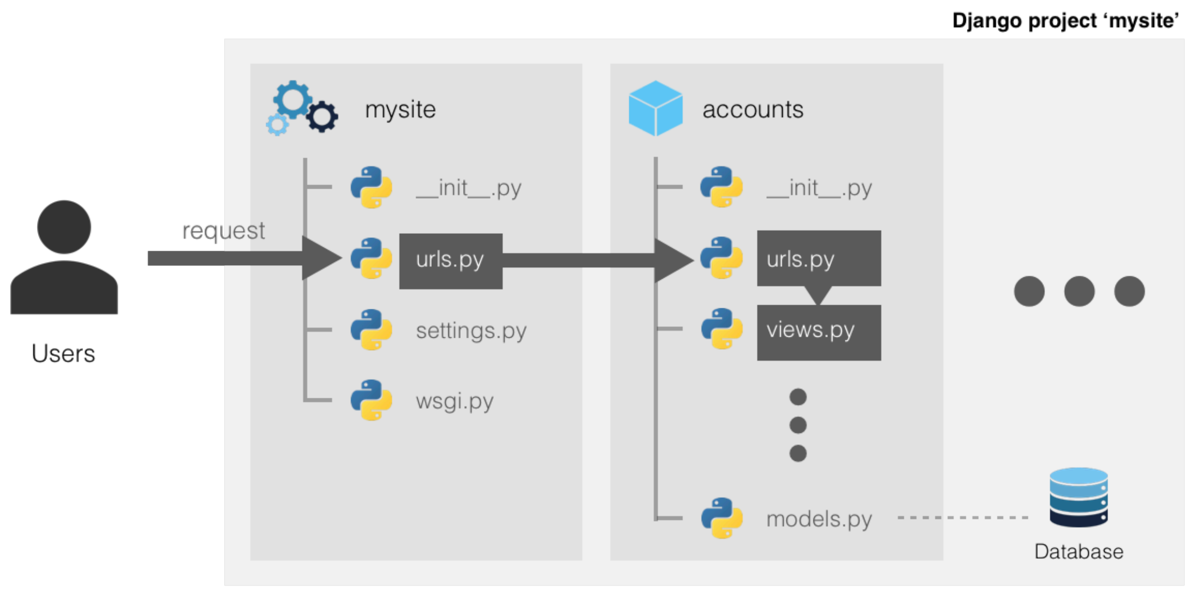


Figure 2.1 Django project structure where a request from user is forwarded to the target view.

Django REST Framework

REST defines architectural constraints which is used for creating Web services that allows interoperability between computer systems. In the UWKGM platform, we implement HTTP methods *GET*, *POST*, *PUT*, *PATHCH*, and *DELETE* and render a response with a payload formatted in JSON. We use Django REST Framework as part of request processing, response rendering processes, and data serialization. The framework also includes authentication mechanism, Web browsable API and more.

The following is a code example of an API endpoint that returns a greeting message:

```
mysite/accounts/views.py
>>> from rest_framework import status
>>> from rest_framework.decorators import api_view
>>> from rest_framework.request import Request
>>> from rest_framework.response import Response
>>>
>>> @api_view(['GET'])
>>> def greet(request: Request) -> Response:
...     return Response({detail: 'Hello %s! ' % request.GET.name},
                              status=status.HTTP_200_OK)
```

After setting request forwarding rules in *mysite/mysite/urls.py* and *mysite/accounts/urls*.py, a GET request sent to *http://.../accounts/greet?name=Django* should return:

```
HTTP_200_OK
. {
.   "detail": "Hello Django!"
. }
```

Another powerful tool that Django REST Framework offers is serializers. Serialization packs the entire process of creating views, forwarding requests to the views, binding the views to a data model, and render a response in JSON format. With a simple script, you can create a *ViewSet* associated with a data model such as *User* and create API endpoints that list, add, edit, and delete user accounts. The following API endpoint allows user accounts to be viewed or edited:

```
mysite/accounts/views.py
>>> from django.contrib.auth.models import User
>>> from rest_framework import viewsets
>>> from accounts.serializers import UserSerializer
>>>
>>> class UserViewSet(viewsets.ModelViewSet):
...   queryset = User.objects.all().order_by('-date_joined')
...   serializer_class = UserSerializer
```

After adding a URL parsing rule, a GET request to *http://.../accounts/* should list the user accounts, a POST request to *http://.../accounts/* should create a new account, and so on.

Dorest: Extending Django REST Framework

Some endpoint functions in the UWKGM platform rely on other endpoint functions to process their requests. One way to make them work is to build two sets of codes – one consists of logical functions and another contains API endpoint that call the former. The reason that endpoint functions cannot call each other directly is because Django REST Framework architectural design dictates that they must define a positional parameter that accepts an instance of **rest_framework.request.Request**; they must also return an instance of **rest_framework.response.Response**. The request's payload is accessible using **Request.GET** or **Request.data**, which return a dictionary of the payload.

Our goal of extending the framework is to eliminate the parameter of type **Request** from the endpoint functions and allow them to define their actual parameters and return type. They should behave the same way the endpoint functions conforming to the framework do, meaning that the framework's decorators such as access control and throttling (rate limiting) should still work. For example, by using our extension *Dorest*, the greeting endpoint that we explained earlier would become:

```
mysite/accounts/views.py
>>> from dorest import endpoint
>>>
>>> @endpoint(['GET'])
>>> def greet(name: str) -> str:
...     return 'Hello %s!' % name
```

This revised endpoint **greet** returns the same result as its earlier version does. The difference is that any Python functions can also call **greet** directly. With this new architecture, we no longer need to create the two sets of codes and endpoint functions can now call each other directly.

Redefining Endpoints

We have encountered some challenges making a regular Python function an API endpoint that works with Django REST Framework. The first challenge is to convert

Request into valid arguments before passing them on to the function; or if the arguments sent from the caller is not of type **Request**, i.e. the call is a direct call, there should not be any conversion.

The function that determines whether the parameters should be converted is function **wrapper** nested in decorator **endpoint** in **dorest.managers.struct.decorators**. If the caller function sent only one parameter of type **Request**, then it would try to extract the payload of the **Request** instance and store all parameters in variable **parameters**.

In some cases where an endpoint function needs other information from the **Request** instance (e.g. which user sent the request), you can tell Dorest to pass it as one of the parameters by setting the decorator's **include_request** to the name of the parameter where you want Dorest to pass the instance. For example:

```
>>> @endpoint(['GET'], include_request='django_request')
>>> def greet(name: str, django_request: Request) -> str:
...     return '%s says hello to %s' % (str(Request.user), name)
```

Now that we can determine whether or not to convert the parameters, our next challenge is parsing them. The parsing process is written in method **parse** of class **Endpoint** defined in **dorest.managers.struct.describe**.

As Dorest received a request from a user, it determines and examines the target endpoint function then stores the function's metadata, i.e. the information that describes the function, as an instance of **Endpoint**. Next, it attaches the metadata to the **Request** instance by creating an attribute named **META**. Now that the request from the user has the description to help the parser parse the parameters, Dorest is ready to send it to function **wrapper** of decorator **endpoint**.

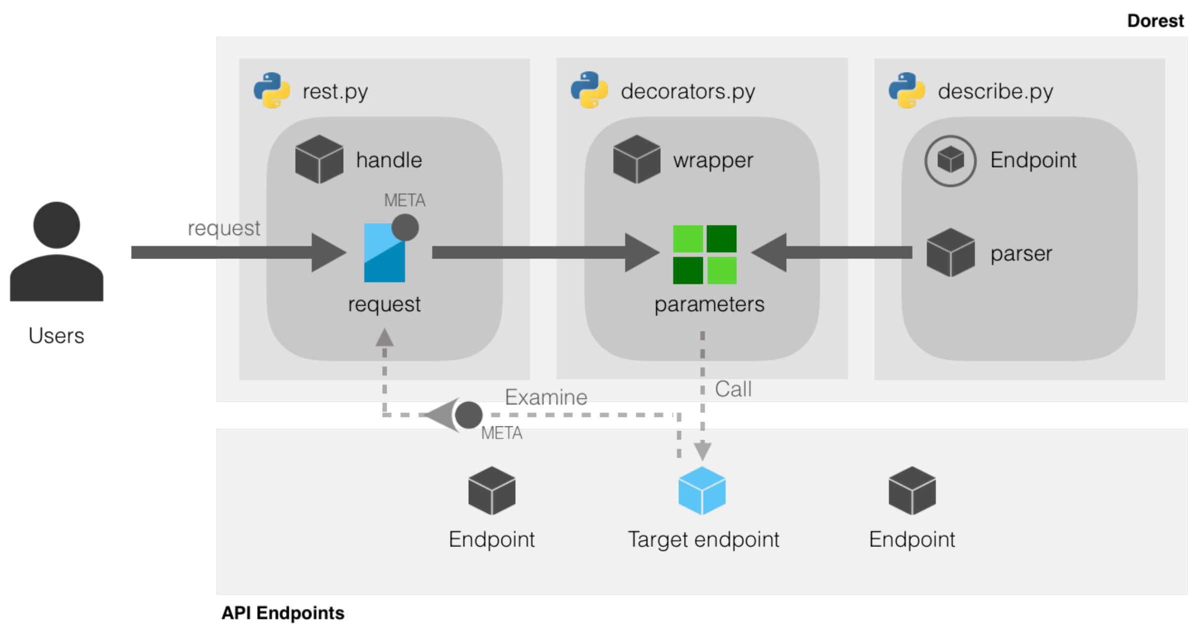


Figure 2.2 Parameters parsing process

As function **wrapper** received the request and decided that the parameters must be parsed before it can call the actual endpoint function, the **wrapper** calls method **parse**. The method first tries to parse the parameters using the metadata; if it cannot find the metadata for any parameters, it will try to guess and parse them.

An **Endpoint** instance not only stores the type of parameter for parsing, but its description and example (if provided) as written in the function's docstring as well. The instance also store metadata describing the function itself and the returning value. Function **greet** with a docstring would look like this:

```
>>> @endpoint(['GET'])
>>> def greet(name: str) -> str:
...     """Say hello to someone
...
...     :param name: Someone's name
...             :ex: Steve
...     :return: A greeting message
...     """
...
...     return 'Hello %s!' % name
```

Upon creation, an **Endpoint** instance **e** of function **greet** would be:

```
. e.func = <function greet at {address in memory}>
. e.description = "Say hello to someone"
. e.args = [Param(name = "name",
                  annotation = "str",
                  description = "Someone's name",
                  example = "Steve",
                  default = None)]
. e.kwargs = []
. e.returns = namedtuple(annotation = "str",
                         Description = "A greeting message")
```

All of the information above is useful for automatic generation of the endpoint's description in JSON format. We will revisit this part of Dorest in the *Structured Endpoint* section to explain how all of this information can be useful in practice. Note that class **Param** is also defined in **dorest.managers.struct.describe**.

Structured API Endpoints

As an endpoint function can now process both requests sent via RESTful API and direct calls, we no longer need to write two sets of codes to facilitate our complex API structure. However, we still need to write sets of rules that forward the requests to our endpoints, and any changes in the endpoint structure (adding, editing, renaming, moving, or deleting files or functions) would require us to revise the rules.

With Dorest's *Structured Endpoints*, we aim to eliminate the task of writing and editing the forwarding rules whenever the endpoint structure changes, reducing the workload in maintaining the service. The idea is that you can make your endpoints available simply by put them in nested Python packages and tell Dorest where the root package is located. Dorest will then associate the URL of each endpoint with its location relative to the root package.

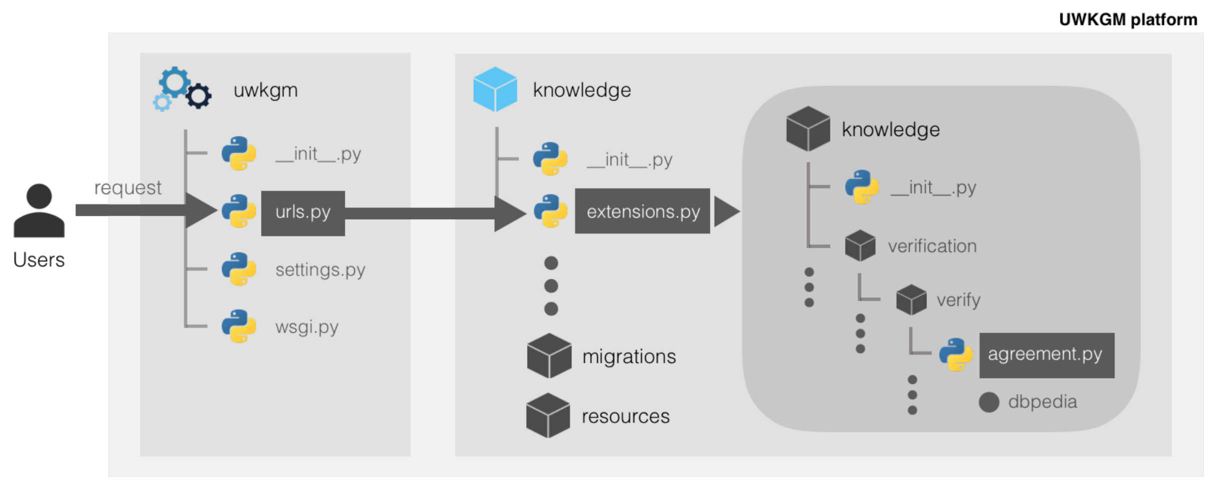


Figure 2.3 Request handling in *Structured API Endpoints*

Figure 2.3 illustrates part of the UWKGM platform where one of its endpoint functions is located. As shown in the flow, Django forwards the user's request according to the rule defined in *urls.py*, which points to *extensions.py*:

```
uwkgm/uwkgm/urls.py
>>> urlpatterns = [..., path('%s/knowledge/' % prefix,
                         include('knowledge.extensions'))]
```

Suppose the domain name is *https://uwkgm.nii.ac.jp/* and *prefix* = '*api*', any requests with URL starting with *https://uwkgm.nii.ac.jp/api/knowledge/* will be handled by Dorest's *Structured Endpoints* manager as indicated in the *knowledge* app:

```
uwkgm/uwkgm/knowledge/knowledge/extensions.py
>>> from dorest.managers import rest
>>>
>>> rest.forward(r'.*',
                 at=__name__,
                 to='knowledge.knowledge',
                 using=rest.Managers.STRUCTURED_ENDPOINTS)
```

The command **forward** forwards all URLs (as indicated by the regular expression *r'.*'*) from the *extensions* module (*at=__name__*) to the root of target structured endpoints (*to='knowledge.knowledge'*) using Dorest's *Structured Endpoints* manager.

Now, since the target endpoint in Figure 2.3 is function **dbpedia** in *agreement.py*, the URL for this endpoint should be *https://uwkgm.nii.ac.jp/api/knowledge/verification/verify/agreement/dbpedia*.

If the given URL does not include any specific endpoint function within the target module (the URL in previous paragraph without */dbpedia*), Dorest will look, from top to bottom, for the first valid endpoint function it could find. Thus, if **dbpedia** is the top endpoint function, Dorest will regard it as the target endpoint.

The command **forward** operates somewhat like a wrapper for Django's *urlpatterns* variable, in which you define a set of rules for URL forwarding. Normally when you define the *urlpatterns* variable in *urls.py*, Django reads the variable and call the target *view* function according to the request it has received. It should also be reasonable to consider *urlpatterns* as an attribute of a module where it resides. Therefore, what command **forward** do is essentially defining *urlpatterns* attribute for Django to make its decision as to where it should forward the requests, but in the way that fits Dorest's architecture.

Dorest separates the structured endpoints from other scripts in the app where the endpoints belong, establishing a clear boundary of the structure. In fact, you can put your structured endpoints anywhere outside your app as long as Python can reach their location. The structured endpoints can also work together with scripts that follow Django REST Framework architectural style using our command **extend**. For example, you could add the following command to *extensions.py* before command **forward**:

```
>>> rest.extend(r'ext', at=__name__, to='uwkgm.knowledge.urls')
```

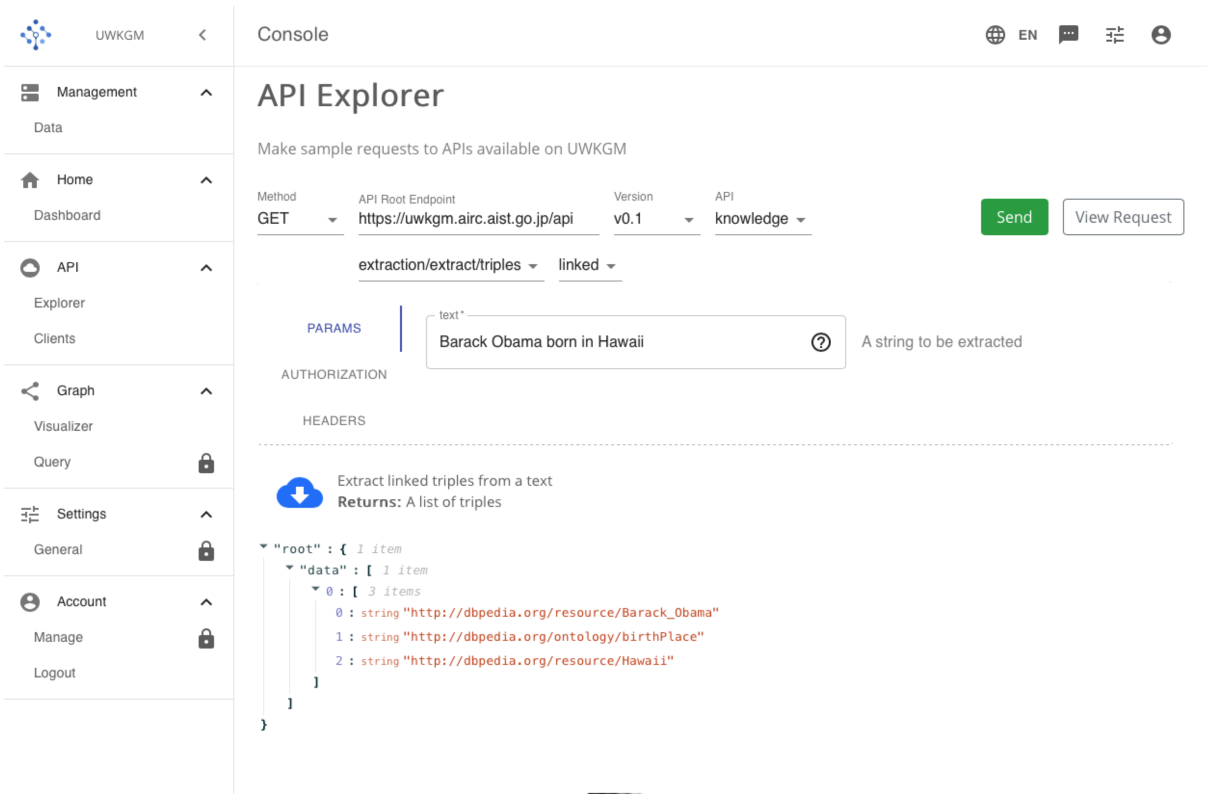
With command **extend**, you can add a dynamic URL rule for the *knowledge* app such as *https://uwkgm.nii.ac.jp/api/knowledge/ext/{variable}*, while still maintain your structured endpoints.

The requests sent to *Structured Endpoints* are handled by function **handle** in **dorest.managers.struct.rest**. Depending on the URL pattern, function **handle** may execute the endpoint function or return a description of single/multiple endpoints. GET requests containing parameter ** are queries regarding the entire structure or substructure of a given *Structured Endpoints*. For example, Dorest would return a structure of all endpoints in **uwkgm.knowledge.knowledge.verification** should it received a GET request with URL *https://uwkgm.nii.ac.jp/api/knowledge/verfication ?***. To get a description of any particular endpoint, use parameter *, for example:

```
GET https://uwkgm.nii.ac.jp/api/knowledge/extraction/
    extract/triple/linked?*
HTTP_200_OK
. {
.   "help": {
.     "name": "linked",
.     "methods": ["GET"],
.     "traceback": "extraction.extract.triples.linked",
.     "description": ["Extract linked triples from a text"],
.     "arguments": {
.       "required": [
.         {
.           "name": "text",
.           "annotation": "<class 'str'>",
.           "description": "A string to be extracted",
.           "example": "Barack Obama born in Hawaii",
.           "default": ""
.         }
.       ],
.       "optional": []
.     },
.     "return": {
.       "annotation": "",
.       "description": ["A list of triples"]
.     }
.   }
. }
```

Our Web-based interactive API Explorer, shown in Figure 2.4, utilizes these JSON-formatted descriptions of an endpoint and structured endpoints.

The description above is a JSON render of an **Endpoint** instance generated by the class's method **rest**. Once function **handle** in **dorest.managers.struct.rest** received a request, it will check first whether the client asked for the description of structured endpoints; if not, it will try to locate the target endpoint function from the URL using function **_get_endpoint** in the same module. If the endpoint is a valid function, it will either return the description or execute the function, depending on the request.



API Explorer

Make sample requests to APIs available on UWKGM

Method: GET
API Root Endpoint: https://uwkgm.airc.aist.go.jp/api
Version: v0.1
API: knowledge
extraction/extract/triples
linked

text*: Barack Obama born in Hawaii
A string to be extracted

Extract linked triples from a text
Returns: A list of triples

```
"root" : { 1 item
  "data" : [ 1 item
    0 : [ 3 items
      0 : string "http://dbpedia.org/resource/Barack_Obama"
      1 : string "http://dbpedia.org/ontology/birthPlace"
      2 : string "http://dbpedia.org/resource/Hawaii"
    ]
  ]
}
```

Figure 2.4 The UWKGM platform’s API Explorer

To execute the endpoint function, function **handle** use Django REST Framework’s **api_view** decorator to make sure that other decorators that belong to the framework and being applied to the endpoint function work properly. To clarify, the framework’s decorators add certain attributes to the endpoint function and **api_view** reads these attributes to modify the endpoint’s behavior. For instance, the decorator **permission_classes** adds **permission_classes** attribute to the endpoint function to attach permission policies. Later as the framework processes the request, it will check whether the user who sends the request has the permissions required to access the endpoint before executing the function.

As it looks for the target endpoint function, function **_get_endpoint** may redirect its search to another location if a redirecting attribute has been set for the module it was examining. For example, the following code in **uwkgm.knowledge.knowledge.extraction.extract** redirects any GET requests with URL *https://uwkgm.nii.ac.jp/api/knowledge/extraction/extract* to **uwkgm.knowledge.knowledge.extraction.extract.triples**:

```
uwkgm/knowledge/knowledge/extraction/extract/__init__.py
>>> from dorest.managers.struct import rest
>>> from . import triples
>>>
>>> rest.redirect(methods=['GET'], at=__name__, to=triples)
```

And since **triples** is a module, function **_get_endpoint** would look for the top endpoint function in *triples.py* and, therefore, regard function **linked** as the target endpoint.

Resource Management

In addition to Dorest's *Structured Endpoints*, you can opt for its structured resource manager. The manager stores resources in a different location than the structured endpoints but their structures are identical.

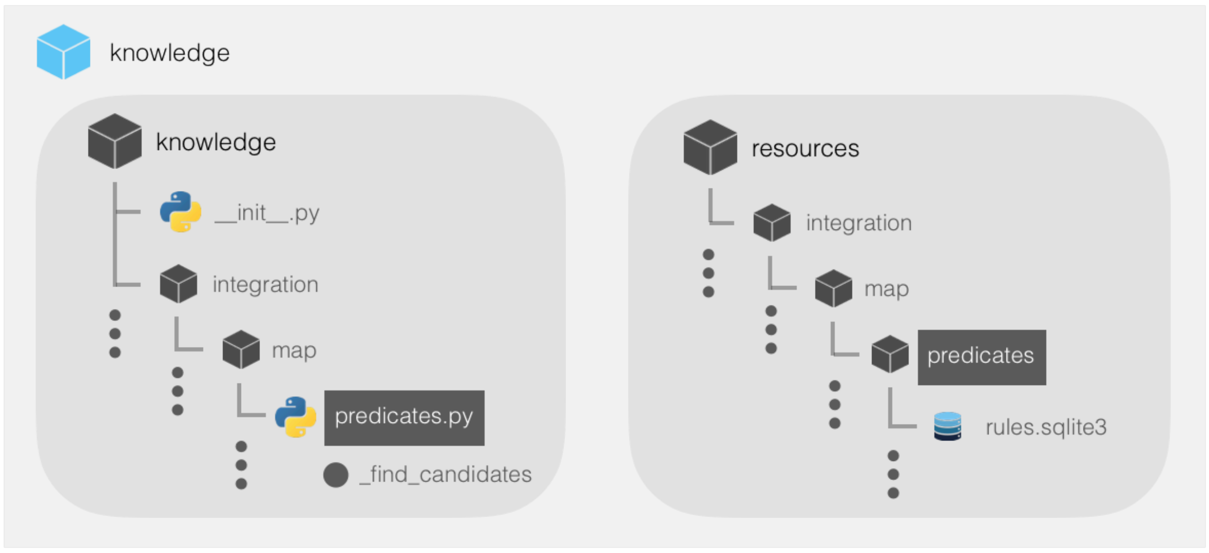


Figure 2.5 Endpoint and resource structures

The app *knowledge* in the UWKGM platform uses the structured resource manager. As shown in Figure 2.5, the app contains packages *knowledge* and *resources*. In this example, function **_find_candidates** in *predicates.py* requires access to a database file *rules.sqlite3*. The function reads the file using the resource manager as followed:

```
uwkgm/knowledge/knowledge/integration/map/predicates.py
>>> from dorest.managers.struct import resources
>>>
>>> def _find_candidates(predicate: str) -> List[Tuple[str, str]]:
...     rules_db_conn = sqlite3.connect(resources.resolve(__name__,
                                          'rules.sqlite3'))
```

Function **resolve** takes a full name of function or module as its first argument (e.g. **__name__** in the above example is `'uwkgm.knowledge.knowledge.integration.map.predicates'`) and determines the root location of the target resource files (folder *predicates* in Figure 2.5). The second parameter is the name of the file (which may include the subdirectory where it resides).

In case you wanted to store resources separately for each function, you could create subdirectories within the directory associated with the module. For example, you could create a folder named *_find_candidates* in folder *predicates*, move *rules.sqlite3* to the subfolder, then replace the first parameter of function **resolve** as followed:

```
>>> def _find_candidates(predicate: str) -> List[Tuple[str, str]]:
...     rules_db_conn = sqlite3.connect(resources.resolve(
                                          _find_candidates,
                                          'rules.sqlite3'))
```

To associate a directory with structured endpoints, call function **register** in *__init__.py* located in the endpoints' root directory:

```
uwkgm/knowledge/knowledge/__init__.py
>>> from dorest.managers.struct import resources
>>>
>>> resources.register('knowledge.resources', to=__name__)
```

Function **register** sets attribute **__resources** to *__init__.py*. The attribute contains the name of the resource package given as the first parameter to function **register**. When you call function **resolve**, it searches for the attribute in the parent packages of the given module or function, then use the attribute to locate the location of the resources.

<u>Generic API</u>

The generic mechanism facilitates multiple implementations of a logic. When we designed the UWKGM platform, our goal was for it to be able to work with multiple database management systems (DBMS). The platform should allow you to switch to any other systems of your choice. While it is simply not feasible for us to develop and update drivers for all DBMS, Dorest's *Generic API* allows anyone to build drivers that work with whatever DBMS they prefer to use.

Support for multiple DBMS is one use case of *Generic API*. In brief, it operates somewhat like a switch: You have multiple implementations of the same logic; which one to use depends on your configuration. For example, in the UWKGM platform, app *database* uses *Generic API* for graph database management. Two packages in the app are involved: *database* and *drivers*:

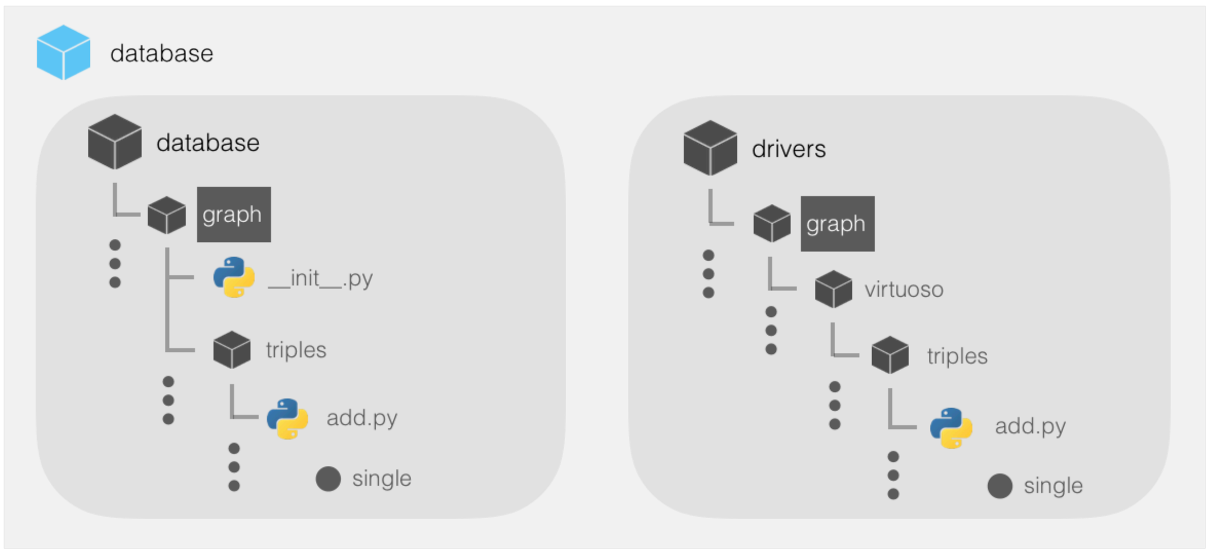


Figure 2.6 Graph database management using Generic API

Package *database* in Figure 2.6 contains a collection of generic APIs, including *graph*. We registered the location of the package containing our implementation (graph database drivers) in *__init__.py* using function **register**:

```
uwkgm/database/database/graph/__init__.py
>>> from dorest.managers.struct import generic
>>>
>>> generic.register('database.drivers.graph',
                     to=__name__,
                     using='virtuoso')
```

Function **register** set attribute **__generic** to *__init__.py*. The attribute stores the location of the package containing the implementations (`'database.drivers.graph'`) and the implementation being used (`'virtuoso'`). The root directory of the implementation package contains implementation names (currently only *virtuoso* is available). Similar to *Structured Endpoints*, the structure of package *virtuoso* (*database.drivers.graph.virtuoso*) should match that of its generic package (*database.database.graph*).

To execute its implementation, a generic function must call function **resolve**:

```
uwkgm/database/database/graph/triples/add.py
>>> from dorest.managers.struct import generic
>>> from dorest.managers.struct.decorators import endpoint
>>>
>>> @endpoint(['GET'])
>>> def single(triple: Tuple[str, str, str]) -> str:
...   return generic.resolve(single)(triple,
                                      os.environ['UWKGM_GRAPH'])
```

Function **resolve** takes one argument, which can be the generic function itself or its full name (e.g., `'database.drivers.graph.triples.add.single'`). Function **resolve** then locates the implementation function by looking for attribute **__generic** in the generic function's parent packages. Once found, it returns the target function, which can be called directly.

Configurations and Environment Variables

Dorest reads JSON or YAML configuration files from a location specified in attribute **DOREST** in Django project's *settings.py*. In addition to basic configurations, you can create environment-based configuration files in a different location. The locations of configuration files for the UWKGM platform are as followed:

```
uwkgm/uwkgm/settings.py
>>> BASE_DIR = os.path.dirname(os.path.dirname(
               os.path.abspath(__file__)))
>>> DOREST = {
>>>   'CONFIGS': {'PATH': '%s/configs' % BASE_DIR},
>>>   'ENV': {
>>>     'PATH': '%s/env' % BASE_DIR,
>>>     'NAME': 'UWKGM_ENV'
>>>   }
>>>   ...
>>> }
```

Folders *configs* and *env* in the project's root directory contains YAML files. Dorest reads and stores them in a memory where they can be accessed directly using function **resolve**. For example, *core.yaml* contains the following configuration:

```
uwkgm/configs/core.yaml
. api:
.   name: uwkgm
.   title: UWKGM
```

All configuration files are stored in a memory as a dictionary, with first level keys matching the filenames and values containing configurations in the files. The above example translates into the following dictionary:

```
. {
.   "core": {
.     "api": {
.       "name": "uwkgm",
.       "title": "UWKGM"
.     }
.   }
. }
```

To access **title** anywhere in the project, call function **resolve**:

```
>>> from dorest import configs
>>>
>>> configs.resolve('core.api.title')
UWKGM
```

The function returns a value (e.g. given parameter `'core.api.title'`) or a dictionary (e.g. given parameter `'core.api'`).

Environment-based configuration works similarly to basic configuration, which we explained earlier, except one additional layer that separates configurations based on the environment variable specified in *settings.py*. For example, our configuration for database endpoints depends on environment variable **UWKGM_ENV**:

```
uwkgm/env/database.yaml
. development:
.   mysql:
.     address: localhost
.     port: 3306
.
. production:
.   mysql:
.     address: uwkgm-mysql-service
.     port: 3306
```

For all environment-based configuration files, their top-level keys (`'development'` and `'production'` in the above example) are the names of the environments. You can access the environment-based configuration using function **resolve**:

```
>>> from dorest import env
>>>
>>> env.resolve('database.mysql.address')
```

Suppose we set **UWKGM_ENV** = `'development'`, the function call above should return `'localhost'`. Its parameter is similar to that of function **resolve** in **configs**, with the first element pointing to the configuration filename. The top-level keys are omitted as they are environment names, so the second element points to the second-level key and so on.

Variables and Functions in Configurations

Dorest defines additional syntax for configuration files that allows access to variables and functions. There are two types of variables: relative and predefined, and one type of functions: predefined. Relative variables give you access to values stored in other parts of the configuration. For example, we want the UWKGM platform to send registration confirmation emails from `'register@uwkgm.com'`, we set the platform's configuration as followed:

```
uwkgm/configs/core.yaml
. api:
.   name: uwkgm
.   domain: com
.
. email:
.   domain: "{{core.api.name}}.{{core.api.domain}}"

uwkgm/configs/accounts.yaml
. registration:
.   email: "register@{{core.email.domain}}"
```

Relative variables are wrapped in double curly brackets (e.g. `"{{core.api.name}}"`) and point to specific part of the configuration dictionary the same way the parameter of function **resolve** in the previous section does.

Predefined variables and functions are Dorest's off-the-shelf feature. The variables are in *vars.py* and the functions are in *funcs.py*; both files are in **dorest.configs**. Similar to relative variables, predefined variables and functions are wrapped in double curly brackets. However, the opening brackets must be followed by $, then the variable/function name (e.g. `"{{$base_dir}}"`, `"{{$env('name': 'PYTHONPATH')}}"`). The variable **base_dir** refers to attribute **BASE_DIR** in the project's *settings.py*.

For predefined functions, the syntax for their arguments is similar to that of a dictionary, except that you do not need paired curly brackets inside the function's paired round brackets. Common functions are **env(name: str)**, which retrieve the value of given environment variable, and **envyml(var: str)**, which operates like function **resolve** in **dorest.env**.

Chapter 3

Platform Architecture and Implementation

Collaboration is of the essence of the UWKGM platform. Its architectural design prioritizes human involvement, maximizes transparency and quality, and facilitates controlled access. The platform's ability to keep track of the changes applied to the knowledge graph, including who made the change and what method they used to make the change, ensures consistency and validity of the graph. This chapter will walk you through all aspects of the platform, including how they were implemented and how can they be useful in collaborative work.

Overview

The platform has three main components: data management, knowledge graph toolkit, and user management. Data management covers graph databases and all related graph modification records. This backbone component helps knowledge engineers to build and maintain their knowledge graphs. As the graph grows, our toolkit provides a collection of functionalities via Web API for developers and researchers to leverage the power of well-connected knowledge. Lastly, the system administrators have full control over who has access to the tools and who can make changes to the graph.

Many of the platform's functionalities available via Web API are built to support graphic user interface. We built a Web user interface separately from the platform to simplify data management and administrative processes, as well as providing data visualization tool and interactive API Explorer for developers to learn and test our toolkit. As the platform is available via Web API, you can also develop your own user interface to fit the needs of your organization.

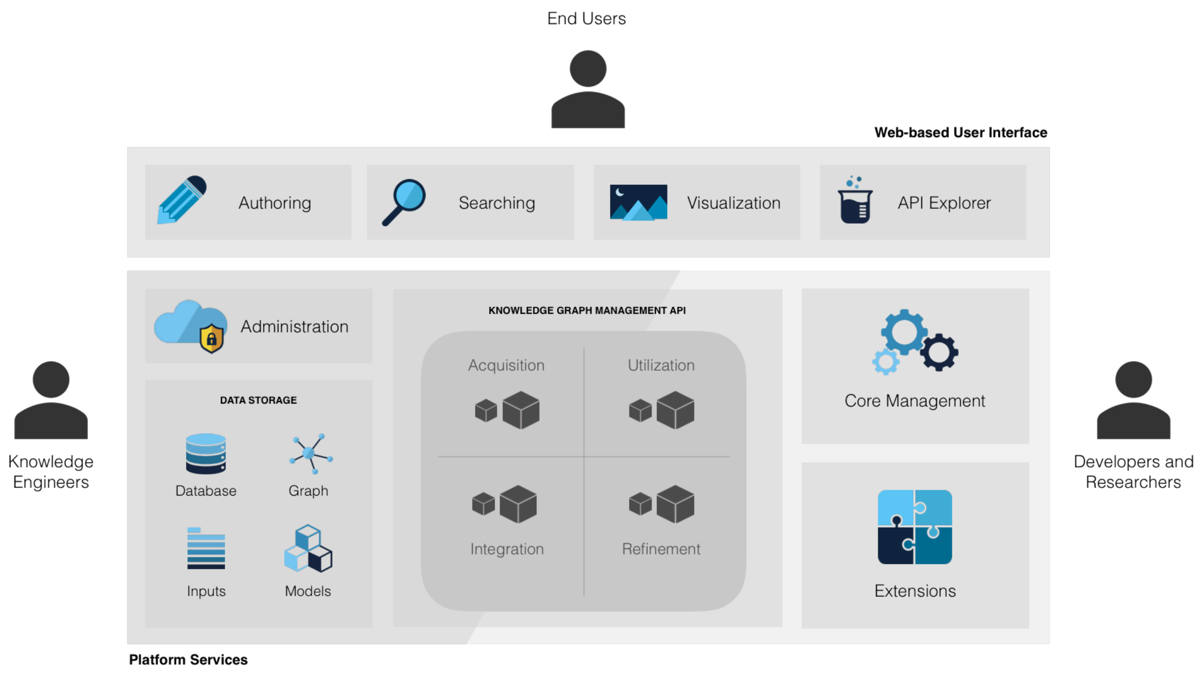


Figure 3.1 Use case diagram of the platform and our user interface

As a Django project, the platform contains three applications (*knowledge*, *database*, and *accounts*) that are connected to three databases:

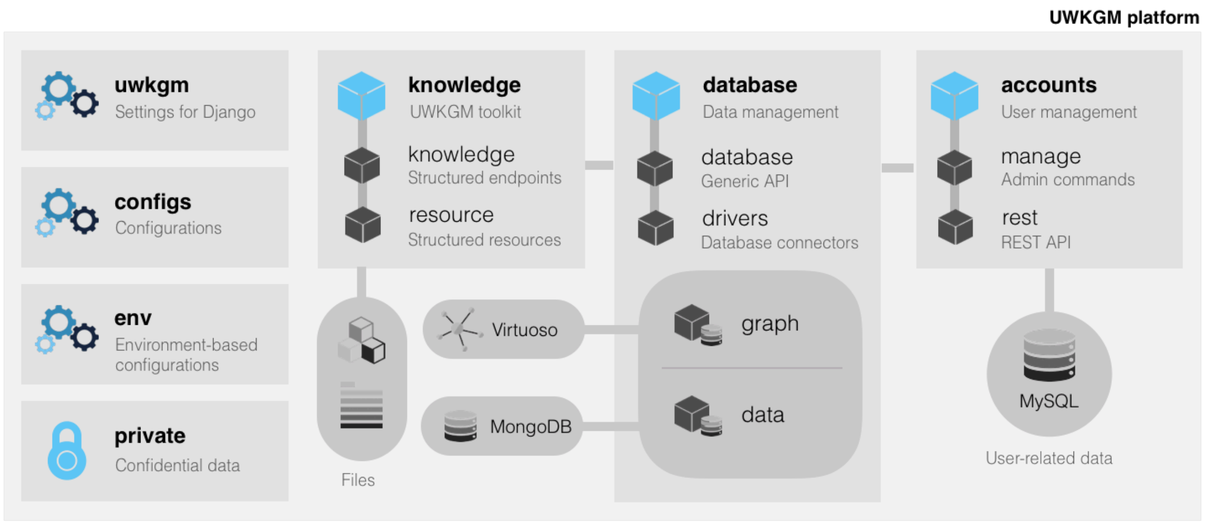


Figure 3.2 Part of the platform as a Django project

Data Management

We use Dorest's *Generic API* to build database connectors that support multiple database backends (depending on available drivers). This section will focus on two connectors: one for graph database and another for graph-modifier database. For each connector, our explanation will start with its logic functions, followed by more details on our implementation of the logics using our choice of database backend. The user management component uses Django's database connector and will be covered in a separate section.

Graph Database

The logics pertaining to a knowledge graph involve managing graph entities and their relationships. The database connector has basic abilities to add and remove these fundamental elements. In addition, some of the tools in the platform's graph toolkit and our user interface require access to the graph database so we implemented their database-related logics as part of the connector. We expect these additional features of the connector to grow in number as the platform expands.

In knowledge graphs, a relationship between two entities is represented by a triple (subject, predicate, object). Each entity has its own Uniform Resource Identifier (URI) adhering to the Resource Description Framework (RDF). For example, a triple that represents relationship between Barack Obama and his birthplace, Hawaii, is:

```
("http://dbpedia.org/resource/Barack_Obama",
 "http://dbpedia.org/ontology/birthPlace",
 "http://dbpedia.org/resource/Hawaii")
```

We use SPARQL[2] as our query language. Most queries contain a set of triple patterns. For example, the following query finds Barack Obama's birthplace:

[2] https://www.w3.org/TR/rdf-sparql-query/

```
. PREFIX    dbr:  <http://dbpedia.org/resource/>
. PREFIX    dbo:  <http://dbpedia.org/ontology/>
.
. SELECT ?place WHERE {
.   dbr:Barack_Obama dbo:birthPlace ?place
. }
```

PREFIX shortens a query that contains entities which appear repetitively. You can replace `dbr:Barack_Obama` in the above example with `<http://dbpedia.org/resource/Barack_Obama>`.

The rest of this section will explain the platform's functions related graph database.

Basic Functions

Find

Task: Find a relationship between two entities

SPARQL Query:

```
SELECT DISTINCT ?predicate WHERE { <subject> ?predicate <object> }
```

Add

Task: Add a triple to the graph

SPARQL Query:

```
INSERT DATA {GRAPH <graph> {<subject> <predicate> <object>}}
```

Delete

Task: Delete a triple from the graph

SPARQL Query:

```
DELETE DATA {GRAPH <graph> {<subject> <predicate> <object>}}
```

Edit

Task: Edit a triple in the graph

Instruction: Delete an existing triple and add a new triple to replace the predicate

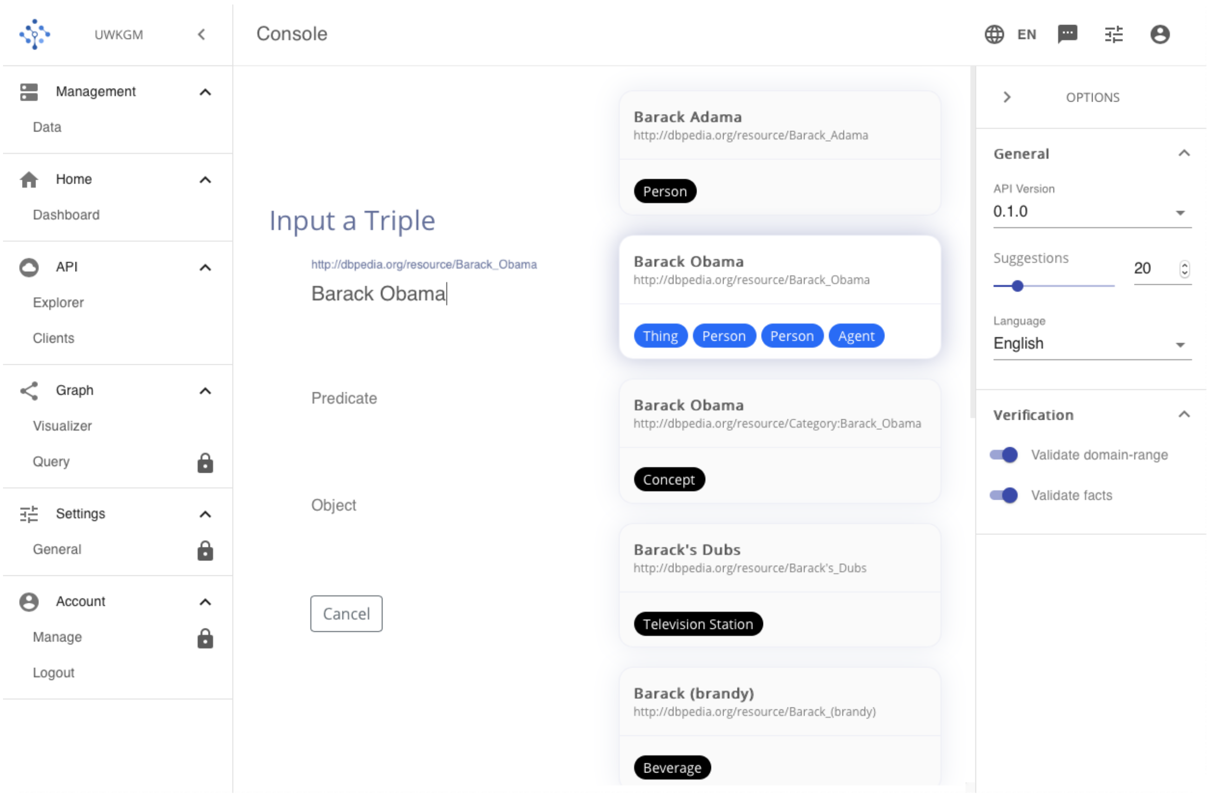
Additional Functions

At the time of writing we had created two additional functions for the graph connector, both supports our Web user interface. The two functions are in **database.database. graph.entities.find** module. The first function, **candidates**, finds candidate entities and relations from a given complete (e.g. Barack Obama) or incomplete (e.g. Bara) string. The function uses graph database's indexing to provide interactive suggestions as users input an entity's label. It returns three types of matches: exact matches, first matches, and partial matches. To find the matches, it performs string search on entities' label using `<http://www.w3.org/2000/01/rdf-schema#label>` (`rdfs:label`) relationship (the predicate of which is a string, e.g., `(dbp:Barack_Obama, rdfs: label, "Barack Obama"^^xsd:string)`. *Exact matches* are entities having a label that matches the search string perfectly. *First match* means the first part of the label matches the search string. *Partial match* does not concern where the match occurs. Our implementation of function **candidates** using SPARQL is based on the following query:

```
. SELECT DISTINCT ?entity ?label ?type WHERE {
.   ?entity rdfs:label ?label; a ?type .
.   ?label bif:contains "search_terms"
.   FILTER (lang(?label) = "en")
. }
```

`bif:contains` is supported by Virtuoso[3], a graph database management software that we use, for full text search on indexed entries. Note that this feature finds *exact matches* and *first matches* much faster than *partial matches*. Therefore, in our implementation, the function tries to find *exact* and *first matches* first. If it could not find one, it would search for *partial matches*.

[3] https://github.com/openlink/virtuoso-opensource



Input a Triple

http://dbpedia.org/resource/Barack_Obama
Barack Obama

Predicate

Object

Barack Adama
http://dbpedia.org/resource/Barack_Adama
Person

Barack Obama
http://dbpedia.org/resource/Barack_Obama
Thing Person Person Agent

Barack Obama
http://dbpedia.org/resource/Category:Barack_Obama
Concept

Barack's Dubs
http://dbpedia.org/resource/Barack's_Dubs
Television Station

Barack (brandy)
http://dbpedia.org/resource/Barack_(brandy)
Beverage

General
API Version: 0.1.0
Suggestions: 20
Language: English

Verification
Validate domain-range
Validate facts

Figure 3.3 Suggestions for entities in our Web UI

We apply **AND** operator on search terms. For example, a search string `"Barack Obama"` would become `"Barack" AND "Obama"` in the query command. Separating terms allows us to perform word-level partial search such as `?label bif:contains "Obama" AND "Bara*"`. This search strategy is particularly useful for fetching suggestions while the user is tying the search string. It also allows the search terms to be in any order.

The second function, **single**, finds triples which a given entity is one of their elements. For example, `single("dbr:Barack_Obama")` returns a dictionary containing three lists of triples with `dbr:Barack_Obama` as their subject, predicate, or object (`{"subject": [...], "predicate": [...], "object": [...]}`). The implementation of **single** in SPARQL is based on the following query (given subject):

```
. SELECT DISTINCT ?predicate ?predicateLabel ?object ?objectLabel
. WHERE {
.   <subject> ?predicate ?object
.   OPTIONAL { ?predicate rdfs:label ?predicateLabel }
.   OPTIONAL { ?object rdfs:label ?objectLabel }
. }
```

Configurations

The *Generic API* package for the UWKGM platform's graph database connector is in **uwkgm.database.database** (see Dorest's *Generic API* for more explanation). The package name is **graph** and its *__init__.py* contains a code that registers package **virtuoso** in **uwkgm.database.drivers.graph** as the selected backend.

We created a client instance from **SPARQLWrapper** in *__init__.py* of the package **virtuoso**. The instance handles communication with a SPARQL endpoint via HTTP protocol. All modules in the package import this instance to access the database. The environment-based configuration for SPARQL endpoint address and port is in *database.yaml* in **uwkgm.env**.

Graph-Modifier Database

The UWKGM platform keeps all records of modifications to the knowledge graph. Basic information on the records includes the modification instruction, the user that issued it, creation time, API version, and committing status. Any registered users can issue a graph modifier but only users with administrative privileges can commit it. Uncommitted modifiers do not affect the graph.

We use MongoDB[4] to store the modifiers and *PyMongo*[5] as the database connector. MongoDB stores data records as BSON documents, which are similar to JSON and Python dictionary. Because of this similarity, PyMongo can convert a dictionary to a BSON document and put it directly in MongoDB database. BSON documents are stored in a *collection*, and *collections* constitute a database. As data in MongoDB has a flexible schema, PyMongo creates a database **uwkgm** and a collection **triples** when a user issues the first modifier. No initialization required during platform setup.

The following is an example of BSON document for *add* modifier:

[4] https://www.mongodb.com
[5] https://api.mongodb.com/python/current/

```
. {
.   "_id": ObjectID("5e1f0de46508c22a3a26615f"),
.   "action": "add",
.   "created": ISODate("2020-02-20T22:04:36.500Z"),
.   "issuer": "foo",
.   "api": "0.1.0.16",
.   "committed": true,
.   "triple": {
.     "subject": {
.       "label": "Ise Grand Shrine",
.       "entity": "http://dbpedia.org/resource/Ise_Grand_Shrine"
.     },
.     "predicate": {
.       "label": "location",
.       "entity": "http://dbpedia.org/ontology/location"
.     },
.     "object": {
.       "label": "Mie Prefecture",
.       "entity": "http://dbpedia.org/resource/Mie_Prefecture"
.     }
.   }
. }
```

For *delete* modifier, the only difference is *action*, which stores `"delete"` instead of `"add"`. For *edit* modifier, another difference is that the document stores *originalTriple* in addition to *triple*.

Configurations

The *Generic API* package for graph-modifier database connector is in **uwkgm. database.database**. The package name is **data** and its *__init__.py* contains a code that registers package **mongo** in **uwkgm.database.drivers.data** as the selected backend.

We created a client instance from **PyMongo** in *__init__.py* of the package **mongo**. The instance handles communication with a MongoDB endpoint. All modules in the package import this instance to access the database. The environment-based configuration for MongoDB endpoint address and port is in *database.yaml* in **uwkgm.env**.

Knowledge Graph Toolkit

The tools are in the platform's app **knowledge** (**uwkgm.knowledge**), which uses Dorest's *Structured Endpoints* (the request URL forwarding command is in the app's *extensions.py*). The structured-endpoint package **uwkgm.knowledge.knowledge** contains the four main UWKGM components, i.e., knowledge extraction, integration, verification, and completion. This section will describe the implementation of the components.

Knowledge Extraction

There are currently two endpoints for this component, each extract certain elements from a text. The first endpoint, **openie**, in **…extraction.extract.triples.raw** uses Stanford CoreNLP's Open Information Extraction (OpenIE)[6] to extract open-domain triples from a text. For example, given a sentence `"Barack Obama was born in Hawaii"`, OpenIE returns two triples: `("Barack Obama", "be", "bear")` and `("Barack Obama", "be bear in", "Hawaii")`. Since OpenIE operates on open domain and with limited or no training data, the result needs to be processed further to eliminate or change non-relevant output before we can add it to a knowledge graph. We created another endpoint **linked** in **…extraction.extract.triples** to partially automate the process.

Function **linked** uses other functions in the toolkit to extract triples from a text, then map the triples to graph entities. For example, given a sentence `"Barack Obama born in Hawaii"`, the function returns:

```
[
  ("http://dbpedia.org/resource/Barack_Obama",
   "http://dbpedia.org/ontology/birthPlace",
   "http://dbpedia.org/resource/Hawaii")
]
```

[6] https://stanfordnlp.github.io/CoreNLP/openie.html

The function takes several steps to link the entities. First, it calls function **openie** to get candidate triples, then function **...integration.map.entities** to find entities in the text:

```
>>> from ...extraction.raw import openie
>>> from ...integration import aggregate, map
>>>
>>> text = "Barack Obama born in Hawaii"
>>> triples = openie(text)
[
  ["Barack Obama", "bear in", "Hawaii"]
]

>>> entities = map.entities(text)
[
  {
    "Barack Obama": "http://dbpedia.org/resource/Barack_Obama",
    "Hawaii": "http://dbpedia.org/resource/Hawaii"
  }
]
```

Next, it aggregates **triples** and **entities** (by replacing each triple’s subject and object with their entities) using function **...integration.aggregate.triples**:

```
>>> aggregated_triples = aggregate.triples(triples, entities)
[
  ("http://dbpedia.org/resource/Barack_Obama",
   "bear in",
   "http://dbpedia.org/resource/Hawaii")
]
```

Finally, it maps the triples’ predicates to their entities using function **...integration.map.predicates**:

```
>>> map.predicates(aggregated_triples)
[
  ("http://dbpedia.org/resource/Barack_Obama",
   "http://dbpedia.org/ontology/birthPlace",
   "http://dbpedia.org/resource/Hawaii")
]
```

Knowledge Integration

The objective of the knowledge-integration component is to map the extracted data into the existing knowledge base. Two modules in package **…integration.map** contains our core functions for the task. The first module, *entities*, maps subject and object of a triple to their associated entities; the second module, *predicates*, does the same for predicates. As there are multiple ways to map the entities and predicates, and research on this topic is still on-going, we decided to create the two modules for mapping functions, with expectation that new functions will be added later as the research progresses.

The method that we use to extract subject and object entities – at the time of this writing – relies on DBpedia Spotlight (Daiber et al., 2013). DBpedia Spotlight performs named-entity extraction on unstructured information. Function **dbpedia** in *entities.py* send a POST request to DBpedia Spotlight endpoint for extraction. The important parameter is **confidence** which we set its default value to .5. The less the value is, the more likely the tool will spot entities, but of course with less confidence.

Function **scoring** in *predicates.py* performs a similar task but with a different method. The function calls another function, **_find_candidates**, to check a predicate against a set of rules in *rules.sqlite3* and produce candidates with associated mapping scores. The candidate that has the highest score and does not violate a domain-range agreement is selected as the linked predicate (see **…verification.verify.agreement.dbpedia** for detailed explanation of the verification method). The SQLite database is stored in a single file and was originally converted from *rules.tsv* (using function **_migrate_rules**).

In addition to the mapping functions in **…integration.map**, we created another function, **...integration.aggreate.triples**, which replaces subject and object of a triple with their entities using a dictionary that function **confidence** produces.

Knowledge Verification

With a large and complex knowledge base such as DBpedia, finding the right entity, especially using automatic tools, is an error-prone process. Verifying domain-range agreement is one way to help reduce the risk of adding erroneous relationships. To illustrate, consider the following relationship:

```
("http://dbpedia.org/resource/Barack_Obama",
 "http://dbpedia.org/ontology/birthPlace",
 "http://dbpedia.org/resource/Hawaii")
```

Now, if we replaced the predicate with `http://dbpedia.org/ontology/standard`, the relationship would not make any sense and, thus, violate a domain-range agreement. RDF schema defines *range*, which can be used to check for domain-range agreement (`http://www.w3.org/2000/01/rdf-schema#range`) between the predicate and object. Our **dbpedia** function in **...verification.verify.agreement** use the schema to verify the agreement based on the following SPARQL query:

```
. SELECT DISTINCT ?validRange WHERE {
.   <predicate> rdfs:range ?range .
.   <object> ?validRange ?range
. }
```

If **validRange** is found, i.e., the result is not empty, then the function concludes that the given triple is valid in terms of domain-range agreement.

User Management

Following Django's user model, we defined two types of users: superuser and regular user. All users need to be active and verify their registered emails to be able to access the platform. Superusers can assign *staff* status to any regular users, giving them the permission to access Django's admin page. Also, while regular users, including ones with *staff* status, can issue graph modification commands, only superusers can commit those commands and make actual changes to the graph.

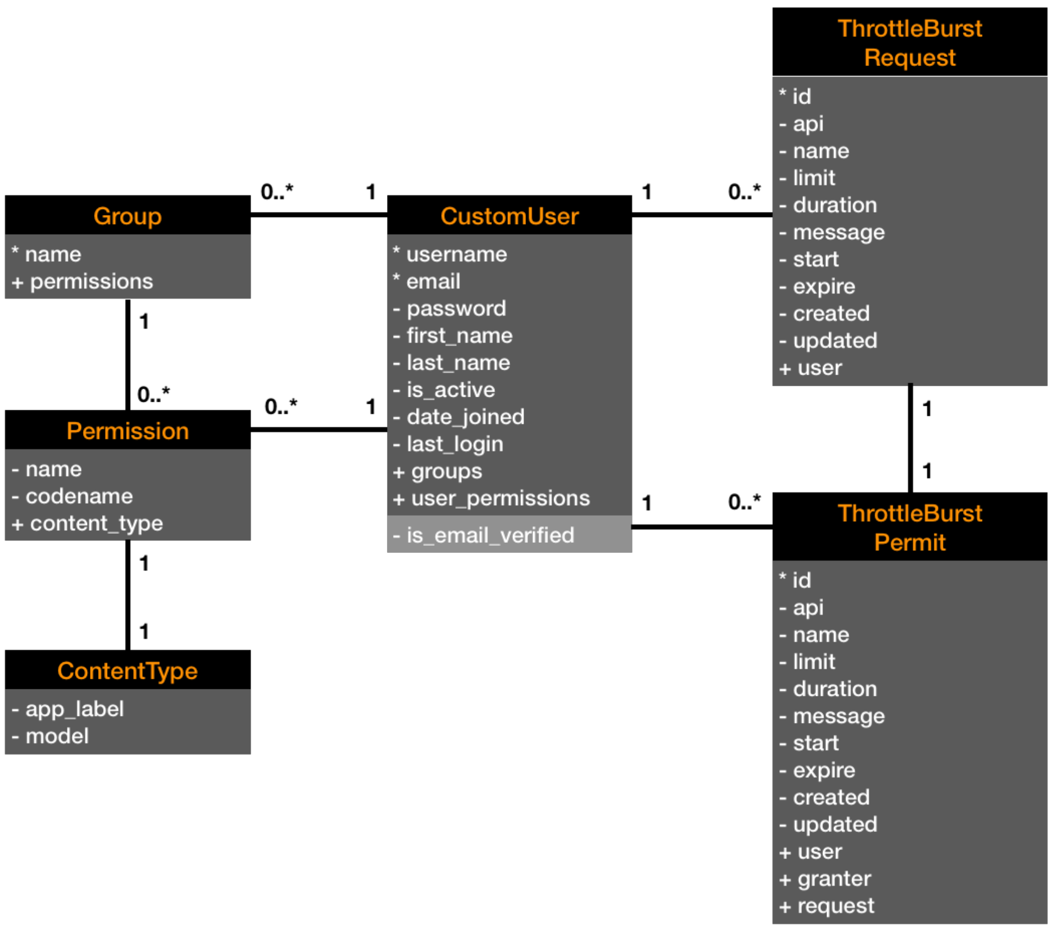


Figure 3.4 Relationships among user model and related models

User Model

In Django, a data model is defined by a Python class which extends **Model** class in **django.db.models**. For user management, Django defines class **AbstractBaseUser** (in **django.contrib.auth.base_user**) based on **Model** and extends the class as **AbstractUser**. We extended **AbstractUser** and named the modified model **CustomUser** (Django has its default class **User**; adding the prefix *Custom* helps clarify that our user data model is different than that of Django).

We use a combination of **email** and **password** for authentication (by setting attribute **USERNAME_FIELD** of class **CustomUser** to `"email"`) but maintain **username** in case users want to change their emails. Both **email** and **username** must be unique. We added **is_email_verified** (Boolean) to keep email verification status and set **is_active** to *false* until the email is verified. A user can be assigned to groups and given permissions. We left **Group**, **Permission**, and **ContentType** unchanged from their definitions in Django.

Serialization

Django REST Framework's serializer simplifies data-bound Web API creation. Instead of creating separate API views (endpoints) to add, edit, delete, view, and list your data from scratch, a serializer helps you to reduce the code required to access and manage a data model. We created a serializer **CustomUserSerializer** in **accounts.rest.serializers.accounts** for our **CustomUser** model. Due to certain limitations of the framework, we needed to write additional code to serialize relational fields (**groups** and **user_permissions**) and **password** (since passwords need to be hashed before they can be stored in the database).

We bound **CustomUserSerializer** to a view set named **UserViewSet** in **accounts.rest.views.accounts**. **UserViewSet** extends the framework's **ModelViewSet** to bound **retrieve**, **create**, **update**, and **destroy** views to our user model serializer. These views are defined as methods, which we overrode to fit our logic (e.g. method **create** sends an activation code via email after user creation). To define user-related API endpoints, we use the framework's router and create **CustomUserRouter**, which extends the framework's **SimpleRouter**, in **accounts.rest.routers**. The router binds HTTP methods to methods defined in **UserViewSet**. Any request URLs forwarded to the app *accounts* and do not match any rules in **accounts.urls** will be handle by the router. For example, suppose the platform's root of API endpoints is *https://uwkgm.nii.ac.jp/api*, to create an account, send the following request:

```
POST https://uwkgm.nii.ac.jp/api/accounts/
. Header:
.   Authorization: Bearer ...
. Form:
.   username=tester
.   email=tester@uwkgm.com
.   password=uwkgm123
```

With a valid authorization code, you should get the following response:

```
HTTP_201_CREATED
. {
.   "username": "tester",
.   "email": "tester@uwkgm.com",
.   ...
. }
```

If the username and email have already been registered, the response should be:

```
HTTP_400_BAD_REQUEST
. {
.   "username": ["A user with that username already exists."],
.   "email": ["user with this Email address already exists."],
. }
```

For development environment, our setting directs Django to save all sent emails in *dev/sent_emails* in the project directory. You can change email settings in the project's *settings.py*.

Registration

In our implementation, only superusers and users that belong to the group *system_user_manager* can create an account. You may build a separate Web server that prevent automatic registration by robots and give the server an account that belongs to the group. Once a user is identified as human, the server can use the information provided by the user to register a new account.

We provide two-step registration, which include user account creation and verification. The previous section explained how to create an account. However, a newly created account will not be active (**is_active** = False) unless its email is verified. Immediately after it created an account, the platform sends an email containing the verification code to the registered email address.

To activate an account, send the following request:

```
PATCH https://uwkgm.nii.ac.jp/api/accounts/{email}/activate
. Header:
.   Authorization: Bearer ...
. Form:
.   code=...
```

If the code is valid, you should get the following response:

```
HTTP_200_OK
. {
.   "detail": "Account activated"
. }
```

Otherwise you will get HTTP_400_BAD_REQUEST for invalid code. If you send the request using HTTP method GET, the platform will resend the activation code to the email specified in the URL. We defined the activation views in class **ActivationViews** (extending the framework's **APIView**) in **accounts.rest.views.activation**. The class also include **Deactivate** which accepts HTTP method PATCH at *https://uwkgm.nii.ac .jp/api/accounts/{email}/deactivate*.

Authentication and Authorization

The UWKGM platform supports basic email/password and token authentication using JSON Web Token (JWT) protocol[7]. Django and Django REST Framework handle basic authentication; but for JWT-based authentication, we use the framework's plugin *Django REST Framework Simple JWT*[8]. JWT is a relatively new standard for securing information exchange. According to the JWT official website:

> "JSON Web Token (JWT) is an open standard (RFC 7519) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. This information can be verified and trusted because

[7] https://jwt.io
[8] https://github.com/davesque/django-rest-framework-simplejwt

it is digitally signed. JWTs can be signed using a secret (with the HMAC algorithm) or a public/private key pair using RSA or ECDSA."

Our settings for *Simple JWT* is stored in a dictionary named **Simple_JWT** in the project's *settings.py*. We use RSA (`"RS256"`) for signing. The functions related to RSA (e.g. password and key generators) are in module **private.rsa**. When you start the platform server, *Simple JWT* will call **get_private_key** and **get_public_key** in **private.rsa** and the keys, together with their password, will be stored in *local/keys/rsa* in the project directory. When you send a request to the platform using JWT authentication code, make sure that the authentication parameter uses `"Bearer"` keyword and not `"Token"` keyword (unless you change **Simple_JWT**'s `"AUTH_HEADER_TYPES" to "Token"`).

Since *Simple JWT* is a plugin for Django REST Framework, it automatically handles JWT-based authentication and provides a set of API views to obtain, refresh, and verify tokens. As JWT tokens allow payload (additional information) to be embedded into a token, we customized one of the plugin's serializers, **TokenObtainPair Serializer**, to add our payload (`"isAdmin": Boolean`) for Web user interface. The custom serializer requires a custom view, which we named **CustomObtainPairView** (extending *Simple JWT*'s **TokenObtainPairView**). We defined the custom view and serializer in **accounts.rest.serializers.jwt**.

To obtain an access token, as well as a refresh token, send the following request:

```
POST https://uwkgm.nii.ac.jp/api/accounts/tokens/obtain
. Form:
.   email={email}
.   password={password}
```

With valid email and password, you should get the following response:

```
HTTP_200_OK
. {
.   "access": {access token},
.   "refresh": {refresh token}
. }
```

Throttling

Rate-limiting is a native feature of Django REST Framework. We defined a set of throttling policies in a dictionary **REST_FRAMEWORK** in the project's *settings.py*. By default, `"base"` policy is applied to every structured endpoint. To apply a different policy on any particular endpoint, set *throttle* parameter of the decorator **endpoint** to the policy name, for example:

```
>>> from dorest.managers.struct.decorators import endpoint
>>>
>>> @endpoint(['GET'], throttle='double')
>>> def hello(name: str) -> str:
...   return 'Hello %s!' % name
```

Suppose the policies in *settings.py* are:

```
>>> REST_FRAMEWORK = {
>>>   ...
>>>   'DEFAULT_THROTTLE_RATES': {
>>>     'base': '10/min',
>>>     'double': '20/min'
>>>   }
>>> }
```

The above setting means that any user can call the endpoint **hello** up to 20 times per minute before getting the following response:

```
HTTP_429_TOO_MANY_REQUESTS
. {
.   'detail': 'Request was throttled.
                Expected available in 59 seconds.'
. }
```

The framework defined throttling classes **UserRateThrottle**, which limits API calls made by a given user, and **ScopedRateThrottle**, which limits API calls by different amounts for various parts of the API. For the former, the policy may be enforced differently on different users, but all endpoints are subject to the same policy. The latter class allow different policies among endpoints, but all users are subject to the same policy for each endpoint. Our throttling class **UserScopedRateThrottle** (extending **ScopedRateThrottle**) combines the flexibilities of both classes mentioned earlier: We allow different policies to be enforces in different parts of the API, and each user may request higher limits.

To make our class work, we created two data models **ThrottleBurstRequest** and **ThrottleBurstPermit** in **accounts.rest.models.throttling**. Users can make multiple throttle-burst requests, each specifies its validity period, desired rate, purpose, among other. Staffs and superusers can grant permits, which may have the same detail as the requests or may be different. When a user makes a request, the platform will check whether the request is within the limit. If not, it will look for permits given to the user and apply one with the highest allowance.

We apply our class to the platform using Dorest's *Structured Endpoint*. If you want to change the throttling class, set your desired class to `"DEFAULT_THROTTLE_CLASSES"` of the dictionary **DOREST** in the project's *settings.py*.

Configuration and Initialization

User account configuration is stored in *configs/accounts.yaml* in the platform's Django project. Part of the configuration file includes, but not limited to, reserved names that cannot be used as a username, a list of groups, a setting for registration verification email, predefined users, and output path for the users' passwords (which would be particularly useful if you use Dorest's predefined function **random_password** to generate the passwords). The list of predefined users should contain at least one superuser, otherwise you will not be able to access the system with administrative privilege.

We only defined one group, which is *user_regular*. You can create other groups to according to your requirements (e.g., *user_trusted* for users with rate-limit exemption); although you may also need further modification to the platform to enforce your policy (e.g. editing throttling class **UserScopedRateThrottle** in **accounts.rest.throttling** to bypass rate-limit check if a user belonged to the group *user_trusted*).

We explained earlier about our serializer for user model **CustomUserSerializer**. In module **accounts.rest.views.accounts**, you will also find another serializer, **Custom UserLimitedSerializer**, which limits user information being serialized. The limited serializer can be useful for redacting sensitive or unnecessary information such as id and password. In *configs/accounts.yaml* file, we defined the setting `"fields"` for the serializers; within it are configurations for **CustomUserSerializer** (`"full"`) and **CustomUserLimitedSerializer** (`"limited"`). Each of the configurations is applied to subclass **Meta** of its serializer.

Extensions

Extensions include services that cannot be implemented directly in the platform. These services may be built using a non-Python language; they may be an executable binary file; or, in many cases, they may be a Web API that cannot be integrated into the platform. Some of the platform's functions requires these extensions to operate; thus, it may be necessary for you to run these services if you need to make those functions available.

In this section we will explain the only extension that we developed. Note that while database management services are not part of the core platform and are required for the platform to work, we do not consider them as extensions.

Java Web API Server

We use a stand-alone, production-grade Spring Boot Web application to host Java-based applications. Spring Boot makes the process of creating and running a highly stable Web application easy, with only a small configuration and a single command needed. Currently, we use the framework for Stanford CoreNLP's Open Information

Extraction (OpenIE) library. While CoreNLP offers a pre-built Java-based Web server, we found that the Web server seemed to be quite unstable to the extent that we could not rely on it to supply certain functionalities to our platform. Therefore, we decided to use only the library instead.

As we use Apache Maven to manage our Java Web API server project, both Spring Boot and CoreNLP, being registered to Maven dependency manager, and can be added directly as its dependencies. See the project's *pom.xml* for all configurations. In directory *src/main/java/api*, subdirectory *core* contains source codes related to Spring Boot and *connectors* contains functions that call the Java libraries. **Request Mapping** in *core/Controller.java* maps HTTP requests to the connectors. For more information on Spring Boot RESTFul Web service, visit *https://spring.io/guides/gs/rest -service/*.

To run the server, install Maven, navigate to the package's root directory, which should contain *pom.xml*, then type command:

```
mvn package
```

Maven will create a subdirectory named *target*, which should contain *uwkgm-spring-1.0.jar* as configured in *pom.xml*. Run the following command to start the Java Web server on port 80:

```
java -jar target/uwkgm-spring-1.0.jar –server.port=80
```

Make sure that you set the platform's configuration to the correct Java Web server's address and port. To change the configuration, edit environment-based configuration file *servers.yaml* in the platform's project subdirectory *env*. Note that the server's port, if not 80, must follows the address, e.g., `"http://localhost:8080"`.

Chapter 4

Development and Deployment

We containerize the UWKGM platform, its extensions, and database management systems for deployment to ensure its consistency and scalability. We use Docker for containerization and Kubernetes for coordinating the containers. While containerizing the systems automates environment setup and accelerates deployment, you still need to manually set up the environment if you wanted to make changes to the platform.

Setup for Development

You can set up the platform's development environment on Ubuntu and Mac OS. Our setup instructions require specific versions of third-party software, which we listed in the appendix. They may or may not work with other versions of the software.

In general, you need to set up the following software to develop the UWKGM platform:

1. Python 3.6 or later
2. Java Development Kit and Apache Maven
3. Homebrew on Linux or Docker (for Virtuoso)
4. MySQL or equivalent systems
5. MongoDB or equivalent systems
6. OpenLink Virtuoso or equivalent systems

Ubuntu

This setup instruction is for Ubuntu Server 18.04 LTS. Before installing Python 3, check whether you already have one installed:

```
. python3 --version
```

Make sure that you have Python version 3.6 or later installed. You also need to install *python3-dev* for MySQL connector, use **apt-get** to install it:

```
. sudo apt-get install python3-dev
```

Next, install OpenJDK and Apache Maven for Java-based extensions:

```
. sudo apt install openjdk-11-jre-headless
. sudo apt install maven
```

If you want to use MySQL, install *mysql-server* and *mysql-config* packages. Django's MySQL backend requires *mysql-config* to connect to the server.

```
. sudo apt install mysql-server
. sudo apt install default-libmysqlclient-dev
```

Follow MongoDB installation tutorial[9] to install the system. Make sure that you choose the right command for your CPU architecture.

```
. wget -qO - https://www.mongodb.org/static/pgp/server-4.2.asc |
  sudo apt-key add -
. echo "deb [ arch=amd64,arm64 ] https://repo.mongodb.org/apt/ubuntu
  bionic/mongodb-org/4.2 multiverse" | sudo tee
  /etc/apt/sources.list.d/mongodb-org-4.2.list
. sudo apt-get update
. sudo apt-get install -y mongodb-org
. sudo systemctl start mongod
```

9 https://docs.mongodb.com/manual/tutorial/install-mongodb-on-ubuntu/

While Virtuoso Open Source can be installed directly from Ubuntu repository, the version that is available is 6.1, which does not satisfy the UWKGM platform's requirements. We found two alternative options, one using Homebrew and another using Docker:

Homebrew

First, install Homebrew for Linux:

```
. sudo apt install linuxbrew-wrapper
```

Then install Virtuoso using Homebrew:

```
. brew install virtuoso
```

Note that Homebrew may ask you to follow additional instruction if command **brew** is not in your **PATH**. In which case, follow the instruction then install Virtuoso again.

Check from the installation where Homebrew installed Virtuoso. In our case, Virtuoso was installed in */home/linuxbrew/.linuxbrew/Cellar/virtuoso/7.2.5.1_1*. Navigate to the directory where Virtuoso is installed, then to the subdirectory that contains *virtuoso.ini*. Finally, start the server:

```
. cd /home/linuxbrew/.linuxbrew/Cellar/virtuoso/7.2.5.1_1
. cd var/lib/virtuoso/db
. ../../../../bin/virtuoso-t -f
```

Make sure that you set **NumberOfBuffers** and **MaxDirtyBuffers** in *virtuoso.ini* to match your available memory. Using their default value, 10000 and 6000, will drastically affect Virtuoso's performance. Every time you make changes to *virtuoso.ini*, you need to restart Virtuoso.

Docker

First, follow Docker’s installation guide[10] to install the community edition:

```
. sudo apt-get update
. sudo apt-get install apt-transport-https ca-certificates curl
  gnupg-agent software-properties-common
. curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo
  apt-key add –
. sudo apt-key fingerprint 0EBFCD88
. sudo add-apt-repository "deb [arch=amd64] https://download.docker
  .com/linux/ubuntu $(lsb_release -cs) stable"
. sudo apt-get update
. sudo apt-get install docker-ce docker-ce-cli containerd.io
```

Next, run Virtuoso using Docker. Replace `DEFAULT_GRAPH` with a graph of your choice and `/home/ubuntu/virtuoso` with the directory on your host machine where you want to store the database.

```
. sudo docker run --name uwkgm-virtuoso \
      -p 8890:8890 -p 1111:1111 \
      -e DBA_PASSWORD=dba \
      -e SPARQL_UPDATE=true \
      -e DEFAULT_GRAPH=http://dbpedia.org \
      -v /home/ubuntu/virtuoso:/data \
      -d tenforce/virtuoso
```

Finally, change **NumberOfBuffers** and **MaxDirtyBuffers** in *virtuoso.ini* then restart the container.

[10] https://docs.docker.com/install/linux/docker-ce/ubuntu/

Troubleshoot

If you encountered error `mysql_config not found` while installing the platform's dependencies:

```
. EnvironmentError: mysql_config not found
```

Make sure that you have installed package *mysql-config* using the following command:

```
. sudo apt install default-libmysqlclient-dev
```

If *mysqlclient* installer reported the following error:

```
. MySQLdb/_mysql.c:38:10: fatal error: Python.h:
  No such file or directory
```

Make sure that you have installed *python3-dev*:

```
. sudo apt-get install python3-dev
```

Mac OS

We use Homebrew to install MySQL, MongoDB, and Virtuoso. We do not recommend using Docker to run these database management systems since Mac OS's file system as a host is incompatible with Linux-based Docker containers and, thus, all of the database files must be stored in the containers instead of the host's file system, which is inefficient and will likely causes complications in maintenance.

First, run the following command to install Homebrew:

```
. /bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/
  Homebrew/install/master/install.sh)"
```

You should be able to use **brew** command right after installation. Run the following commands to install MySQL and MySQL Connector:

```
. brew install mysql
. brew services start mysql
. brew install mysql-client
```

Next follow MongoDB's installation tutorial[11]. We use MongoDB Community Edition version 4.2.

```
. brew tap mongodb/brew
. brew install mongodb-community@4.2
. brew services start mongodb-community@4.2
```

Installing Virtuoso using Homebrew on Mac OS is similar to Ubuntu:

```
. brew install virtuoso
```

After installation, make sure that you set **NumberOfBuffers** and **MaxDirtyBuffers** in *virtuoso.ini* to match your available memory. The configuration file should locate in */usr/local/Cellar/virtuoso/{version}/var/lib/virtuoso/db*. Use **cd** to access the directory and run the following command to start Virtuoso:

```
. virtuoso-t -f
```

Go to your browser and navigate to *http://localhost:8890.* You should see Virtuoso's Conductor page. The default username and password are `"dba"`.

Once you finished installing the database management services, install Python from *https://www.python.org* and Java Development Kit from *http://jdk.java.net*. You will also need to install Apache Maven from *https://maven.apache.org/download.cgi* for packaging the Java Web server (follow Maven installation guide[12] for the instruction).

[11] https://docs.mongodb.com/manual/tutorial/install-mongodb-on-os-x/
[12] https://maven.apache.org/install.html

Troubleshoot

If your installation of the platform's dependencies failed due to a problem with *mysqlclient*, make sure that you have installed *mysql-client* using Homebrew:

```
. brew install mysql-client
```

If *mysql-client* installer notified you that OpenSSL was not symlinked to /usr/local, follow its instruction:

```
. echo 'export PATH="/usr/local/opt/openssl/bin:$PATH"' >> ~/.zshrc
. export LDFLAGS="-L/usr/local/opt/openssl/lib"
. export CPPFLAGS="-I/usr/local/opt/openssl/include"
```

The dependency installer may also report the following error:

```
. mysqlclient 1.3.13 or newer is required; you have ...
```

In which case, you would need to edit your installed Django package. First, navigate to the Django package (if you used Python virtual environment, it should be in your virtual environment directory, e.g., *env/lib/python3.6/site-packages/django)*. In the package, locate *db/backends/mysql/base.py* and edit the script. Find the following lines:

```
>>> if version < (1, 3, 13):
>>>   raise ImproperlyConfigured('mysqlclient 1.3.13 or newer is
      required; you have %s.' % Database.__version__)
```

Then replace the **raise** command with **pass**:

```
>>> if version < (1, 3, 13):
>>>   pass
```

This issue appears to be due to Django's MySQL backend having a problem recognizing *mysqlclient* version 1.3.13 or later, even one was installed.

Platform Setup

Once you finished installing the required software, clone the platform from our GitHub repository:

```
. git clone https://github.com/ichise-lab/uwkgm
```

After the cloning is done, we recommend that you set up Python virtual environment in *UWKGM/api* by navigating to the directory and type the following commands:

```
. sudo apt-get install python3-venv
. python3 -m venv env
```

Install additional package for *pycrpto*:

```
. sudo apt-get install libgmp3-dev
```

Then activate the virtual environment and install packages from *requirements.txt*:

```
. source env/bin/activate
. pip install -r requirements.txt
```

Now that the required packages are installed, create a database in MySQL and you should be ready to start the API server:

```
. sudo mysql -e "CREATE DATABASE UWKGM"
```

In UWKRM root directory, you will find a folder named *dev*. In the folder, we created a configuration file *conf.sh* for the development environment. **UWKGM_ENV** in the file determines what environment the platform will operate (the default is `"development"`). There a few steps needed to start the platform, including loading configurations, activating Python's virtual environment, migrating Django's database backend, and setting environment variables. We created a script for these steps in *dev-api.sh*. To start the API server, execute one of the following commands:

```
. sudo bash dev-api.sh        # Ubuntu
. sh dev-api.sh               # Mac OS
```

If you did not make any changes to the script, the API server should run at port 8001. Navigate to *http://localhost:8001/api/v1.0/admin/* for Django's administration page.

Troubleshoot

If you started the API server using *dev-api.sh* and Django crashed because environment variables **UWKGM_ENV** or **UWKGM_EXT_HOST** were not set, check that they are declared in *dev/conf.sh* and that *dev-api.sh* executes *conf.sh* using command **source**.

Containerization and Deployment

We created a collection of Ubuntu-based scripts that automatically containerizes and deploys the platform, including its extensions and database management systems. You will need Docker, Kubernetes, and Nginx for the deployment. The scripts include installation instructions of all three software. However, we highly recommend that you familiarize yourself with the software before you move into production stage since you will need to make certain adjustments for security and scalability.

Figure 4.1 shows an overview of a Kubernetes cluster deployed using our scripts. The cluster can be deployed on a single machine and a multi-node cluster. Kubernetes is a powerful tool for orchestrating containerized applications. The system deploys, scales, and manages containers automatically as you configured. Orchestrating containers is vital to deploying the UWKGM platform since the platform does not work as a stand-alone application but coordinate with its supporting systems, which are also containerized. While we provide deployment scripts for MySQL and MongoDB, you may instead opt for the systems' cloud services such as MariaDB on Amazon AWS, as well as MongoDB Atlas. Virtuoso Commercial Edition users may also choose Virtuoso's clustering deployment[13] instead of deploying using Kubernetes.

[13] http://vos.openlinksw.com/owiki/wiki/VOS/VirtClusteringDiagrams

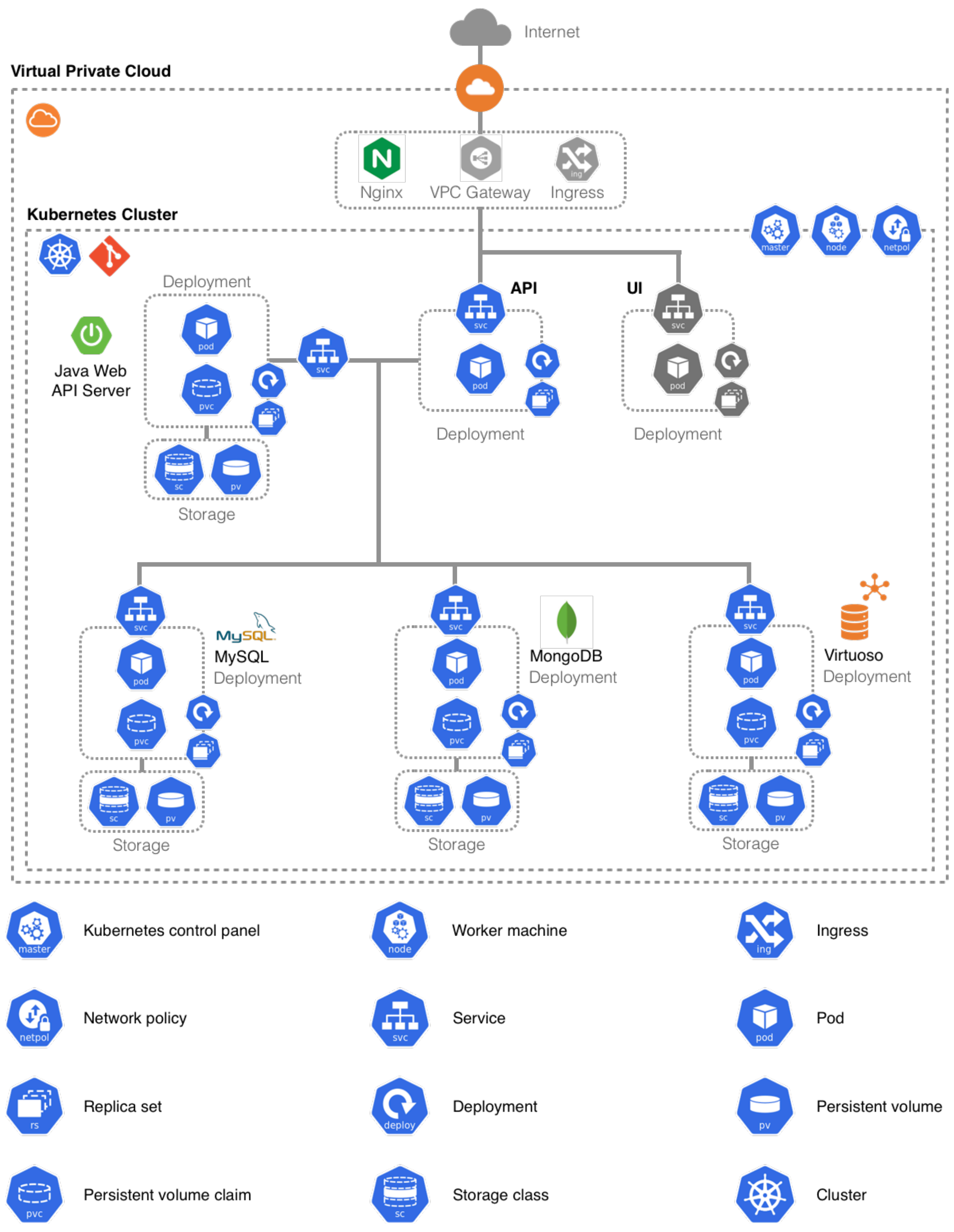


Figure 4.1 A cluster of UWKGM containers in virtual private cloud

If you manage the cluster on your server, i.e., not on any third-party services such as Amazon AWS, then you will need to configure your network policy. We use Nginx to route incoming traffic to the Web API and UI containers, while also defined additional rules that direct certain traffic to other containers for maintenance. Excluding UI, which is not part of the platform, we created five deployments, each with its own Kubernetes service. In Kubernetes, a service acts as a network service for a set of pods, which encapsulate containers and necessary infrastructures.

The API server connects to other systems using Kubernetes' internal network. For example, it connects to the Java Web API server at *http://uwkgm-spring-service*. Open *UWKGM/servers/spring/spring.template.yaml* and you will see a service named *uwkgm-spring-service*. A service of type *NodePort* exposes itself to networks outside Kubernetes. Therefore, to access an application running in a deployment (of a set of containers), e.g., Mongo Express[14] (MongoDB Web UI), you need to declare a service as *NodePort* and create a rule to direct certain traffic to the service. Once declared a *NodePort*, Kubernetes maps the service's internal ports to external ports where you can connect directly from outside the cluster. Getting the ports depends on the software that you use to manage Kubernetes.

Deployment Scripts

To deploy the platform and its dependencies on Kubernetes, you will need to run a few scripts in *UWKGM/kubernetes*.

Table 4.1 Deployment scripts for Kubernetes

Filename	Description
conf.sh	Configurations, including package versions
env.sh	Environment-based configurations (environment variable: **UWKGM_ENV**)
init.sh	Initialization script that inputs parameters for deployment and update
parser.sh	Input parser
start.sh	Start screen that show configurations before deployment and update
nginx.sh	Nginx automatic configuration script
setup.sh	Setup script for Nginx, Docker, Kubernetes, and Minikube on Ubuntu 18.04
deploy.sh	First-time deployment script
update.sh	Deployment-update script

[14] https://github.com/mongo-express/mongo-express

For users of Ubuntu 18.04, after you downloaded all project's files, run *setup.sh* using command **bash** if you have not set up Nginx, Docker, Kubernetes, and Minikube. The script will also set up Docker's local repository at port 5000.

```
. sudo bash setup.sh
```

Before you deploy the platform, make sure that Docker has set up a local repository and Kubernetes is running:

```
. sudo docker container list
. sudo kubectl get service
```

The first command should display docker containers currently running. In the *IMAGE* column, there should be *registry:2* running at port 0.0.0.0:5000->5000/tcp, meaning that Docker's local registry is working properly. The second command should list services running within the Kubernetes cluster. Once the systems are up and running, you should be ready to deploy the platform:

```
. sudo bash deploy.sh
```

The deployment script will first ask you to input a few configurations, the first one being the environment where you are deploying the platform. The default configuration is `production` (press Enter or Return to use default configuration). We also defined a variant setting named `ext` for every environment. In `ext` environments, the platform's Web API server operates on the assumption that other containers are not deployed in the same Kubernetes cluster. We use `ext` environments for testing the production server on a machine with limited resources.

The next configuration is of the master production server's address. This setting tells the UI application where to look for the main production server. The next setting does the same for `ext` production server. Next, you will need to tell Virtuoso how much memory you will make available for it to operate. Virtuoso may consume a large amount of memory, depending on your graphs. For DBpedia graph, we recommend at least 32 GB of memory. Lastly, input your default graph URI, then the deployment script will show you the summary of the deployment. Press Enter or Return to start the process.

Note

If you want to deploy specific containers into the cluster, use parameter **-c** or **-components**. Our current options are: api, mysql, mongo, virtuoso, and spring. For example:

```
. sudo bash deploy.sh -c=api,mysql
```

Deployment instructions vary among the containers. However, their common process is containerizing and deploying the container to the cluster. The instructions and configurations are located in folder *kubernetes* in each component's root directory.

At the end of the entire deployment process, the deployment script shows Docker images, services and deployments in the Kubernetes cluster, and exposed addresses and ports of the services. The script has already created networking rules for Nginx so you should be able to access the platform. However, if could not connect to it, try executing *nginx.sh* to recreate the rules:

```
. sudo bash nginx.sh
```

MySQL

We created two deployments that are related to MySQL. In folder *UWKGM/servers/mysql*, there are two subfolders, namely *admin* and *mysql*. The former contains Kubernetes deployment scripts that pull the phpMyAdmin (PMA) version 4 directly from Docker repository. The deployment serves as Web-based control panel for MySQL. To change PMA version, edit *conf.sh* in the subdirectory.

Our configuration for MySQL deployment includes allocating a persistent volume[15]. The allocated space is 100 MB. If you want to allocate more space, you will need to set the storage capacities of both `PersistentVolume` and `PersistentVolumeClaim`.

⚠ Security Notice

The default MySQL root user's password is "password". **We strongly recommend that you change to a more secure password in production**.

In MySQL 8.0, *caching_sha2_password* is the default authentication plugin rather than *mysql_native_password*. However, current version of Django's MySQL backend raised an error when we tried to connect to the database using the plugin. As a workaround, our deployment script (*deploy.sh* in subdirectory *mysql*) creates another one-time instance from MySQL image in addition to the main MySQL deployment. Once created, the instance will try to set the authentication plugin for the root user to *mysql_native_password*, and will also create a database named UWKGM. Since the deployment of MySQL takes some time to finish, the deployment script may make several attempts to apply the changes.

Troubleshoot

If Django reported an error with caching SHA2 password, try redeploying MySQL:

```
. sudo bash deploy.sh -c=mysql
```

The deployment script will try to set the authentication plugin to *mysql_native_password*.

MongoDB

Similar to MySQL, we created MongoDB deployment (*mongo*) for the Non-SQL database, and Mongo Express deployment *(express*) for its Web-based control panel in *UWKGM/servers/mongo*. MongoDB deployment also contains a persistent volume

[15] https://kubernetes.io/docs/concepts/storage/persistent-volumes/

starting at 100 MB. The default username is `"root"`, and password is `"password"`, which should be changed in production.

⚠ Security Notice

Mongo Express deployment includes pre-configured username and password. Its Web UI does not require any login to access the database. Since our deployment script automatically creates a rule that exposes Mongo Express to connections outside the Kubernetes cluster, **we highly recommend that you disable the rule unless for diagnostic purposes**.

Virtuoso

Since Virtuoso has its own admin Web UI, we do not need additional deployment for control panel. The Virtuoso deployment (in *UWKGM/servers/virtuoso*) contains a persistent volume for database backend storage, the initial size of which is 60 GB as we tested it with DBpedia graph. There is also another persistent volume of 40 GB for graph importing. To import graphs, store your graph files (such as *.ttl) in the directory indicated by variable `UWKGM_VIRTUOSO_HOST_EXT_PATH` in *conf.sh*. Kubernetes will map the directory to the Virutoso instance's internal directory. Our `UWKGM_KUBE_HOST_PATH` is defined in *UWKGM/kubernetes/conf.sh*. By default, our import commands (in *dbpedia.sql*) look for *.ttl and *.owl files and import the files as a graph with URI *http://dbpedia.org*. To change these settings, edit *import.sql*.

⚠ Security Notice

The default Virtuoso root user's password is "password". **We highly recommend that you change to a more secure password in production**.

Note that importing large graphs such as DBpedia may take several hours.

Java Web Server

We use the latest version of Apache Maven to build a container image for the Spring Boot application. The scripts are in *UWKGM/servers/spring*. Unlike the databases, which are expected to grow over time, the Java Web server does not change after packaging. Therefore, there is no need for persistent volumes. Also, since the server does not require management, we do not expose it to outside connection, thereby reducing security risks. All of the source codes for the server are located in subfolder *package*.

Web API Server

The server's deployment scripts are in *UWKGM/api/Kubernetes*. Similar to the Java Web server, there is no need for persistent volumes. As indicated in the Docker file, at the beginning of the deployment, the API server must perform migrations, which create tables and relations in the database according to defined data models. In current version of the UWKGM platform, only app *accounts* requires migration.

After migration, the platform also initialize pre-configured groups, permissions, and user accounts. The initialization commands are in method **ready** of class **Accounts Config** in **accounts.apps**. Django calls the method both during migration and when the server starts. The problem is the initialization commands cannot be executed unless after migration since tables may not exist before migration. Therefore, we defined variable `UWKGM_STATE` to prevent the migration commands from being executed before the migration. The instruction is as followed:

1. Set `UWKGM_STATE` to `"migrating"`
2. Migrate using Django commands **makemigrations** and **migrate**
3. Set `UWKGM_STATE` to `"running"`
4. Start the server

Once containerized, all container images are stored in Docker local repository.

Chapter 5

API Endpoint References

This chapter lists all Web API endpoints on the UWKGM platform, except extensions' endpoints, as well as those of the database management systems, since your choices of the third-party software and services may be different than ours. The description of each endpoint includes its description, relative URL, parameters, and responses. In each section, you will find a root URL, which shorten the endpoints' URLs in our explanation. For example, suppose the root URL of a collection of structured endpoints is …*/greet* and an endpoint's absolute URL is *https://uwkgm.nii.ac.jp/api/v1.0/greet/morning*, the relative URL would be …*/greet/morning*.

The description of each parameter begins with its name. If the parameter is required, i.e., no default value provided, it is asterisked, and you need to include its value in your requests; otherwise, you will find its default value after its name. Following the name is the type of the value and explanation.

Graph Manipulation

Root URL: …/api/v1.0/database

Django app: database

Root endpoint package: database.database

Graph Database

…/graph/entities/find/candidates **GET**

Structured Endpoint

Finds candidates of entities given a partial or full label

🔒 Restricted to registered users

This endpoint returns three list of candidates:

- 'exact_matches' list contains candidates which labels match the search label perfectly
- 'first_matches' list contains candidates which labels start with the search label
- 'partial_matches' list contains candidates which labels partially match the search label

Parameters:

label:* **str**	The partial or full search label
limit: **int** = 10	The maximum number of candidates in each of the response lists
language: **str** = "en"	A language filter for the candidates' labels

query_limit: **int** = None

The maximum number of candidates specified in the database query command

perfect_match_only: **bool** = False

Finds only candidates that exactly match the search label

graph: **str** = {default_graph_uri}

Graph URI

Responses:

200_OK	The three lists of candidates ({"exact_matches": ..., "first_matches": ..., "partial_matches": ...})

…/graph/entities/find/candidates_aggregated	**GET**

Structured Endpoint

Lists numbers of candidates that exactly match the given labels

Restricted to registered users

Parameters:

labels:* **List[str]**	A list of labels
language: **str** = "en"	A language filter for the candidates' labels

query_limit: **int** = None

The maximum number of candidates specified in the database query command

perfect_match_only: **bool** = True

Finds only candidates that exactly match the search label

graph: **str** = {default_graph_uri}

Graph URI

Responses:

200_OK	A dictionary containing the labels and their numbers of candidates

…/graph/entities/find/single	**GET**

Structured Endpoint

Finds triples which the given entity is a part of (as subject, predicate, object, or any)

Restricted to registered users

Parameters:

entity:* **str**	An entity to find the triples
limit: **int** = 10000	The maximum number of returned triples

as_element: **str** = None

Find triples where the entity is their 'subject', 'predicate', or 'object'

graph: **str** = {default_graph_uri}

Graph URI

Responses:

200_OK	Lists of triples pertaining to the entity's role ({'subject': [...], 'predicate': [...], 'object': [...]})

…/graph/triples/add/single	GET

Structured Endpoint

Adds a triple to the graph database

Restricted to superusers

Parameters:

triple:* **Tuple[str, str, str]**

Entities of (subject, predicate, object)

graph: **str** = {default_graph_uri}

Graph URI

Responses:

200_OK	Triple adding status

…/graph/triples/delete/single	GET

Structured Endpoint

Deletes a triple from the graph database

Restricted to superusers

Parameters:

triple:* **Tuple[str, str, str]**

Entities of (subject, predicate, object)

graph: **str** = {default_graph_uri}

Graph URI

Responses:

200_OK	Triple deleting status

…/graph/triples/find/single **GET**

Structured Endpoint

Checks whether a triple exists in the graph

🔒 Restricted to registered users

Parameters:

triple:* **Tuple[str, str, str]**

Entities of (subject, predicate, object)

graph: **str** = {default_graph_uri}

Graph URI

Responses:

200_OK	True if the triple exists in the graph

Graph Modifier Database

…/data/texts/add/single **GET**

Structured Endpoint

Adds a document containing the given text

🔒 Restricted to registered users

Parameters:

document:* **dict**	A dictionary containing the text Example: {"text": "Barack Obama was born in Hawaii"}

Responses:

200_OK	Added document's id

…/data/triples/add/single **GET**

Structured Endpoint

Adds a document containing an add-triple modifier

🔒 Restricted to registered users

Parameters:

triple:* **Tuple[str, str, str]**

A triple (subject, predicate, object)

Example: ["http://dbpedia.org/resource/Barack_Obama",
"http://dbpedia.org/property/birthplace",
"http://dbpedia.org/resource/Hawaii"]

text_id: **str** = None

If the triple was extracted from a text, 'text_id' would store the reference to the original text.

Responses:

200_OK	Added document's id

…/data/triples/add/check **GET**

Structured Endpoint

Checks whether a triple does not exist in the graph

Restricted to registered users

Parameters:

triple:* **Tuple[str, str, str]**

A triple (subject, predicate, object)

Example: ["http://dbpedia.org/resource/Barack_Obama",
"http://dbpedia.org/property/birthplace",
"http://dbpedia.org/resource/Hawaii"]

Responses:

200_OK	True if the triple does not exist in the graph

…/data/triples/commit/auto **GET**

Structured Endpoint

Commits triple modifiers that have not been committed before to the graph database

Restricted to superusers

Responses:

200_OK	The number of triple modifiers committed in this transaction

…/data/triples/delete/single **GET**

Structured Endpoint

Deletes a document containing a delete-triple modifier

🔒 Restricted to registered users

Parameters:

triple:* **Tuple[str, str, str]**

A triple to delete from the graph

Example: ["http://dbpedia.org/resource/Barack_Obama",
"http://dbpedia.org/property/birthplace",
"http://dbpedia.org/resource/Hawaii"]

Responses:

200_OK	Deleted document's id

…/data/triples/edit/single **GET**

Structured Endpoint

Edits a document containing a delete-triple modifier

🔒 Restricted to registered users

Parameters:

original_triple:* **Tuple[str, str, str]**

The original triple to be edited (subject, predicate, object)

Example: ["http://dbpedia.org/resource/Barack_Obama",
"http://dbpedia.org/property/birthplace",
"http://dbpedia.org/resource/Hawaii"]

triple:* **Tuple[str, str, str]**

The replacing triple (subject, predicate, object)

Example: ["http://dbpedia.org/resource/Barack_Obama",
"http://dbpedia.org/property/birthplace",
"http://dbpedia.org/resource/United_States"]

Responses:

200_OK	Edited document's id

…/data/triples/find/multiple **GET**

Structured Endpoint

Finds multiple triple modifiers

🔒 Restricted to registered users

Responses:

200_OK	A list of triple modifiers

…/data/triples/remove/single **GET**

Structured Endpoint

Deletes a triple modifier

🔒 Restricted to registered users

Parameters:

document_id:* **str**	Document id of the triple modifier

Responses:

200_OK	The number of triple modifiers removed in this transaction

Knowledge Graph Toolkit

Root URL: …/api/v1.0/knowledge

Django app: knowledge

Root endpoint package: knowledge.knowledge

Knowledge Extraction

…/extraction/extract/triples/linked **GET**

Structured Endpoint

Extracts linked triples from a text

URL Forwarding

../extraction/extract

🔒 Restricted to registered users

Parameters:

text:* **str**

A string to be extracted	Example: "Barack Obama born in Hawaii"

Responses:

200_OK	A list of triples

…/extraction/extract/triples/raw/openie **GET**

Structured Endpoint

Extracts triples using Stanford CoreNLP OpenIE library via CoreNLPConnector in Java Web API

🔒 Restricted to registered users

Parameters:

text:* **str**

A string to be extracted.	Example: "Barack Obama born in Hawaii"

Responses:

200_OK	A list of triples

Knowledge Integration

…/integration/map/entities/dbpedia_spotlight **GET**

Structured Endpoint

Maps entities from a text

🔒 Restricted to registered users

Parameters:

text:* **str**	A string (to be mapped). Example: "Barack Obama born in Hawaii"

endpoint: **str** = {default_endpoint_dbpedia}

Annotator endpoint. Example: "http://model.dbpedia-spotlight.org/en/annotate"

confidence: **float** = 0.5

Minimum threshold of confidence value of found entities

Responses:

200_OK	A dictionary of mapped entities (URI)

…/integration/map/predicates/scoring **GET**

Structured Endpoint

Replaces predicates with linked predicates (URIs)

🔒 Restricted to registered users

Parameters:

triples:* **List[Tuple[str, str, str]]**

A list of triples to be processed [(subject, predicate, object), ...]

Example: [["http://dbpedia.org/resource/Barack_Obama",
"bear in", "http://dbpedia.org/resource/Hawaii"]]

n_candidates_threshold: **int** = 5

A maximum number of candidates to be verified.

The verification algorithm may take a very long time to verify a large set of candidates.

Responses:

200_OK	A list of triples with predicates replaced

…/integration/aggregate/triples **GET**

Structured Endpoint

Replaces subjects and objects in a list of triples with entities

Restricted to registered users

Parameters:

triples:* **List[Tuple[str, str, str]]**

A list of triples [(subject, predicate, object), …]

Example: [["Barack Obama", "bear in", "Hawaii"]]

entities:* **Dict[str, str]**

A dictionary of known entities

Example: {"Barack Obama": "http://dbpedia.org/resource/Barack_Obama",
"Hawaii": "http://dbpedia.org/resource/Hawaii"}

Responses:

200_OK	A list of triples with subjects and objects replaced with their entities

Knowledge Verification

…/verification/verify/agreement/dbpedia **GET**

Structured Endpoint

Verifies domain-range agreement using DBpedia's ontology

Restricted to registered users

Parameters:

triple:* **Tuple[str, str, str]**

A triple (subject's entity, predicate, object's entity)

Example: ["http://dbpedia.org/resource/Barack_Obama", "bear in",
"http://dbpedia.org/resource/Hawaii"]

predicate:* **str** Entity of the predicate to be verified

Example: "http://dbpedia.org/ontology/birthPlace"

Responses:

200_OK	True if the predicate does not constitute domain-range violation

User Management

Root URL: …/api/v1.0/accounts

Django app: accounts

Root endpoint package: accounts

<u>User Accounts</u>

…/	GET
Lists accounts	

🔒 Restricted to superusers and users with account-management permissions

Responses:

200_OK	A list of users

…/	POST
Creates an account	

🔒 Restricted to superusers and users with account-management permissions

Responses:

201_CREATED	New account's detail

…/{username}	GET
Shows an account detail	

🔒 Restricted to account owners, superusers, and users with account-management permissions

Responses:

200_OK	Account's detail

…/{username} **DELETE**

Deletes an account

🔒 Restricted to superusers and users with account-management permissions

Responses:

200_OK

…/{username} **PATCH**

Edits an account

🔒 Restricted to account owners, superusers, and users with account-management permissions

🔒 Restricted to superusers and users with account-management permissions

Data:

email: **str**	Email
password: **str**	Password
first_name: **str**	First name
last_name: **str**	Last name
is_active: **str**	Activation status 🔒
is_staff: **str**	Staff status 🔒
is_superuser: **str**	Superuser status 🔒
groups: **str**	Groups 🔒

user_permissions: **str**

Permissions 🔒

is_email_verified: **bool**

Email verification status 🔒

date_joined: **datetime**

Date joined 🔒

last_login: **datetime**

Last login 🔒

Responses:

201_CREATED	New account’s detail

…/{email}/activate	GET
Sends an activation code to a registered account’s email	

🔒 Restricted to account owners

Responses:

200_OK	Activation-code sending status

…/{email}/activate	PATCH
Activates an account	

🔒 Restricted to account owners

Parameters:

code:* **str**	Activation code

Responses:

200_OK	Activation status

…/{email}/deactivate	PATCH
Deactivates an account	

🔒 Restricted to account owners, superusers, and users with account-management permissions

Responses:

200_OK	Deactivation status

…/password/reset	GET
Sends password-reset code to a registered email	

🔒 Restricted to account owners, superusers, and users with account-management permissions

Parameters:

email:* **str**	Email address

Responses:

200_OK	Deactivation status

…/password/reset/…	PATCH
Resets an account’s password	

Restricted to account owners, superusers, and users with account-management permissions

Password-reset URLs are automatically generated by Django REST Framework.

Responses:

200_OK	Password update status

…/groups	GET
Lists groups	

Restricted to superusers and users with permission "auth.view_group"

Responses:

200_OK	A list of groups

…/permissions	GET
Lists permissions	

Restricted to superusers and users with permission "auth.view_group"

Responses:

200_OK	A list of permissions

Authentication

…/tokens/obtain	**POST**
Generates access and refresh tokens	

Parameters:

email:* **str**	Email address
password:* **str**	Password

Responses:

200_OK	Access and refresh tokens
401_UNAUTHORIZED	

…/tokens/refresh	**POST**
Generates an access token from a refresh token	

Parameters:

refresh:* **str**	Refresh token

Responses:

200_OK	Access token
401_UNAUTHORIZED	

…/tokens/verify	**POST**
Verifies access or refresh token	

Parameters:

token:* **str**	Access/refresh token

Responses:

200_OK

401_UNAUTHORIZED

Throttling

…/{username}/throttling/burst/requests/	GET

Lists rate-limit burst requests belonging to the user

🔒 Restricted to account owners, superusers, and users with account-management permissions

Responses:

200_OK	A list of rate-limit burst requests

…/{username}/throttling/burst/requests/	POST

Creates a rate-limit burst request

🔒 Restricted to account owners, superusers, and users with account-management permissions

Parameters:

user:* **str**	Username
api: **str**	API name
name: **str**	Rate limit names from choices defined in Django REST Framework's setting in the project *settings.py*
limit: **int**	A new number of requests limited per time frame
duration: **int**	Time frame in seconds
start: **datetime**	The rate-limit's start date
expire: **datetime**	The rate-limit's expiration date

Responses:

200_CREATED	The detail of created burst request

…/{username}/throttling/burst/requests/{request_id}	GET

Shows a rate-limit burst request detail

🔒 Restricted to account owners, superusers, and users with account-management permissions

Responses:

200_OK	The detail of the burst request

…/{username}/throttling/burst/requests/{request_id} **PATCH**

Edits a rate-limit burst request

🔒 Restricted to account owners, superusers, and users with account-management permissions

Parameters:

user:* **str**	Username
api: **str**	API name
name: **str**	Rate limit names from choices defined in Django REST Framework's setting in the project *settings.py*
limit: **int**	A new number of requests limited per time frame
duration: **int**	Time frame in seconds
start: **datetime**	The rate-limit's start date
expire: **datetime**	The rate-limit's expiration date

Responses:

200_OK	The detail of created burst request

…/{username}/throttling/burst/requests/{request_id} **DELETE**

Deletes a rate-limit burst request

🔒 Restricted to account owners, superusers, and users with account-management permissions

Response:

204_NO_CONTENT

…/{username}/throttling/burst/permits/	**GET**
Lists rate-limit burst permits granted to a user	

🔒 Restricted to account owners, superusers, and users with account-management permissions

Response:

200_OK	A list of permits granted to the user (Granter information is omitted for API requests made by account owners)

…/{username}/throttling/burst/permits/	**POST**
Creates a rate-limit burst permit	

🔒 Restricted to superusers and users with account-management permissions

Parameters:

user:* **str**	Username of the grantee
granter:* **str**	Username of the granter
api: **str**	API name
name: **str**	Rate limit names from choices defined in Django REST Framework's setting in the project *settings.py*
limit: **int**	A new number of requests limited per time frame
duration: **int**	Time frame in seconds
start: **datetime**	The rate-limit's start date
expire: **datetime**	The rate-limit's expiration date
request: **str**	Rate-limit request's ID

Response:

201_CREATED	The detail of created burst permit

…/{username}/throttling/burst/permits/{permit_id}	**GET**
Shows a rate-limit burst permit detail	

🔒 Restricted to account owners, superusers, and users with account-management permissions

Responses:

200_OK	The detail of the burst permit (Granter information is omitted for API requests made by account owners)

…/{username}/throttling/burst/permits/{permit_id}	**PATCH**
Edits a rate-limit burst permit	

🔒 Restricted to superusers and users with account-management permissions

Parameters:

user:* **str**	Username of the grantee
granter:* **str**	Username of the granter
api: **str**	API name
name: **str**	Rate limit names from choices defined in Django REST Framework's setting in the project *settings.py*
limit: **int**	A new number of requests limited per time frame
duration: **int**	Time frame in seconds
start: **datetime**	The rate-limit's start date
expire: **datetime**	The rate-limit's expiration date
request: **str**	Rate-limit request's ID

Response:

200_OK	The detail of edited burst permit

…/{username}/throttling/burst/permits/{permit_id}	**DELETE**
Deletes a rate-limit burst permit	

🔒 Restricted to superusers and users with account-management permissions

Responses:

204_NO_CONTENT

Appendix

Third-Party Software

The UWKGM platform and its associated software use the third-party products listed below under their licenses. Each third-party product may rely on other products that are not listed here. We have provided the URLs to our third-party products' website for further information.

Core UWKGM Platform

		License
Python 3.8.1	*https://www.python.org*	**Python Software Foundation**
	An interpreted, high-level, general-purpose programming language	
Django 3.0.3	*https://www.djangoproject.com*	**BSD**
	A high-level Python Web framework	
PyJWT 1.7.1	*https://github.com/jpadilla/pyjwt*	**MIT**
	A Python implementation of RFC 7519 (JSON Web Token)	
PyMySQL 0.9.3	*https://github.com/PyMySQL/PyMySQL*	**MIT**
	A pure-Python MySQL client library based on PEP 249 (Python Database API Specification)	
PyYAML 5.3	*https://github.com/yaml/pyyaml*	**MIT**
	YAML parser and emitter for Python	
SPARQL Wrapper 1.8.5	*https://rdflib.github.io/sparqlwrapper*	**W3C**
	A wrapper around a SPARQL service	

Cython BLIS 0.4.1 *https://github.com/explosion/cython-blis* **⊕ [16]**

The Blis linear algebra routines as a self-contained Python C-extension

Catalogue 0.0.8 *https://github.com/explosion/catalogue* **MIT**

Lightweight function registries for Python libraries

Certifi 2019.6.16 *https://github.com/certifi/python-certifi* **Mozilla Public 2.0**

A collection of Root Certificates

CFFI 1.13.2 *https://cffi.readthedocs.io* **© [17]**

C Foreign Function Interface for Python

Chardet 3.0.4 *https://github.com/chardet/chardet* **GNU LGPL 2.1**

The Universal Character Encoding Detector

Cryptography 2.8 *https://github.com/pyca/cryptography* **Apache 2.0 & BSD**

A package which provides cryptographic recipes and primitives to Python developers

Cymem 2.0.3 *https://github.com/explosion/cymem* **MIT**

Memory-management helpers for Cython

Django CORS Headers 3.1.0 *https://github.com/adamchainz/django-cors-headers* **MIT**

A Django App that adds Cross-Origin Resource Sharing (CORS) headers to responses

Django REST Framework 3.10.3 *https://www.django-rest-framework.org* **© [18]**

Web APIs for Django

Simple JWT 4.3.0 *https://github.com/davesque/django-rest-framework-simplejwt* **MIT**

A JSON Web Token authentication backend for the Django REST Framework

IDNA 2.8 *https://github.com/kjd/idna* **© [19]**

Support for the Internationalised Domain Names in Applications (IDNA) protocol

Importlib Metadata 1.3.0 *https://importlib-metadata.readthedocs.io/en/latest* **Apache 2.0**

A library which provides an API for accessing an installed package's metadata

ISO Date 0.6.*0* *https://github.com/gweis/isodate/*

ISO 8601 Date/Time Parser

[16] https://github.com/explosion/cython-blis/blob/master/LICENSE
[17] 2012 – 2018, Armin Rigo, Maciej Fijalkowski
[18] 2011-present, Encode OSS Ltd
[19] 2013-2018, Kim Davies

More Itertools 8.0.2 *https://github.com/more-itertools/more-itertools* **MIT**

More routines for operating on iterables beyond itertools

MurmurHash 1.0.2 *https://github.com/explosion/murmurhash* **MIT**

CPython bindings for MurmurHash

MySQL Connector 2.2.9 *https://dev.mysql.com/doc/connector-python/en* **MySQL Connector**

MySQL driver written in Python

MySQL Client 1.4.6 *https://github.com/PyMySQL/mysqlclient-python* **GPL 2.0**

MySQL database connector for Python

NLTK 3.4.5 *http://www.nltk.org* **Apache 2.0**

Natural Language Toolkit

NumPy 1.18.0 *https://numpy.org* **BSD**

The fundamental package for scientific computing with Python

Plac 1.1.3 *https://github.com/micheles/plac* **BSD 2-Clause**

A command line parser

Preshed 3.0.2 *https://github.com/explosion/preshed* **MIT**

Cython Hash Table for Pre-Hashed Keys

Psutil 5.6.3 *https://github.com/giampaolo/psutil* **BSD 3-Clause**

Cross-platform library for process and system monitoring in Python

Pycparser 2.19 *https://github.com/eliben/pycparser* **©** [20]

A parser for the C language, written in pure Python

PyCrypto 2.6.1 *https://www.dlitz.net/software/pycrypto* **©** [21]

The Python Cryptography Toolkit

PyMongo 3.10.0 *https://github.com/mongodb/mongo-python-driver* **Apache 2.0**

The Python driver for MongoDB

PyParsing 2.4.2 *https://github.com/pyparsing/pyparsing* **MIT**

A Python Parsing Module

[20] 2008-2017, Eli Bendersky

[21] https://github.com/dlitz/pycrypto/blob/master/COPYRIGHT

Pytz 2019.2	*https://pythonhosted.org/pytz*	**MIT and Zope Public 2.1**
	World time zone definitions, modern and historical	
RDFLib 4.2.2	*https://github.com/RDFLib/rdflib*	**©**[22]
	A Python library for working with RDF	
Requests 2.22.0	*https://requests.readthedocs.io*	**Apache 2.0**
	HTTP library for Python	
Six 1.12.0	*https://github.com/benjaminp/six*	**MIT**
	Python 2 and 3 compatibility library	
SpaCy 2.2.3	*https://spacy.io*	**MIT**
	Industrial-strength Natural Language Processing in Python	
SQL Parse 0.3.0	*https://github.com/andialbrecht/sqlparse*	**BSD 3-Clause**
	A non-validating SQL parser module for Python	
Srsly 0.2.0	*https://github.com/explosion/srsly*	**MIT**
	Modern high-performance serialization utilities for Python	
Thinc 7.3.1	*https://github.com/explosion/thinc*	**MIT**
	A lightweight deep learning library	
Tqdm 4.41.0	*https://github.com/tqdm/tqdm*	**Mozilla Public 2.0 & MIT**
	A Fast, Extensible Progress Bar for Python and CLI	
Urllib3 1.25.3	*https://urllib3.readthedocs.io*	**MIT**
	HTTP client for Python	
Wasabi 0.4.2	*https://ines.io*	
	A lightweight console printing and formatting toolkit	
Zipp 0.6.0	*https://github.com/jaraco/zipp*	**MIT**
	A pathlib-compatible Zipfile object wrapper	

[22] 2002-2017, RDFLib Team

UWKGM Extensions

OpenJDK 9 *https://openjdk.java.net* **Apache 2.0**

Open-source implementation of the Java Platform, Standard Edition and related projects

Maven 3 *http://maven.apache.org* **Apache 2.0**

A software project management and comprehension tool

Spring Boot 2.0 *https://spring.io/projects/spring-boot* **Apache 2.0**

Spring Boot makes it easy to create stand-alone, production-grade Spring based Applications

CoreNLP 3.9.2 *https://stanfordnlp.github.io/CoreNLP* **GPL 3 or later**

Stanford CoreNLP provides a set of human language technology tools

Database Management Systems

MySQL Community Edition *https://www.mysql.com/products/community* **GPL 2**

An open-source relational database management system

Virtuoso Open-Source Edition *http://vos.openlinksw.com* **GPL 2**

A middleware and database engine hybrid

MongoDB Community Server *https://www.mongodb.com* **MongoDB, Inc. & Apache 2.0**

A cross-platform document-oriented database program

Deployment

Docker *https://www.docker.com* **Apache 2.0**

A set of platform-as-a-service products that use OS-level virtualization to deliver software in packages

Kubernetes *https://kubernetes.io* **Apache 2.0**

Automated container deployment, scaling, and management

Nginx *https://nginx.org* **BSD 3-Clause**

A web server which can also be used as a reverse proxy, load balancer, mail proxy and HTTP cache

References

Angeli, G., Premkumar, M. J. J., & Manning, C. D. (2015, July). Leveraging linguistic structure for open domain information extraction. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)* (pp. 344-354).

Daiber, J., Jakob, M., Hokamp, C., & Mendes, P. N. (2013, September). Improving efficiency and accuracy in multilingual entity extraction. In *Proceedings of the 9th International Conference on Semantic Systems* (pp. 121-124).

Kertkeidkachorn, N., & Ichise, R. (2018). An automatic knowledge graph creation framework from natural language text. *IEICE TRANSACTIONS on Information and Systems*, 101(1), 90-98.

Lertvittayakumjorn, P., Kertkeidkachorn, N., & Ichise, R. (2017, November). Resolving range violations in DBpedia. In *Joint international semantic technology conference* (pp. 121-137). Springer, Cham.

Marr B. (2018, May). How Much Data Do We Create Every Day? The Mind-Blowing Stats Everyone Should Read. *Forbs*. Available online: https://www.forbes.com/sites/bernardmarr/2018/05/21/how-much-data-do-we-create-every-day-the-mind-blowing-stats-everyone-should-read

Rahoman, M. M., & Ichise, R. (2016). Automatic erroneous data detection over type-annotated linked data. *IEICE TRANSACTIONS on Information and Systems*, 99(4), 969-978.

Ebisu, T., & Ichise, R. (2018, April). Toruse: Knowledge graph embedding on a lie group. In *Thirty-Second AAAI Conference on Artificial Intelligence*.