UNIVERSITY OF UDINE

DEPARTMENT OF MATHEMATICS, COMPUTER AND PHYSICAL SCIENCE

READERSOURCING 2.0
TECHNICAL DOCUMENTATION

MICHAEL SOPRANO AND STEFANO MIZZARO

v1.0-alpha

Contents

List of Figures

List of Tables

1 Introduction

This technical documentation provides an overview of *Readersourcing 2.0*. Initially, a recap of its general architeture is presented and subsequently, for each of its components, a brief recap of their role and purpose is presented along with some specific aspects, such as the technology used, the internal architecture, the structure of the database and more. This is done by using different types of diagrams belonging to the UML standard (unless otherwise specified) which are drew according to the set of style rules for that standard proposed by Fowler [1].

2 General Architecture

Readersourcing 2.0 is composed of more than one application. Indeed, there must be one application that acts as a server to gather all the ratings given by readers and one that acts as a client to allow readers to effectively rate publications. There is one additional component since the task of editing files encoded in PDF format is carried out by an ad hoc software library exploited by the server side application. An overview of the architecture of Readersourcing 2.0 is shown in Figure 1.

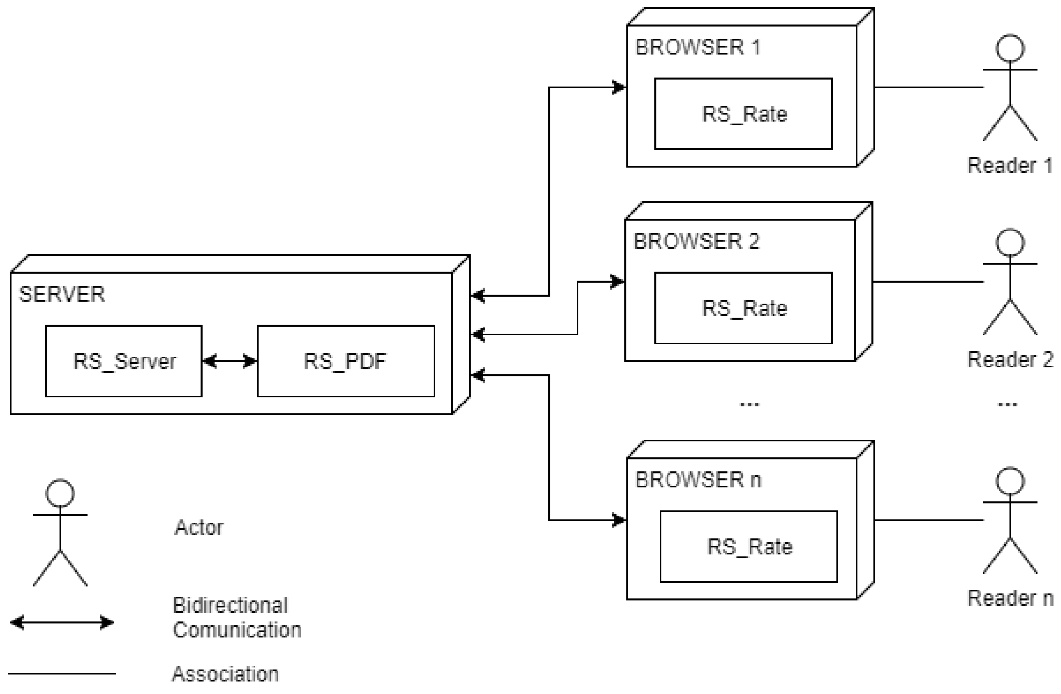

Figure 1: General architecture of Readersourcing 2.0.

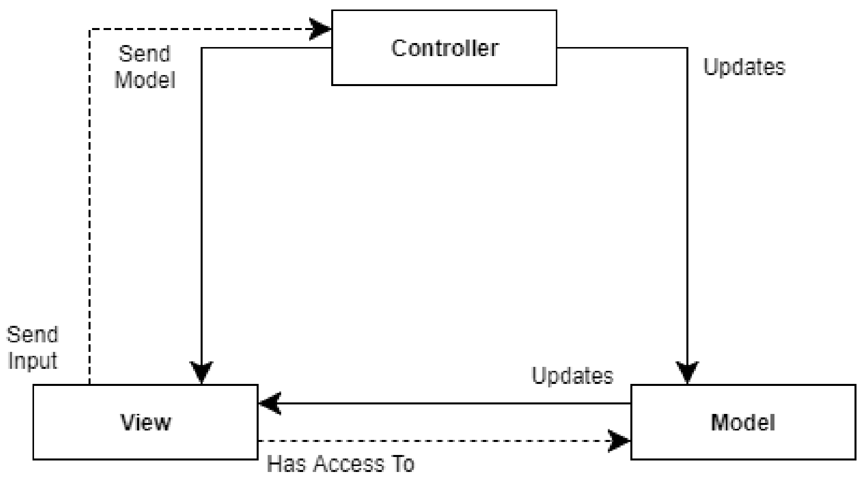

Figure 2: Intuitive scheme of the MVC pattern (NOT UML).

3 RS_Server

RS_Server [4] is the server-side application which has the task to aggregate the ratings given by readers and to use the RSA and TRA models to compute quality scores for readers and publications. An instance of the server-side application is deployed along one of *RS_PDF*. Then, there are up to n different browsers of as many users which communicate with the server and every one of them has an instance of *RS_Rate*, which is the true client. Both RS_PDF and RS_Rate will be described in the following. This setup means that every interaction between readers and server is carried out through clients installed on readers' browsers and these clients have to handle the registration and authentication of readers, the rating action and the download action of edited publications.

3.1 Implementation and Technology

The technology used to develop RS_Server is an open-source web application framework called *Ruby on Rails*[1] (it is also called *RoR* or *Rails* only; more specifically, *Rails* is the framework built above *Ruby*, the actual programming language). It allows to build applications strongly based on an architectural design pattern called *Model-View-Controller* (often abbreviated as *MVC*).

The MVC pattern allows to separate the control logic of the program from data presentation and business logic. Thefore, it allows to obtain an effective architecture since the first moments of its design phase. An intuitive representation of the structure that this pattern allows to obtain is shown in Figure 2. This structure consists of three distinct entities, called *Controller*, *Model* and *View*. These entities have the task of, respectively, managing control logic, encapsulating business logic and implementing data presentation.

[1] `https://rubyonrails.org/`

The Controller has direct access to the Model and to the View; from the latter, generally, it receives the user input and on its basis the Controller itself updates the internal state of the Model using its methods. Finally, the Controller sends the updated Model to the View, which is then exploited by the View itself to obtain and display the results of the processing. A generic software can have more than one Controller, where each of them can manage more than one Model instances. In MVC frameworks dedicated to the development of web applications such as Rails, in fact, it is common practice to have a number of Controllers equal to the number of entities modeled within the application domain. Furthermore, there may be more than one View implementation to present the internal state of a specific type of Model.

The use of MVC pattern is not the only founding principle of Rails. One of the most important principles on which Rails itself is based for the developement of quality applications is "Convention Over Configuration". In other words, the framework tries to minimize the decisions that the developer must take during the construction of its application by adopting standard conventions that he can modify if he needs more flexibility. The founding principles of Rails can be deepened by reading the so-called *Rails Doctrine*[2].

As a last note, Rails is a continuously developing framework and is used industrially by several well-known industry players such as *GitHub*, *SoundCloud*, *Airbnb* and others. It is, therefore, a widespread and appreciated technology, for which there is an active community and a lot of learning material.

3.2 Communication Paradigm

The use of a modern MVC framework such as Rails allows to develop various kind of web applications. One of the possibilities is to create a *Web Service*, which is a software component capable of carrying out various operations made remotely available through the exchange of messages encoded in a standard interchange format such as *JSON*, all thanks to a transport layer built above the basic Internet protocols like *HTTP*. All this, however, must be carried out according to a paradigm that defines precisely what are the functionalities (resources and operations) actually available and which messages must be received in order to access them.

One of the possible communication paradigms for Web Services is *RESTful* (*REpresentational State Transfer*). Within this paradigm, the functionalities of a Web Service are represented by resources identified by different URIs and the type of HTTP message sent establishes the operation to be performed. The result of the operation initiated by the message received from the Web Service is a new message encoded according to same interchange format of the one which has been sent and it is the client's responsibility to correctly interpret and use the response of the Web Service itself.

RS_Server, therefore, is a Web Service (*Server API-Only*, according to Rails terminology) based on a communication paradigm composed of RESTful interfaces and on the exchange

[2] `https://rubyonrails.org/doctrine/`

of messages encoded in JSON format through the transport layer provided by the HTTP protocol.

The communication interface of RS_Server is constantly evolving and, for this reason, it makes no sense to fully include it in this document. However, it is possible to consult it freely and to see examples of requests that can be made by visiting the URL below.

```
https://web.postman.co/collections/4632696-c26fc049-7021-4691-b
eb3-97cebfb60adb?workspace=8a3ef37e-60b1-4b49-8782-e73d2a6e3a8c
```

To provide an example, a subset of the RESTFul interface of RS_Server is shown in Table 1. These operations are all those available to handle one of the entities of the application domain, namely the publications. Let's then suppose that a user triggers a show operation for a publications characterized by an identifier equal to 1 by visiting the corresponding endpoint. The JSON-encoded response of RS_Server would be something like the one below.

```json
{
  "id": 1,
  "doi": "10.1140/epjc/s10052-018-6047-y",
  "title": "Uncertainties in WIMP dark matter scattering revisited",
  "author": "John Ellis",
  "creator": "Springer",
  "producer": null,
  "...": ...,
  "created_at": "2018-08-02T13:27:46.988Z",
  "updated_at": "2018-08-02T13:27:49.135Z",
  "...": ...,
}
```

3.3 Database

To implement the storage of edited publications, user authentication and the other functionalities it is necessary to define the structure of a database, which is indeed shown in Figure 3. There are three entities modeled within the application domain of Readersourcing 2.0:

- **Users**: models the users of the system itself, which are characterized by their personal data and an optional *ORCID*;
- **Ratings**: models the ratings given by readers of publications which are characterized by a score;

Endpoint	**HTTP Message**	**Operation**	**Description**
/publications.json	GET	Index	Fetches the entire collection of Publications.
/publications/1.json	GET	Show	Returns the Publication with identifier equal to 1.
/publications/lookup.json	POST	Lookup	Searches for a Publication; if it doesn't exists, it is fetched from the given URL.
/publications/random.json	GET	Random	Returns a random Publication.
/publications/1/is_rated.json	GET	Is Rated	Checks if the Publication with identifier equal to 1 has been rated by at least one reader.
/publications.json	POST	Create	Creates a new Publication.
/publications/fetch.json	POST	Fetch	Fetches a Publication from the given URL.
/publications/refresh.json	GET	Refresh	Fetches again an existing Publication.
/publications/1.json	PUT	Update	Updates the Publication with identifier equal to 1.
/publications/1.json	DELETE	Delete	Deletes the Publication with identifier equal to 1.
...	...	...	...

Table 1: Subset of the RESTFul interface of RS_Server.

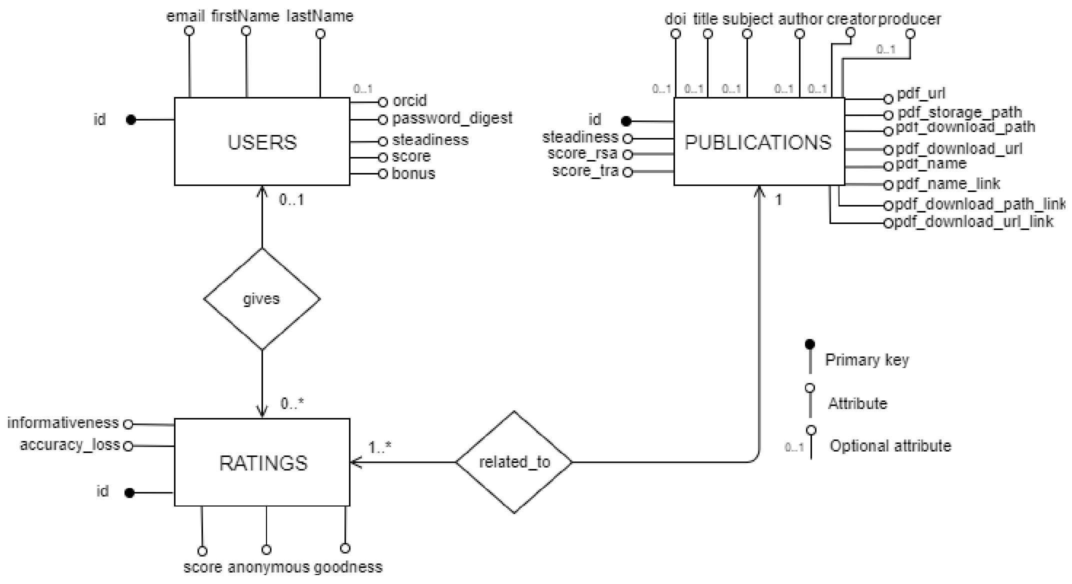

Figure 3: Entity-Relationship schema of the database (NOT UML).

- **Publications**: models the publications rated by their readers which are characterized by an optional *DOI*, by various metadata and by a whole series of attributes used to manage the paths on the server filesystem in order to guarantee a correct storage of the original and edited files encoded in PDF format.

Moreover, each of these entities is characterized by further attributes (*steadiness*, *informativeness*, . . .) which represent the scores/parameters computed by the Readersourcing models.

In the schema shown in Figure 3 are also represented two relationships (*gives* and *related_to*) that exist between these three entities. These relationships allow to "tie together" the entities to which they refer and they ensure compliance with the *referential integrity* constraint.

In particular, the *gives* relation establishes that a user can give $[0, \ldots, n]$ different ratings, while a single rating can be expressed at most by a user. At first glance, the multiplicity equal to 0 described by the schema shown in Figure 3 regarding users may seem strange. The meaning of this constraint is to allow the expression of anonymous ratings. Likewise, the relation *related_to* establishes that a rating is relative to a certain publication, while a publication can be characterized by $[1, \ldots, n]$ different ratings. Moreover, this structure allows to comply in a "natural" way with other constraints, such as the fact that if at least at least one publication does not exist, no ratings have to exist.

3.4 Class Diagram

Figure 4 shows a diagram of the main classes of RS_Server. As one can see, the convention for which there is an MVC triple for each of the entities modeled in the application domain is followed, althought Views are not shown in the diagram because in this case they are just methods. The Controller methods represent actions that a user can perform on individual entities or on collections of them, thus mapping the endpoints of the communication protocol used in order to allow the communication between RS_Server and the instances of RS_Rate. As for the Models, their attributes represent the characteristics of the reference entity, while their methods encapsulate the business logic.

Furthermore, there are two additional Controllers[3] responsible for managing user authentication. RS_Server, as specified previously, is a Web Service; this means that the user interface is presented directly on the instances of RS_Rate and, therefore, those instances send messages to which RS_Server responds once the necessary processing has been completed, according to the RESTful communication paradigm. Because of this design choice, it is not possible to use the "classic" server-side approach to user authentication according to which some information relative to the logged user are saved in the session data, since RESTful paradigm is *stateless*. To be able to authenticate himself, therefore, the user client must attach to each request a *token* that identifies its user as valid within the system. Therefore, a *token-based* authentication approach has been implemented.

When a user performs the first request to RS_Server since some time, he must fill in the login form. If these inserted credentials exist in the database they are encrypted (a *payload* is obtained) and used together with a unique *signature* to create an alphanumeric JSON string, i.e. the actual token, a copy of which is saved inside the database. This token thus generated is sent to the RS_Rate instance of the user itself which stores it in a secure cookie characterized by an expiration date after which the procedure must be repeated. At each subsequent request to RS_Sever, the instance of RS_Rate attaches[4] the previously obtained token in order to demonstrate that its user has successfully completed the authentication procedure. As for RS_Server, if a token is present it is extracted and decoded and if it corresponds to one of those saved in the database, then the user identified by the payload is authorized to proceed. An intuitive scheme of the process procedure is shown in Figure 5.

4 RS_PDF

RS_PDF [2] is the software library which is exploited by RS_Server to actually edit files encoded in PDF format. It is called into question as soon as a reader requests to save for later the publications that he's reading. More in detail, it is a software characterized by a command line interface and this means that RS_Server can use it directly since they are deployed one along the other, without using more communication channels and paradigms.

[3] *Application Controller* and *AuthenticationController*

[4] In the *Authorization* header of the HTTP package

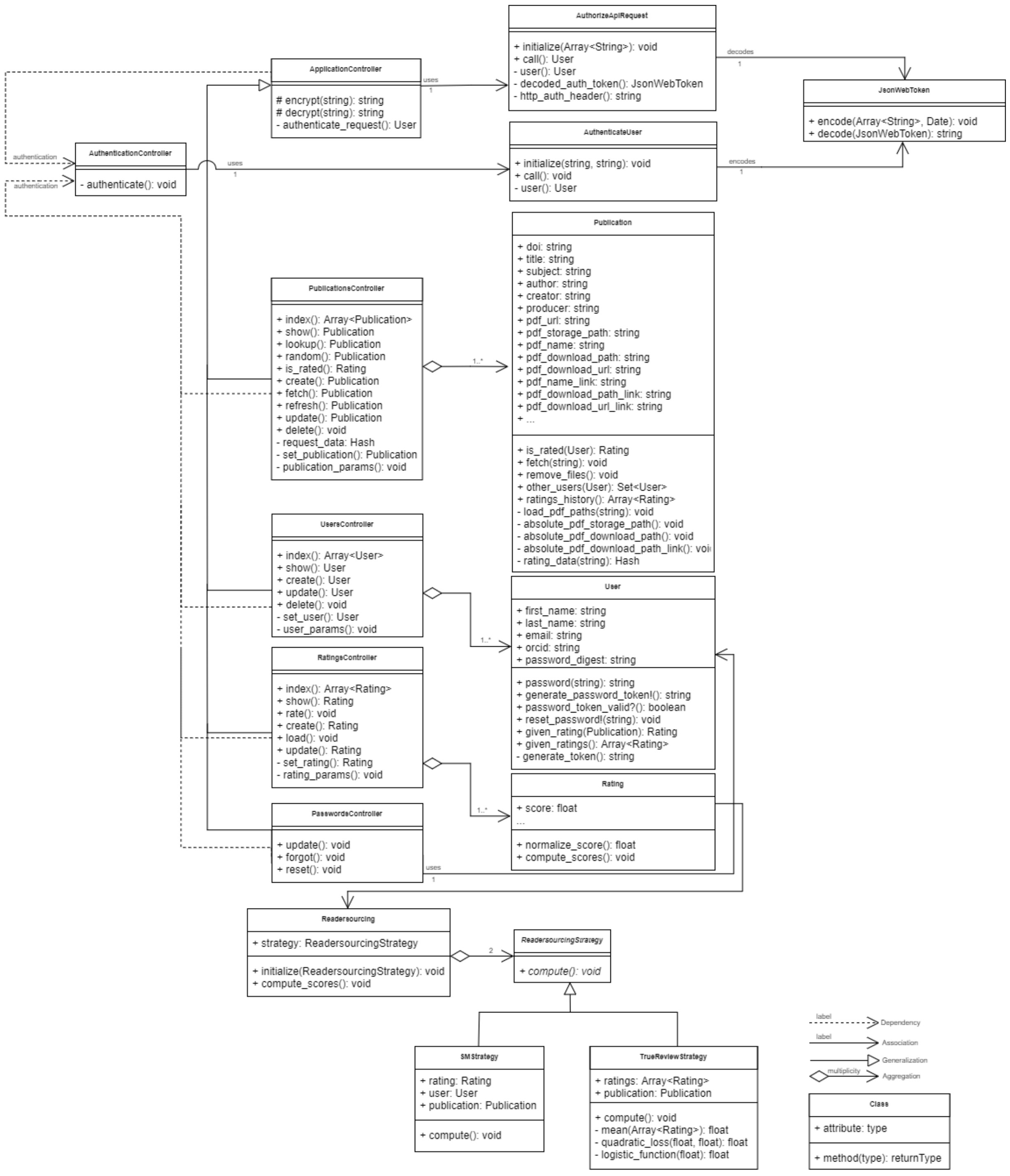

Figure 4: Class diagram of RS_Server.

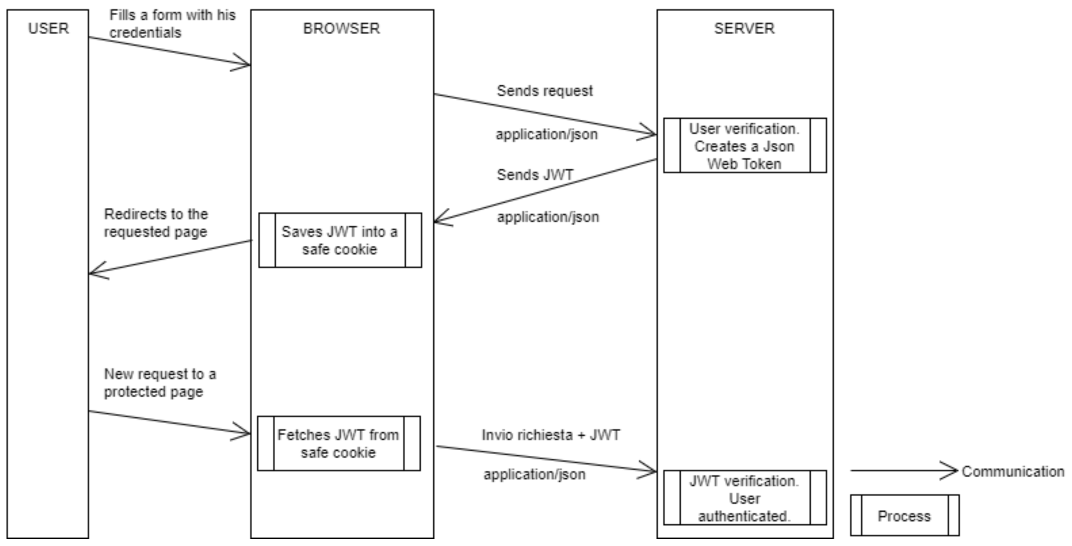
USER
Fills a form with his credentials
BROWSER
SERVER
Sends request
application/json
User verification. Creates a Json Web Token
Sends JWT
application/json
Redirects to the requested page
Saves JWT into a safe cookie
New request to a protected page
Fetches JWT from safe cookie
Invio richiesta + JWT
application/json
JWT verification. User authenticated.
Communication
Process

Figure 5: Representation of the token-based authentication process (NOT UML).

4.1 Implementation and Technology

The tecnology used to develop RS_PDF is an object-oriented programming language called *Kotlin*, which main feature is to be fully compatible with the *Java Virtual Machine*. This feature is of great importance because it allows a developer to exploit code contained in any other software published in *JAR* format and, more generally, to import any *Java* class, interacting with them through the syntax of Kotlin itself.

This programming language has been chosen because it has many modern features (it has been created just three years ago) and it is supported rather intensively; furthermore, there are openings to other platforms that have greatly expanded its use possibilities. The most important reason, however, is that the underlying tool used to actually edit files encoded in PDF format is *PDFBox*[5], which is a software library developed with Java and proposed as a complete toolkit to edit files in that specific format. So, our library is a wrapper for PDFBox that takes advantage of the modern features of Kotlin to add a reference inside a publication that a reader requests to save for later and to achieve a better integration between the components of Readersourcing 2.0.

Kotlin has been created by JetBrains[6] which, in the first half of 2017, signed an agreement with Google to let Kotlin become a first-class language for development on the Android platform[7]. In the same year, moreover, Jetbrains announced the possibility to compile programs written in Kotlin directly into machine language, thus avoiding the use of the JVM.

On the web is possible to find different pages with comparisons between Kotlin and other

[5]`https://pdfbox.apache.org/`
[6]`https://www.jetbrains.com/`
[7]`https://blog.jetbrains.com/kotlin/2017/05/kotlin-on-android-now-official/`

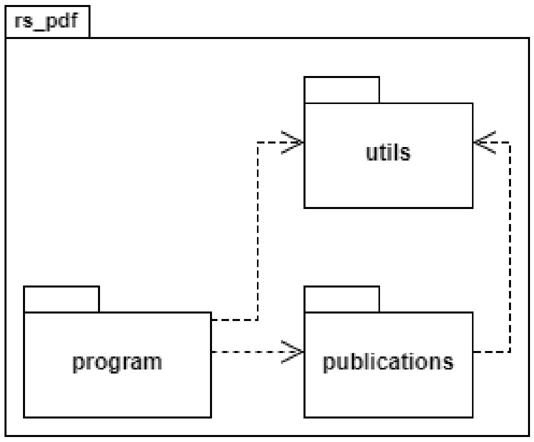

Figure 6: Package diagram of RS_PDF.

languages, including the official one[8] made by JetBrains with Java, and several articles[9] of developers enthusiastic about this programming language.

4.2 Package Diagram

Figure 6 shows a diagram of the packages in which RS_PDF is divided. This is a useful diagram since it provides a high-level overview of the internal architecture of a software.

In particular, the interaction with RS_Server takes place within the package **program**. The server-side component itself can use the functionalities of RS_PDF by executing it on the JVM, with a special set of command line options. Within this package, therefore, the parsing of the values received for each of these options and the management of the execution flow on the basis of these values take place.

The package **utils** has the task of providing useful tools to the remaining components of RS_Rate. Inside it there are shared constants and methods that allow to access to the logging functionality. As it can be seen by looking at the diagram shown in Figure 6, the other packages depend on it, in particular for some of the values of its constants.

The package **publications** contains the business logic to handle files encoded in PDF format that must be edited. Its classes follow the logic of the MVC pattern, although its exploiting is not bound by the used technology as in the case of an application developed with Rails. There is, therefore, a Controller which takes into account the execution parameters analyzed in the package **program** and updates the internal state of one or more instances of the Model which will be as many as the files encoded in PDF format that must be edited. This operation involves loading the input files and adding a link to RS_Server on a new page, taking advantage of the functionalities of *PDFBox*. As a last note, a View is not necessary because RS_PDF simply saves the changes in a new PDF file and, then, ends its execution.

[8] `https://kotlinlang.org/docs/reference/comparison-to-java.html`
[9] `https://medium.com/@octskyward/why-kotlin-is-my-next-programming-language-c25c001e26e3`

4.3 Class Diagram

Figure 7 shows a diagram of the main classes of RS_PDF which details the internal structure of the architectural elements outlined into the diagram shown in Figure 6.

The classes contained within the package **publications** are structured in a way which is similar to what Rails forces in RS_Server and most of the processing carried out by RS_Rate takes place within them. The Model contains the connections with PDFBox and its methods exploit these connections to actually edit files encoded in PDF format.

A single exception to this structure is the use of the *Parameters* class; in particular, it is only a *data class*, i.e. a class whose sole purpose is to store data of various kinds. This instance, once created, is sent to the Model by the Controller through the interfaces of the Model itself. If it is necessary to send further data, the only thing to do consists in adding them to the data class, thus avoiding modifying the signatures of the methods of the Model.

Regarding the contents of the *program* and *utils* packages, there is not much else to add with respect to what was said during the description of the diagram shown in figure 6.

4.4 Commmand Line Interface

The behavior of RS_PDF is configured during its startup phase by RS_Server through a set of special command-line options. For this reason, it is useful to provide a list of all the options that can be used if it is necessary to use RS_PDF in other contexts, modify its implementation or for any other reason. However, it is designed to work with a default configuration if no options are provided. This list of command line options in shown in Table 2.

To provide an execution example, let's assume a scenario in which there is the need of edit some files encoded in PDF format with the following prerequisites:

- there is a folder containing n files to edit at path `C:\data`;
- the edited files must be saved inside a folder at path `C:\out`;
- the file in JAR format containing the library is called `RS_PDF-v1.0-alpha.jar`;
- the JAR file containing RS_PDF is located inside the folder at path `C:\lib`;
- the authentication token received from RS_Server is `eyJhbGciOiJIUzI1NiJ9....XpC9PMXOjtjRd4NBCtB1a4SfBEi6ndgqsE3k_cEI6Wo`[10]
- the publication identifier received from RS_Server is `1`.

The execution of RS_PDF is started with the following command:

```
java -jar C:\lib\RS_PDF-v1.0-alpha.jar -pIn C:\data -pOut C:\out -a
    eyJhbGciOiJIUzI1NiJ9....XpC9PMXOjtjRd4NBCtB1a4SfBEi6ndgqsE3k_cEI6Wo -pId 1
```

[10]Token truncated for spacing reasons

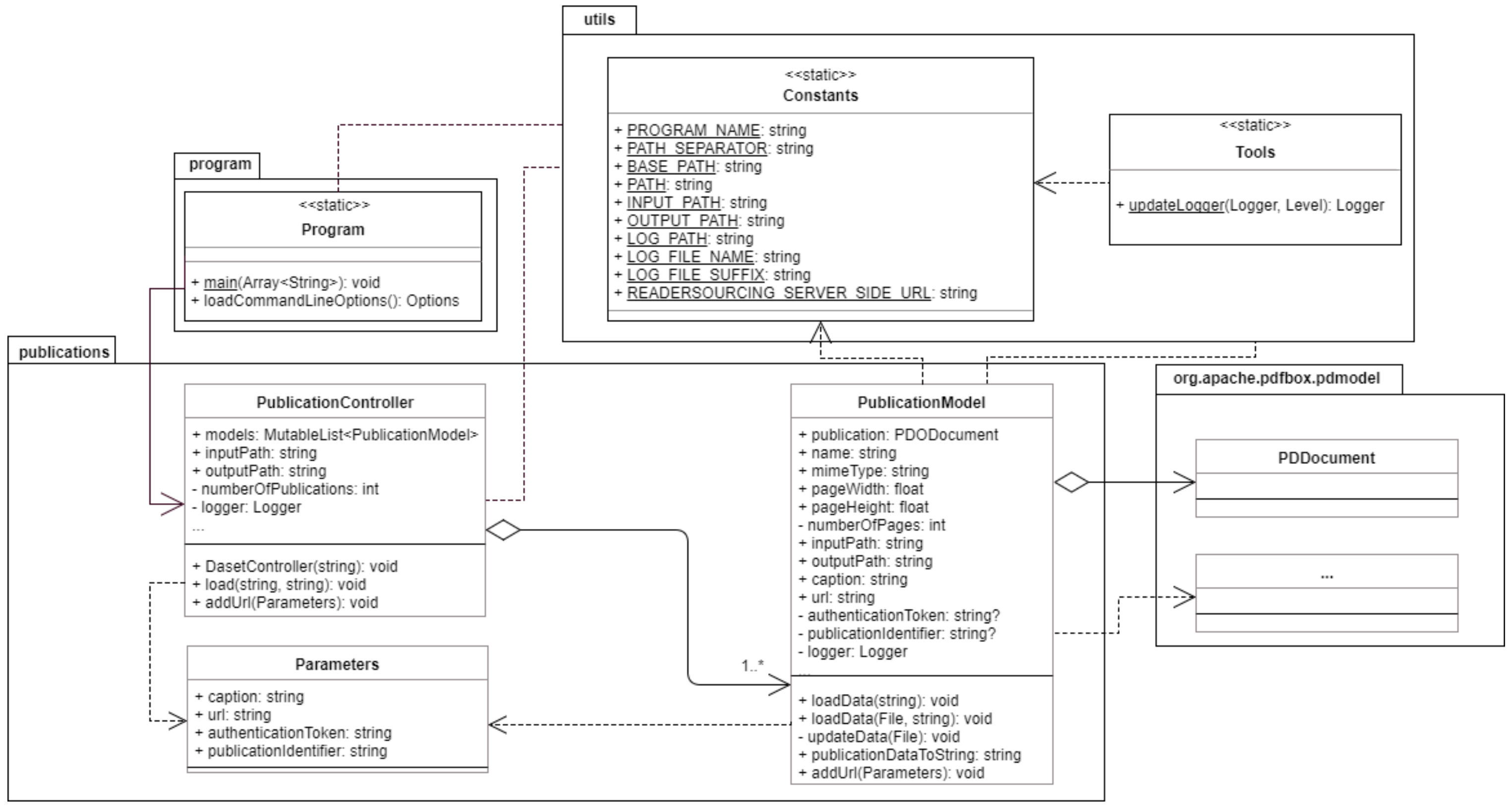

Figure 7: Class diagram of RS_PDF.

Short	Long	Description	Values	Req.	Deps.
`--pIn`	`--pathIn`	Path on the filesystem from which to load the PDF files to be edited. It can be a file or a folder.	String representing a relative path.	No	`--pOut`
`--pOut`	`--pathOut`	Path on the filesystem in which to save the edited PDF files. It must be a folder.	String representing a relative path.	No	`--pIn`
`--c`	`--caption`	Caption of the link to add.	Any string.	Yes	No
`--u`	`--url`	Url to add.	A valid URL.	Yes	No
`--a`	`--authToken`	Authentication token received from the server-side component.	A valid authentication token received from the server-side component.	No	`--pOut` `--pIn` `--pId`
`--pId`	`--publicationId`	Identifier for a publication present on the server-side component.	A valid publication identifier received from the server-side component.	No	`--pOut` `--pIn` `--a`

Table 2: Command line options of RS_PDF.

5 RS_Rate

RS_Rate [3] is an extension for *Google Chrome*[11] and the client that readers actually use to rate publications; this means that every interaction with RS_Server is carried out through this client.

5.1 Implementation and Technology

Google Chrome extensions are developed using standard web technologies such as *HTML*, *CSS* and *Javascript*. Therefore, they are simple "collections" of files packaged in a *CRX* archive. This particular format is nothing more than a modified version of a ZIP archive with the addition of some special headers exploited by Google Chrome.

As for the Javascript component, RS_Rate does not actually uses the "pure" language but instead uses jQuery, a library developed with the aim of simplifying the selection, manipulation, management of events and the animation of *DOM* elements in HTML pages, as well as implementing *AJAX* features. These AJAX features are widely used by RS_Rate to improve the user experience during its use.

[11]`https://www.google.com/chrome/`

References

[1] Martin Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. 3rd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003. ISBN: 0321193687.

[2] Michael Soprano and Stefano Mizzaro. *Readersourcing 2.0: RS_PDF*. Oct. 2018. DOI: `10.5281/zenodo.1442598`. URL: `https://doi.org/10.5281/zenodo.1442597`.

[3] Michael Soprano and Stefano Mizzaro. *Readersourcing 2.0: RS_Rate*. Oct. 2018. DOI: `10.5281/zenodo.1442599`. URL: `https://doi.org/10.5281/zenodo.1442599`.

[4] Michael Soprano and Stefano Mizzaro. *Readersourcing 2.0: RS_Server*. Oct. 2018. DOI: `10.5281/zenodo.1442630`. URL: `https://doi.org/10.5281/zenodo.1442630`.