UNIVERSITY OF UDINE

DEPARTMENT OF MATHEMATICS, COMPUTER SCIENCE AND PHYSICS

READERSOURCING 2.0: DOCUMENTATION

MICHAEL SOPRANO AND STEFANO MIZZARO

v1.0.15

Contents

List of Figures

List of Tables

1 Introduction

The Readersourcing 2.0 ecosystem has been built within a research project co-funded by SISSA Medialab[1] and University of Udine.[2] It has been presented by Soprano et al. [5] during the IRCDL 2019 Conference.[3] The paper can be found also on Zenodo.[4] The code and the related documentation is available on GitHub.[5]

Initially, a recap of its general architecture is presented, followed by a brief description of the role and purpose of each of its components. Specific aspects, such as the technology used, the internal architecture, and the structure of the database, are also discussed. This is achieved through the use of different types of diagrams adhering to the UML standard (unless otherwise specified), which are drawn according to the style rules for that standard proposed by Fowler [2].

2 General Architecture

Readersourcing 2.0 is an ecosystem composed of more than one component. Specifically, it includes RS_Server [9] (presented in Section 3), which acts as a server to gather all the ratings given by readers, and RS_Rate [8] (presented in Section 5), which acts as a client, enabling readers to rate publications. Although all operations can be carried out directly on the web interface provided by RS_Server, another component, RS_PDF [6] (presented in Section 4), is responsible for annotating PDF files using a dedicated software library. This component is utilized by the server-side application. Additionally, RS_Py (presented in Section 6) provides a fully functional implementation of the RSM and TRM models described by Soprano et al. [5].

Figure 1 provides an overview of the architecture of Readersourcing 2.0, and in the following, we briefly describe these four components. We summarize the capabilities of the ecosystem in Section 7.

3 RS_Server

RS_Server [9] is the server-side component that has the task of collecting and aggregating the ratings given by readers and using the RSM and TRM models described by Soprano et al. [5] in order to compute quality scores for readers and publications. An instance of RS_Server is deployed along one of RS_PDF. Then, there are up to n different browsers along with their end-users, which communicate with the server: each of them is characterized by an instance of RS_Rate. Both RS_PDF and RS_Rate are described in the following.

[1] `https://medialab.sissa.it/`

[2] `https://www.uniud.it/`

[3] `https://ircdl2019.isti.cnr.it/`

[4] `https://zenodo.org/record/1446468`

[5] `https://github.com/Miccighel/Readersourcing-2.0`

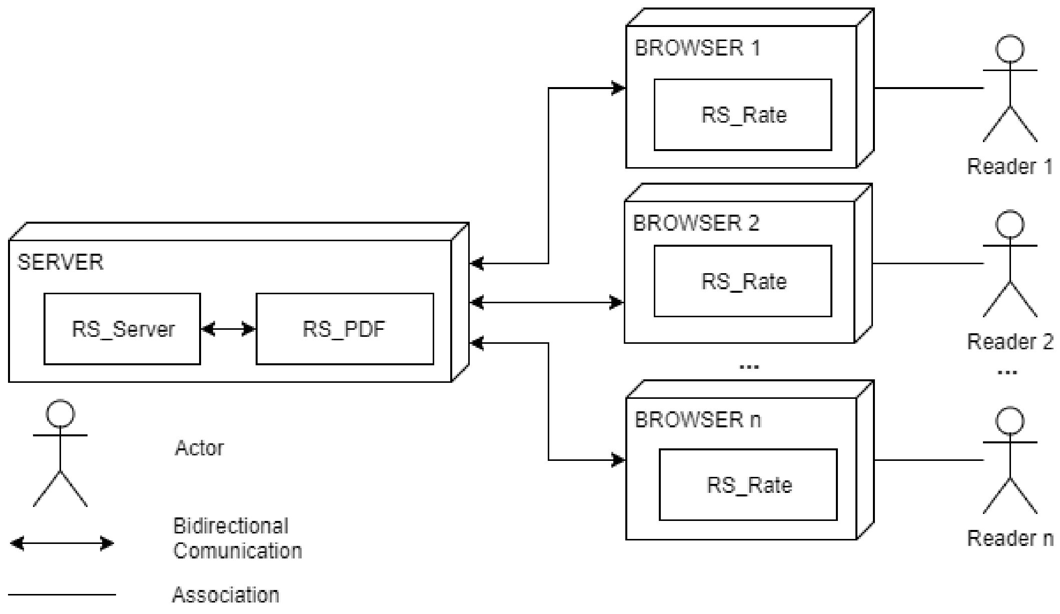


Figure 1: Architecture of Readersourcing 2.0 (NOT UML).

This setup facilitates interactions between readers and the server through clients installed on readers' browsers or by utilizing the stand-alone web interface provided by RS_Server. These clients are responsible for managing the registration and authentication of readers, handling rating actions, and managing the download actions of link-annotated publications.

During the design phase of RS_Server, strategies were adopted to ensure extensibility and generality. This includes:

(i) Straightforward addition of new models.

(ii) A shared input data format among all models.

(iii) A standard procedure for models needing to save values locally in the RS_Server database.

3.1 Implementation and Technology

RS_Server is developed in Ruby on Rails,[6] which is a framework that allows building applications strongly based on the Model-View-Controller (MVC) architectural pattern. The technology used to develop RS_Server is an open-source web application framework called *Ruby on Rails* (also referred to as *RoR* or *Rails* only). More specifically, *Rails* is the framework built above *Ruby*, the actual programming language. It allows building applications strongly based on the Model-View-Controller (MVC) architectural pattern.

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

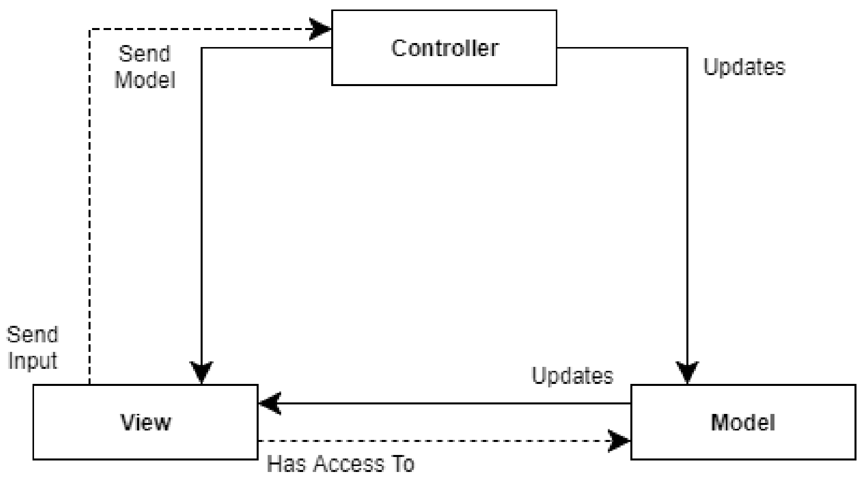


Figure 2: Intuitive scheme of the MVC pattern (NOT UML).

The MVC pattern allows for the separation of the control logic of the program from data presentation and business logic. Therefore, it facilitates the creation of an effective architecture from the initial design phase. Figure 2 provides an intuitive representation of the structure that this pattern enables. The structure comprises three distinct entities: the *Controller*, responsible for managing control logic; the *Model*, tasked with encapsulating business logic; and the *View*, responsible for implementing data presentation.

The Controller has direct access to both the Model and the View. Typically, it receives user input from the View and, based on this input, updates the internal state of the Model using its methods. Subsequently, the Controller sends the updated Model to the View, which then utilizes this information to obtain and display the results of the processing. In a generic software system, there can be more than one Controller, and each Controller may manage multiple instances of the Model. In MVC frameworks designed for web applications, such as Rails, it is common practice to have a number of Controllers equal to the entities modeled within the application domain. Additionally, there may be more than one View implementation to present the internal state of a specific type of Model.

The use of the MVC pattern is not the sole foundational principle of Rails. One of the most crucial principles shaping Rails for the development of high-quality applications is "Convention Over Configuration". In essence, the framework aims to reduce the decisions developers need to make during application construction by embracing standard conventions that can be modified for increased flexibility if necessary. To delve deeper into the foundational principles of Rails, one can refer to the "Rails Doctrine".[7]

As a final note, Rails is a dynamically evolving framework widely employed in the industry by prominent players such as *GitHub*, *SoundCloud*, and others. It is a widely embraced and esteemed technology with an active community and abundant learning resources.

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

3.2 Communication Paradigm

A modern MVC framework like Rails provides the capability to develop various types of web applications. One such possibility is the creation of a *Web Service*, which is a software component capable of executing various operations made remotely accessible through the exchange of messages encoded in a standard interchange format such as *JSON*. This is facilitated by a transport layer built on top of basic Internet protocols like *HTTP*. However, the implementation of this functionality must adhere to a paradigm that precisely defines the available functionalities (resources and operations) and specifies the required messages for accessing them.

One of the communication paradigms for Web Services is *RESTful* (*REpresentational State Transfer*). In this paradigm, the functionalities of a Web Service are represented by resources identified through different URIs, and the type of HTTP message sent determines the operation to be performed. The result of the operation initiated by the received message from the Web Service is a new message encoded in the same interchange format as the one sent. It is the client's responsibility to accurately interpret and utilize the response from the Web Service.

RS_Server is a Web Service (Server API-Only in Rails terminology) following a communication paradigm based on RESTful interfaces. It operates through the exchange of messages encoded in JSON format, utilizing the transport layer provided by the HTTP protocol.

The communication interface of RS_Server is continuously evolving. Given the dynamic nature of its development, providing a comprehensive inclusion in this document is impractical. However, interested parties can freely explore the evolving interface, including examples of requests, by visiting the following URL.

```
https://documenter.getpostman.com/view/4632696/RWTiwfV4?version=
latest
```

To illustrate, Table 1 presents a subset of the RESTful interface of RS_Server. These operations encompass all functionalities related to managing one of the entities within the application domain, specifically, publications. Suppose a user initiates a "show" operation for a publication with an identifier equal to 1 by accessing the corresponding endpoint. In response, RS_Server would provide a JSON-encoded response similar to the example 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 facilitate the implementation of functionalities such as the storage of link-annotated publications and user authentication, a database schema has been defined, as illustrated in Figure 3. Within the application domain of Readersourcing 2.0, three main entities have been modeled:

- **Users**: models the users of the system, characterized by their personal data. Users may have an optional *ORCID* and a boolean indicating whether they wish to receive emails. Additionally, various attributes are utilized to store different types of tokens, enabling operations such as password reset.
- **Ratings**: represents the ratings provided by readers for publications. These ratings are characterized by a numerical score. The entity also includes a boolean to indicate whether a rating is anonymous, and another boolean to track whether it has been edited at a later time.
- **Publications**: models the publications that have been rated by readers. These publications are characterized by an optional *DOI*, various metadata, and a set of attributes used to manage server filesystem paths. These attributes ensure the storage of both the original and link-annotated files, both encoded in PDF format.

In addition, each of these entities is further characterized by additional attributes (*steadiness*, *informativeness*, ...), representing the scores or parameters computed by the Readersourcing models.

Figure 3 also depicts two relationships (*gives* and *related_to*) that exist between these three entities. These relationships facilitate the "tying together" of the entities they reference and ensure compliance with the *referential integrity* constraint.

In particular, the *gives* relation establishes that a user can give $[0, \ldots, n]$ different ratings, with the constraint that a single rating can be expressed at most by a user. The multiplicity equal to 0 regarding users in Figure 3 may seem unusual at first glance. However, this constraint is intended to allow the expression of anonymous ratings.

Similarly, the *related_to* relation establishes that a rating is associated with a specific publication, while a publication can be characterized by $[1, \ldots, n]$ different ratings. This structure naturally accommodates other constraints, such as the condition that if at least one publication does not exist, no ratings need to exist.

Endpoint	**HTTP Message**	**Operation**	**Description**
/publications.json	GET	Index	Fetches the entire collection of Publications.
/publications/list	GET	List	Fetches the entire collection of Publications (server-side view).
/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/1/is_saved_for_later.json	GET	Is Saved For Later	Checks if the Publication with identifier equal to 1 has been saved for later by the current user.
/publications.json	POST	Create	Creates a new Publication.
/publications/is_fetchable.json	POST	Is Fetchable	Checks if the provided URL contains a fetchable Publication.
/publications/extract.json	POST	Extract	Extract the rating url from a link-annotated Publication.
/publications/fetch.json	POST	Fetch	Fetches a Publication from the given URL.
/publications/1/refresh.json	GET	Refresh	Fetches again the Publication with identifier equal to 1.
/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.

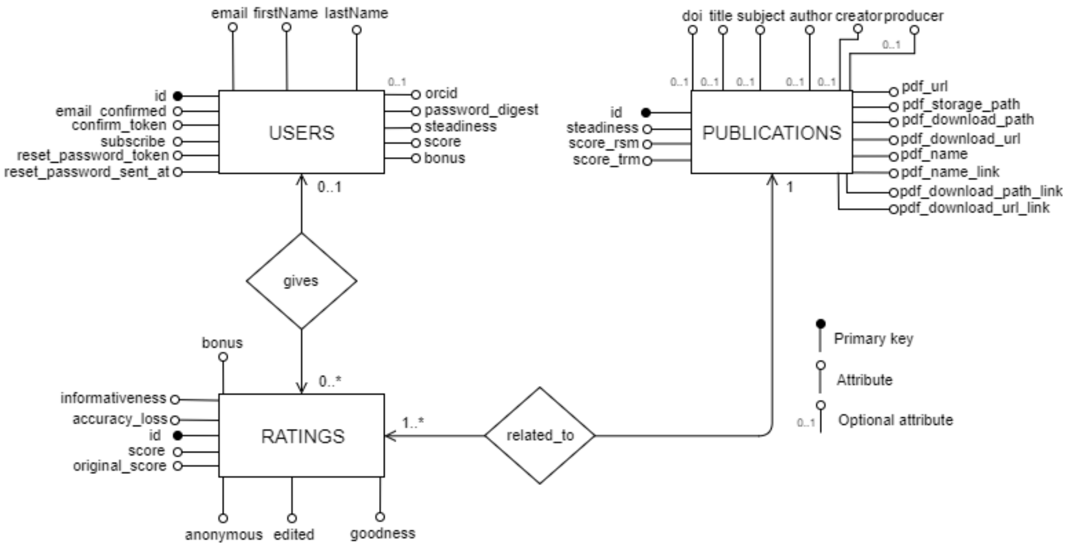


Figure 3: Entity-Relationship schema of the RS_Server database (NOT UML).

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 simple 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.

Additionally, there are two controllers—namely, the *Application Controller* and the *Authentication Controller*—tasked with managing user authentication. As mentioned earlier, RS_Server functions as a Web Service. In this setup, the user interface is presented directly on instances of RS_Rate. Consequently, these instances send messages to which RS_Server responds once the necessary processing is complete, following the RESTful communication paradigm.

Due to this design choice, the traditional server-side approach to user authentication, where information about the logged user is stored in session data, is not viable. This is because the RESTful paradigm is *stateless*. To authenticate themselves, users must attach a *token* to each request, identifying their validity within the system. Therefore, a *token-based* authentication approach has been implemented.

When a user initiates the first request to RS_Server after a period of inactivity, they are required to fill in the login form. If the provided credentials exist in the database, they are encrypted to form a *payload*. This payload, combined with a unique *signature*, results in an

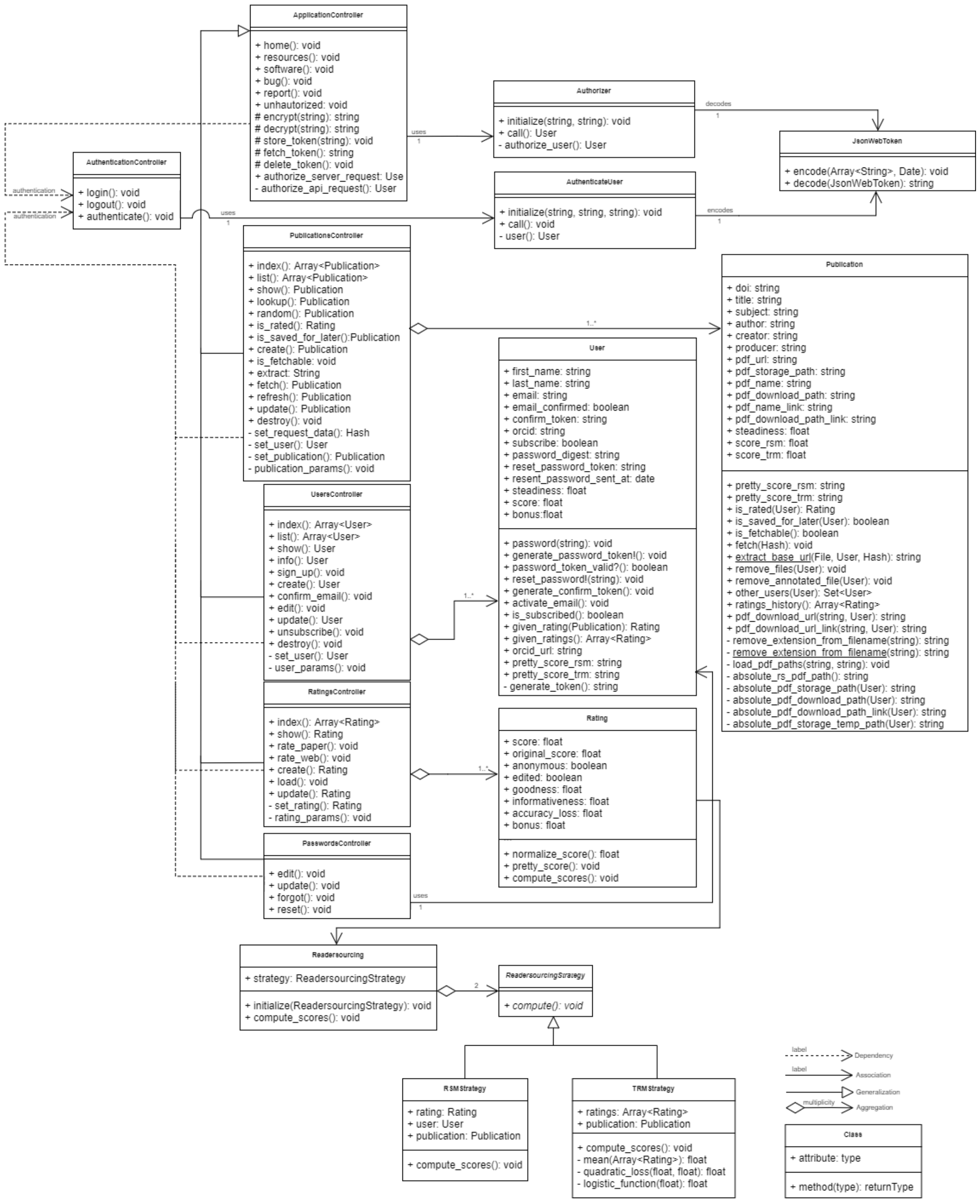


Figure 4: Class diagram of RS_Server.

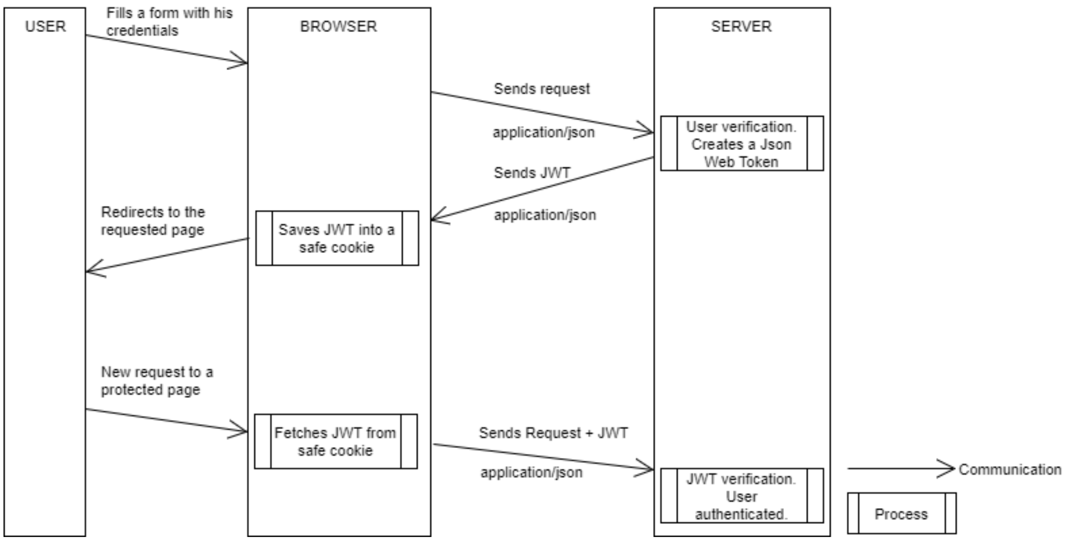

USER
Fills a form with his credentials
BROWSER
SERVER
Sends request
application/json
User verification. Creates a Json Web Token
Sends JWT
application/json
Redirects to the requested page
Saves JWT into a safe cookie
New request to a protected page
Fetches JWT from safe cookie
Sends Request + JWT
application/json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

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

alphanumeric JSON string—the actual token. The generated token is then sent to the user's RS_Rate instance, which securely stores it in a cookie with an expiration date. Subsequent requests to RS_Server from the RS_Rate instance include the previously obtained token in the *Authorization* header of the HTTP payload, demonstrating that the user has successfully completed the authentication procedure.

For RS_Server, upon receiving a request, if a token is present, it is extracted and decoded. If the decoded token corresponds to one saved in the database, the user identified by the payload is authorized to proceed. Figure 5 provides an intuitive illustration of this authentication process.

Additionally, there is another set of classes serving purposes beyond representing MVC triples for the entities in the application domain. Specifically, these classes are internally utilized by the model containing the business logic to manage given ratings and are responsible for implementing the Readersourcing models. The structure of these classes adheres to a design pattern known as *Strategy* (specifically, *Readersourcing*, *ReadersourcingStrategy*, *RSMStrategy*, and *TRMStrategy*). The use of the *Strategy* pattern allows for the integration of new models at a later time without necessitating radical changes to the structure of RS_Server.

3.5 Deploy

There are three main modalities that can be used to deploy a working instance of RS_Server in the *development* or *production* environment. The former environment must be used if there is the need to:

- add custom *Readersourcing*-based models;

- extend/modify the current implementation of RS_Server;
- simply to test it in a safe way.

In the following, three deployment modalities to obtain a working instance of RS_Server are described along with their requirements. The first two modalities allow for starting RS_Server on the local machine, enabling the editing of its source code, and building a Docker image that can be deployed via local containers in a production-ready environment. The third modality enables deploying RS_Server as a Heroku application.

It is strongly suggested to read the section dedicated to the environment variables (Section 3.5.4), as RS_Server will not function properly without them.

3.5.1 Modality 1: Manual

This modality allows for manually downloading and initializing RS_Server's codebase on a local machine. It is the most demanding in terms of prerequisites, as it assumes having a full and working installation of *Ruby*, *JDK* (Java Development Kit), and *PostgreSQL.* Despite these requirements, it offers greater flexibility.

3.5.1.1 Requirements

- Ruby == 2.7.8;
- JDK (Java Development Kit) == 11.0.19;
- PostgreSQL >= 11.2.

3.5.1.2 How To

Clone the RS_Server repository,[8] and navigate to its main directory using a command line prompt (you should see `app`, `bin`, `config`, etc., folders with an `ls` or `dir` command), then type `gem install bundler`. This gem (dependency) provides a consistent environment for Ruby projects, like RS_Server, by tracking and installing the exact gems (dependencies) and versions needed.

To fetch all dependencies required by RS_Server, type `bundle install` and wait for the process to complete.

The next two commands are necessary *only before the first startup of RS_Server* because they will create and set up the database. Ensure that the PostgreSQL service is started and ready to accept connections on port 5432. Type `rails db:create` to create the database and `rails db:migrate` to create the required tables.

Now, create a `.env` file as explained in Section 3.5.5 and set the required environment variables.

[8]`https://github.com/Miccighel/Readersourcing-2.0-RS_Server`

Optionally, you can type `rails db:seed` to seed some sample data into the database. After these commands, everything is ready to launch RS_Server in *development* or *production* mode. To do that, type `cd bin` to move inside the `bin` directory, and then type

```
rails server -b 127.0.0.1 -p 3000 -e development
```

with the proper values for `-b`, `-p`, and `-e` options. If the sample values are used, RS_Server will be started and bound to the `127.0.0.1` IP address with port `3000` and the `development` environment. Every HTTP request, therefore, must be sent to the `http://127.0.0.1:3000` address.

3.5.1.3 Quick Cheatsheet

1. `cd` to main directory;
2. `gem install bundler`;
3. `bundle install`;
4. `rails db:create`;
5. `rails db:migrate`;
6. create and populate the `.env` file;
7. `cd bin`;
8. `rails server -b <your_ip_address> -p <your_port> -e development` or `rails server -b <your_ip_address> -p <your_port> -e production`.

3.5.2 Modality 2: Manual Way (Using Docker)

This modality allows you to download and initialize RS_Server's codebase on a local machine using a faster and less frustrating approach based on Docker, despite being less flexible. Docker is a project that automates the deployment phase by distributing an *image* of an application inside a *container*.

An *image* is a lightweight, standalone, and executable package of software that includes everything needed to run an application: code, runtime, tools, libraries, and settings. This means that there is no need to manually install the runtimes, libraries, or dependencies needed to run an application since the Docker Engine will automatically initialize everything.

A *container* is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another. Only a working installation of `Docker Desktop CE (Community Edition)` is required.

3.5.2.1 Requirements

- Docker Desktop CE (Community Edition);[9]

[9] `https://www.docker.com/products/docker-desktop`

3.5.2.2 How To

Clone RS_Server repository[10] and navigate to its main directory using a command line prompt. Now, type `ls` or `dir`. You should see a `docker-compose.yml` file and a `Dockerfile`. If you do not see them, please make sure you are in the main directory of the cloned repository.

Before proceeding, *ensure that your Docker Engine is running.* Otherwise, the following commands will not work. At this point, two different scenarios can unfold.

Scenario 1: Deploy With Remote Images If there is no need to edit the source code of RS_Server, the Docker Engine can fetch the dependencies required in the `docker-compose.yml` file and initialize the application. The dependencies specified in the file are an image of PostgreSQL for the database and one of RS_Server itself, released on the Docker Hub.[11]

To do this, open the `docker-compose.yml` file and uncomment the following section and, additionally, comment out the remaining lines of code in the file.

```
----------- SCENARIO 1: DEPLOY WITH REMOTE IMAGES ----------
----------- END OF SCENARIO 1: DEPLOY WITH REMOTE IMAGES ----------
```

Next, from the command line prompt, type `docker-compose up` and wait for the process to finish. Note that it may take several minutes. Once the Docker Engine completes the process, a container with a working instance of RS_Server will be started.

Optionally, you can type `docker-compose run rails db:seed` to seed some sample data in the database. RS_Server will be started and bound to the `0.0.0.0` IP address with port `3000` and the `production` environment. Every HTTP request, therefore, must be sent to the `http://0.0.0.0:3000` address.

As can be seen, there is no need to manually start the server by specifying its IP address, port, and environment, or to create and migrate the database. The Docker Engine will perform that automatically. If you want to set a custom IP address or port or switch to the production environment, edit the `command` key inside the `docker-compose.yml` file.

To stop the container, simply type `docker-compose down`.

Scenario 2: Deploy With Local Build If the source code of RS_Server has been edited, the application must be built from scratch by the Docker Engine according to the structure specified in the `Dockerfile`. After the image build phase, the Docker Engine can fetch the required dependencies outlined in the `docker-compose.yml` file and initialize RS_Server, as in the previous scenario.

To do this, open the `docker-compose.yml` file and uncomment the following section. Additionally, comment out the remaining lines of code in the file.

```
----------- SCENARIO 2: DEPLOY WITH LOCAL BUILD ----------
----------- END OF SCENARIO 2: DEPLOY WITH LOCAL BUILD -----------
```

[10]https://github.com/Miccighel/Readersourcing-2.0-RS_Server

[11]https://cloud.docker.com/repository/docker/miccighel/rs_server

Next, from the command line prompt, type `docker-compose up` and wait for the process to finish. Note that it may take several minutes. Once the Docker Engine completes the process, a container with a working instance of RS_Server will be started and bound to the `0.0.0.0` IP address with port `3000` and the `production` environment. Therefore, every request must be sent to the `http://0.0.0.0:3000` address.

Similar to the previous scenario, there is no need to manually start the server by specifying its IP address, port, and environment or to create and migrate the database. If you want to set a custom IP address or port or switch to the production environment, edit the `command` key inside the `docker-compose.yml` file.

To stop the container, simply type `docker-compose down`.

Quick Cheatsheet

- `cd` to main directory;
- create and populate the `.env` file;
- `docker-compose up`;
- `docker-compose run rails db:seed` (optional);
- `docker-compose down` (to shutdown and undeploy).

3.5.3 Modality 3: Deploy on Heroku

Heroku is a cloud platform-as-a-service (PaaS) that simplifies building, deploying, and scaling web applications and services for developers. This deploy modality enables the use of its container registry for a Docker-based production-ready deployment of RS_Server on the platform, facilitated by the `Heroku Command Line Interface (CLI)`.[12] It's important to note that this modality can only be used with the *production* environment of the application.

Regarding the prerequisites for this modality, the developer must create an `app` on Heroku and then provision it with two addons: *PostgreSQL*[13] for the database and one for mail-related functionalities, such as *Twilio SendGrid*.[14] The Heroku tutorials provide a comprehensive overview of the platform. Additionally, a working installation of `Docker Desktop CE (Community Edition)` on the machine used for deployment is required.

3.5.3.1 Requirements

- Heroku account;
- Heroku application provisioned with:
  - PostgreSQL addon;

[12]`https://devcenter.heroku.com/articles/heroku-cli`
[13]`https://elements.heroku.com/addons/heroku-postgresql`
[14]`https://elements.heroku.com/addons/sendgrid`

    - a mail-related addon such as Twilio SendGrid;

- Heroku CLI;
- Docker Desktop CE (Community Edition);

3.5.3.2 How To

Clone RS_Server repository[15] and navigate to the main directory using a command line prompt. Now, type `ls` or `dir`. You should see a `Dockerfile`. If not, please ensure you are in the main directory of the cloned repository.

Before proceeding, *make sure that your Docker Engine is running.* Otherwise, the following commands will not work.

Log in using your credentials by typing `heroku login`. Next, log in to the Heroku container registry by typing `heroku container:login`.

To build and upload your instance of RS_Server using Docker, type

```
heroku container:push web --app your-app-name
```

When the process completes, type `heroku container:release web` to make it publicly accessible.

Optionally, you can type `heroku run rails db:seed` to seed some sample data in the database, and `heroku open` to open the browser and be redirected to the homepage of the `<your_app_name>` application.

Similar to the previous modality, there is no need to manually start the server by specifying its IP address, port, and environment, or to create and migrate the database since Heroku (through the Docker Engine) will take care of that for you.

3.5.3.3 Quick Cheatsheet

- `cd` to main directory;
- `heroku login`;
- `heroku container:login`;
- `heroku container:push web --app <your-app-name>`;
- `heroku container:release web --app your-app-name`;
- `heroku open --app your-app-name` (optional);
- set the environment variables on your Heroku app.

[15] https://github.com/Miccighel/Readersourcing-2.0-RS_Server

3.5.4 Environment Variables

Regardless of the chosen deployment modality, the developer must provide values for (at least a portion of) the *environment variables*, as they cannot be checked into a repository due to safety reasons. In the following, each of these available variables is described along with an explanation of which deployment modality requires their usage.

Environment Variable	**Description**	**Deploy Modality**	**Environment**	**Where To Set**
`SECRET_DEV_KEY`	Private key used to encrypt strings in the `development` environment.	1 - 2	`development`	`.env` file
`SECRET_PROD_KEY`	Private key used to encrypt strings in the `production` environment.	1 - 2 - 3	`production`	`.env` file, Heroku App
`POSTGRES_USER`	Username the admin user of the database	1 - 2 - 3	`development`, `production`	`.env` file, Heroku App
`POSTGRES_PASSWORD`	Password of the admin user of the database	1 - 2 - 3	`development`, `production`	`.env` file, Heroku App
`POSTGRES_DB`	Name of the database	1 - 2 - 3	`development`, `production`	`.env` file, Heroku App
`POSTGRES_HOST`	Hosting address of the database	1 - 2 - 3	`development`, `production`	`.env` file, Heroku App
`DATABASE_URL`	Full connection PostgreSQL connection string of the database.	1 - 2 - 3	`development`, `production`	`.env` file, Heroku App
`SMTP_USERNAME`	Username of the SMTP mail server	1 - 2 - 3	`production`	`.env` file, Heroku App
`SMTP_PASSWORD`	Password of the SMTP mail server	1 - 2 - 3	`production`	`.env` file, Heroku App
`SMTP_DOMAIN_NAME`	Domain of the SMTP mail server	1 - 2 - 3	`production`	`.env` file, Heroku App

Environment Variable	Description	Deploy Modality	Environment	Where To Set
`SMTP_DOMAIN_ADDRESS`	Full address of the SMTP mail server	1 - 2 - 3	`production`	`.env` file, Heroku App
`EMAIL_BUG_REPORT`	Email address to receive bug reports	1 - 2 - 3	`development`, `production`	`.env` file, Heroku App
`EMAIL_ADMIN`	Email address to receive general questions	1 - 2 - 3	`development`, `production`	`.env` file, Heroku App
`RAILS_LOG_TO_STD`	When set to `true`, forces the application to write its logs to the standard output.	1 - 2 - 3	`production`	`.env` file, Heroku App

Table 2: Environment variables of RS_Server.

3.5.5 Setting Variables

To set an environment variable in a local `.env` file, create it inside the main directory of RS_Server. Then, populate it in a `key=value` fashion. To set an environment variable in a Heroku app, simply follow the guide.[16] In Heroku terminology, environment variables are called `config vars`.

To provide an example, the following is the content of a valid `.env` file.

Listing 1 Content of a valid `.env` file.

```
SECRET_PROD_KEY=your_secret_prod_key_value
DATABASE_URL=your_postgresql_database_connection_string
SMTP_USERNAME=your_smtp_username
SMTP_PASSWORD=your_smtp_password
SMTP_DOMAIN_NAME=your_smtp_domain_name
SMTP_DOMAIN_ADDRESS=your_smtp_domain_address
EMAIL_BUG_REPORT=your_bug_report_mail
EMAIL_ADMIN=your_contact_mail
```

[16] `https://devcenter.heroku.com/articles/config-vars`

3.5.6 Sending Mails

RS_Server supports any SMTP-based mail server to send emails for tasks such as confirming user registration, reporting bugs, or recovering forgotten passwords.

Understanding the values used to populate the `SMTP_` environment variables can sometimes lead to ambiguity. Let's consider the case of the proposed add-on, Twilio Sendgrid,[17] both when deploying RS_Server manually and on Heroku. In the first case, after creating an account, you need to verify a single sender address or a whole domain using the provided `DNS records`. To integrate the service into an instance of RS_Server deployed anywhere outside Heroku, you must use a supported SMTP configuration.[18] Thus, the values of the environment variables must be in this form:

```
SMTP_USERNAME: apikey
SMTP_PASSWORD: <your_api_key_value>
SMTP_DOMAIN_NAME: <your_domain_address>
SMTP_DOMAIN_ADDRESS: smtp.sendgrid.net
```

However, while using the addon provided by Heroku,[19] the values provided for the environment variables need to be slightly different:

```
SMTP_USERNAME: <your_sendgrid_account_username>
SMTP_PASSWORD: <your_sengrid_password_account>
SMTP_DOMAIN_NAME: <your_domain_address>
SMTP_DOMAIN_ADDRESS: smtp.sendgrid.net
```

3.5.7 Connecting To The Database

A full connection string to a PostgreSQL database provided through the `DATABASE_URL` variable *takes precedence* over each `POSTGRES_` variable. It is thus important to provide the former environment variable or the set of the latter ones. This holds for both the *development* and *production* environments. Indeed, the final connection string is built as such:

```
<%= ENV['DATABASE_URL'] || "postgresql://#{ENV['POSTGRES_USER'] || 'postgres'}:#{ENV
    ['POSTGRES_PASSWORD']}@#{ENV['POSTGRES_HOST'] || 'localhost'}/#{ENV['POSTGRES_DB
    '] || 'rs_server'}" %>
```

3.5.8 Logging To The Standard Output

An instance of RS_Server deployed in development writes its logs to the standard output as the default behavior. In a *production* environment, on the other hand, the logs are written in the `logs/production.log` file. Thus, forcing Rails to write logs in the standard output using the `RAILS_LOG_TO_STD` variable can be useful for quick debugging purposes when testing the production environment.

[17] https://sendgrid.com/
[18] https://app.sendgrid.com/guide/integrate/langs/smtp
[19] https://elements.heroku.com/addons/sendgrid

4 RS_PDF

RS_PDF [6] is the software library utilized by RS_Server to edit the PDF files and add the required URL when a reader requests to save a publication for later reading.

It is characterized by a command-line interface, allowing RS_Server to use it directly. Since they are deployed together, they can communicate without the need for complex channels and paradigms.

4.1 Implementation and Technology

The technology used to develop RS_PDF is the Kotlin object-oriented programming language, known for its full compatibility with the Java Virtual Machine. This feature is crucial because it allows a developer to leverage 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 incorporates many modern features and receives robust support. Furthermore, it has openings to other platforms, greatly expanding its potential applications. The most significant reason, however, is that the underlying tool used to edit files encoded in PDF format is *PDFBox*,[20] a software library developed with Java and proposed as a complete toolkit for editing files in that specific format. Therefore, RS_PDF serves as a wrapper for PDFBox, adding the necessary links inside the PDFs requested by readers.

Kotlin has been created by JetBrains,[21] which, in the first half of 2017, signed an agreement with Google to elevate Kotlin to the status of a first-class language for development on the Android platform.[22] Moreover, in the same year, JetBrains announced the ability to compile programs written in Kotlin directly into machine language, thus avoiding the use of the JVM.

On the web, it is possible to find different pages with comparisons between Kotlin and other languages, including the official one[23] made by JetBrains with Java, and several articles[24] by developers enthusiastic about this programming language.

4.2 Package Diagram

Figure 6 depicts a diagram illustrating the packages in which RS_PDF is divided. This diagram offers a high-level overview of the internal architecture of the software, providing valuable insights into its structure.

In particular, the interaction with RS_Server takes place within the **program** package. The server-side component itself can utilize the functionalities of RS_PDF by executing it

[20]https://pdfbox.apache.org/

[21]https://www.jetbrains.com/

[22]https://blog.jetbrains.com/kotlin/2017/05/kotlin-on-android-now-official/

[23]https://kotlinlang.org/docs/reference/comparison-to-java.html

[24]https://medium.com/@octskyward/why-kotlin-is-my-next-programming-language-c25c001e26e3

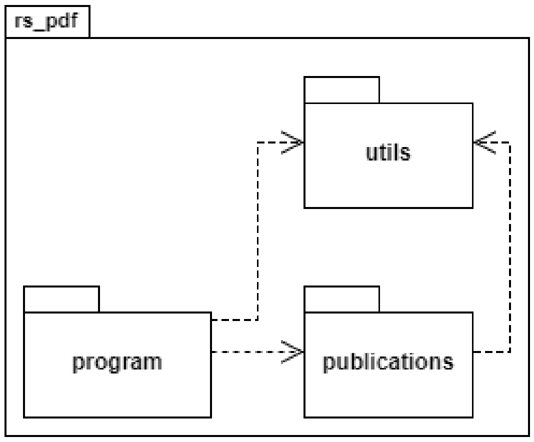


Figure 6: Package diagram of RS_PDF.

on the JVM with a special set of command-line options. Within this package, the parsing of the values received for each of these options and the management of the execution flow based on these values occur.

The **utils** package is responsible for providing useful tools to the remaining components of RS_PDF. Within this package, shared constants and methods are implemented, enabling access to the logging functionality. As evident from the diagram in Figure 6, the other packages depend on it, particularly for some of the values of its constants.

The **publications** package contains the business logic to handle files encoded in PDF format that need to be edited. Its classes adhere to the logic of the MVC pattern, although its utilization is not bound by the technology used, as in the case of an application developed with Rails. Consequently, there is a Controller that takes into account the execution parameters analyzed in the **program** package and updates the internal state of one or more instances of the Model. The number of instances of the Model corresponds to the number of PDF files that must be annotated. This operation involves loading the input files and adding a link to RS_Server on a new page, leveraging the functionalities of PDFBox. As a final note, a View is not necessary because RS_PDF simply stores the changes in a new PDF file and then concludes its execution.

4.3 Class Diagram

Figure 7 displays a diagram of the main classes of RS_PDF, providing detailed insights into the internal structure of the architectural elements outlined in the diagram shown in Figure 6. The classes within the **publications** package are structured similarly to what Rails enforces in RS_Server, with the majority of processing occurring within them. The Model establishes connections with PDFBox, and its methods leverage these connections to actively edit files encoded in PDF format.

A notable exception to this structure is the use of the *Parameters* class, which serves solely as a *data class*—a class dedicated to storing various types of data. Once created, this instance is transmitted to the Model by the Controller through the interfaces of the Model itself. If additional data needs to be sent, it can be effortlessly added to the data class, thereby avoiding modifications to the signatures of the Model's methods. As for the contents of the *program* and *utils* packages, there isn't much more to add beyond what was discussed during the description of the diagram shown in Figure 6.

4.4 Installation

RS_PDF comes bundled with RS_Server, eliminating the need for manual installation when deploying the latter. However, if you wish to use RS_PDF independently, you can simply download the attached `.jar` files from the release section of the repository[25] and place them anywhere on your filesystem.

4.4.1 Requirements

- JDK (Java Development Kit) == 11.0.19;

4.4.2 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 employ 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 is shown in Table 3.

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

- There is a folder containing n files to edit at the path `C:\data`;
- The edited files must be saved inside a folder at the 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 the path `C:\lib`;
- The authentication token received from RS_Server is `eyJhbGciOiJIUzI1NiJ9....XpC9PMXOjtjRd4NBCtB1a4SfBEi6ndgqsE3k_cEI6Wo`;
- 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
```

[25]`https://github.com/Miccighel/Readersourcing-2.0-RS_PDF`

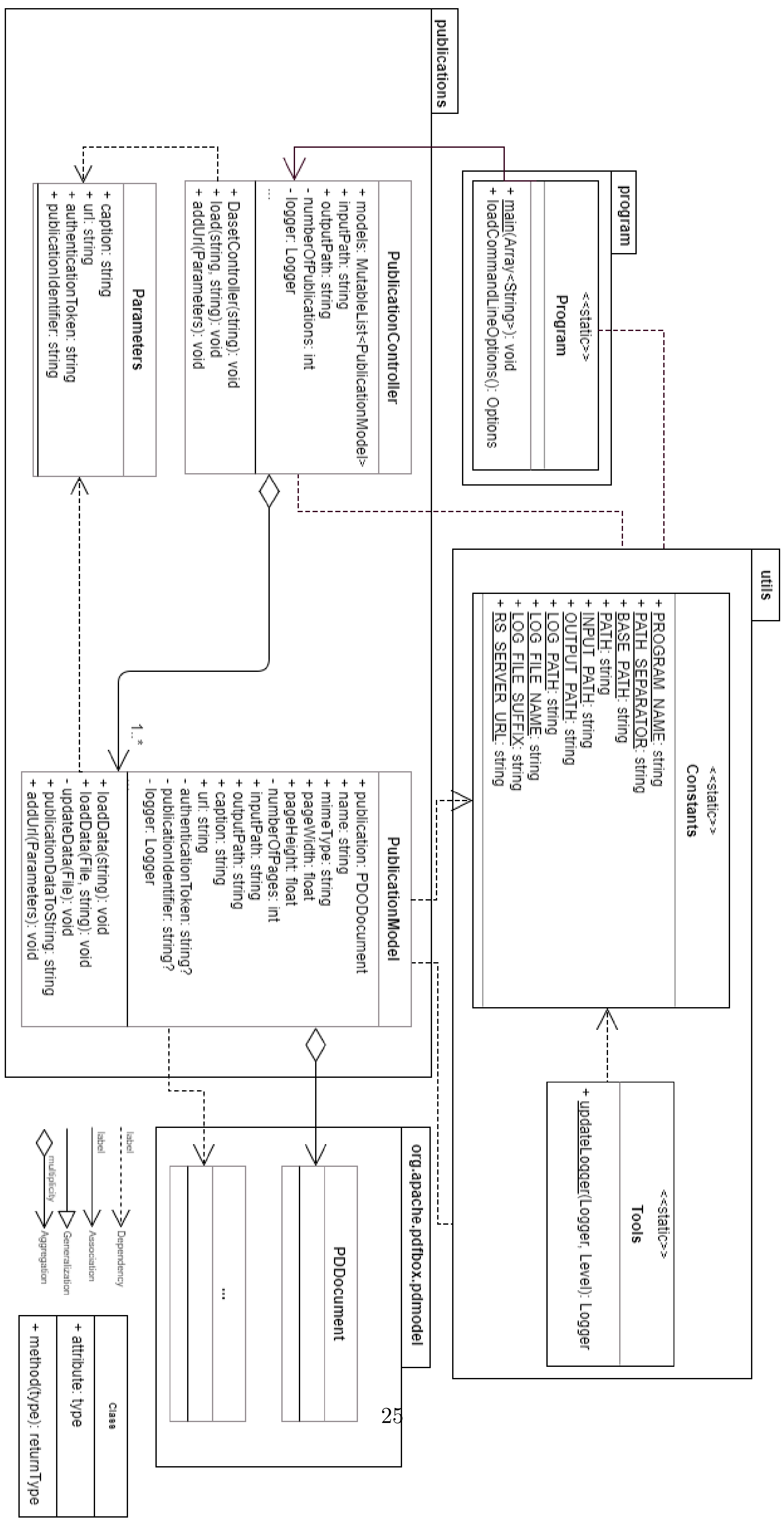


Figure 7: Class diagram of RS_PDF.

Short	Long	Description	Values	Required	Depends On
`--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 RS_Server.	A valid authentication token received from RS_Server.	No	`--pOut`, `--pIn`, `--pId`
`--pId`	`--publicationId`	Identifier for a publication present on RS_Server.	A valid publication identifier received from RS_Server.	No	`--pOut`, `--pIn`, `--a`

Table 3: Command line options of RS_PDF.

5 RS_Rate

RS_Rate [8] is an extension designed to function as a client for the Readersourcing 2.0 ecosystem without requiring access to its website. Compatible with both *Google Chrome*[26] and *Microsoft Edge*,[27] the extension allows users to rate publications directly from their browsers. This eliminates the need to navigate to the main website, streamlining the process of providing ratings for publications.

The primary objective of RS_Rate is to provide readers with a way to seamlessly rate a publication with minimal effort—just a few clicks or keystrokes within the Readersourcing 2.0 ecosystem, contributing to a more dynamic online reading experience. RS_Rate serves as the initial client of our project, extending beyond the web-based interface available on the main portal.

Looking ahead, our vision includes expanding the compatibility of RS_Rate by developing implementations for other major browsers, such as Firefox, Safari, and other popular browsers. Our commitment is to make this rating tool accessible across a broad range of browsers, ensuring users can seamlessly interact with content and provide feedback, regardless of their preferred browser.

5.1 Implementation and Technology

RS_Rate is an extension for Google Chrome and, consequently, also for Microsoft Edge. These 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 use the "pure" language but instead employs jQuery, a library developed with the aim of simplifying the selection, manipulation, management of events, and the animation of DOM elements in HTML pages. It also implements AJAX features. These AJAX features are widely utilized by RS_Rate to enhance the user experience during its use.

5.2 Installation

RS_Rate is freely available on the Google Chrome Web Store. To use it, simply take advantage of the following link and install the currently available version by clicking on the "Get" button shown on the store page. We plan to release a Firefox version in the future.

- **Google Chrome** version available at: `https://chrome.google.com/webstore/detail/readersourcing-20-rsrate/hlkdlngpijhdkbdlhmgeemffaoacjagg?hl=it`

[26]`https://www.google.com/chrome/`
[27]`https://www.microsoft.com/en-us/edge/`

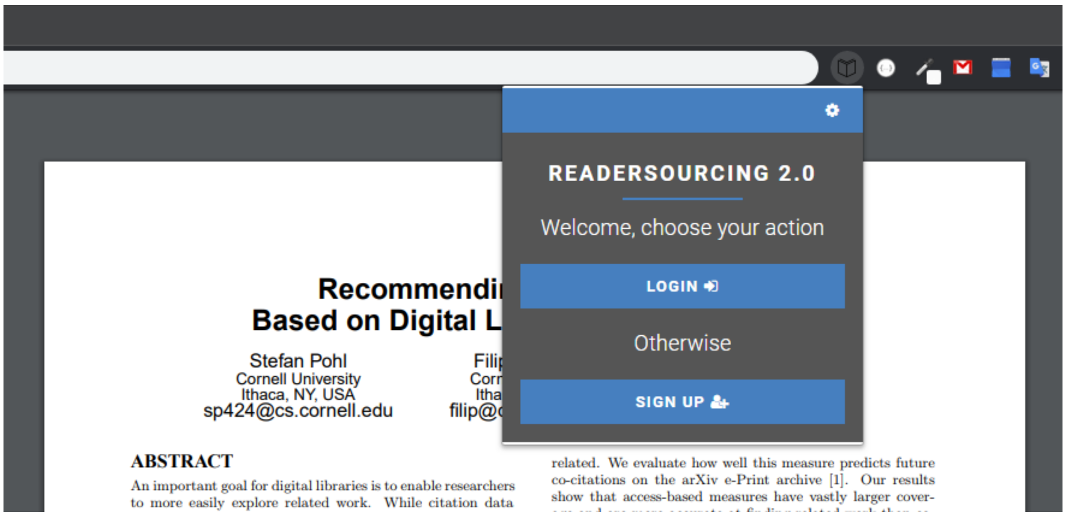


Figure 8: RS_Rate characterized as an extension having a popup action.

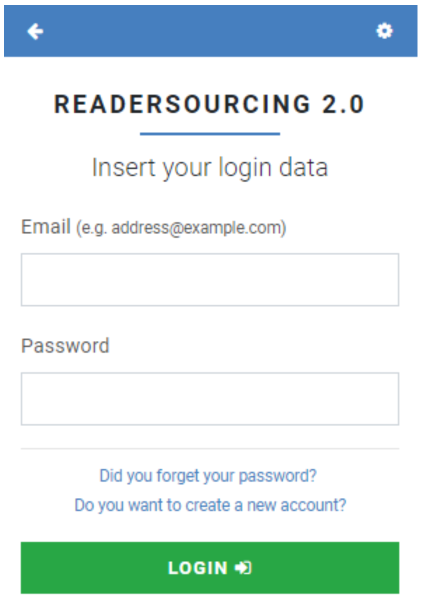
5.3 Usage

Figure 8 illustrates a section of a Google Chrome instance with the extension active for a publication. This scenario depicts the typical situation of a reader visiting a publisher's website to access the PDF of a paper they are interested in. The figure also displays the initial page that a reader encounters when interacting with the client. This page serves as a gateway to the login page, as shown in Figure 9, or to the sign-up page. From the login page, a reader who has forgotten their password can access the password recovery page (not shown), which closely resembles the login page.

If a reader has yet to sign up for Readersourcing 2.0, they can navigate from the page shown in Figure 8 to the one displayed in Figure 11 and fill in the sign-up form. Once they complete the standard sign-up and login operations, they will find themselves on the rating page, as shown in Figure 10.

In the central section of the rating page, a reader can use the slider to choose a rating value in a 0-100 interval. Once they select the desired rating, they only need to click the green *Rate* button, and that's it; with just three clicks and a slide action, they can submit their rating. Furthermore, they can also click the options button and, if preferred, check an option to anonymize the rating they are about to provide. It's important to note that the reader has to be logged in to express an anonymous rating to prevent spamming, which in this case would be a very dangerous phenomenon. When such a rating is processed, the information regarding its reader will not be used, except for preventing the reader from rating the same publication multiple times.

If the reader prefers to provide their rating at a later time instead of immediately rating the publication, they can click the *Save for later* button. This option allows them to take advantage of the editing procedure for publications, which stores a reference (an URL link)


READERSOURCING 2.0

Insert your login data

Email (e.g. address@example.com)

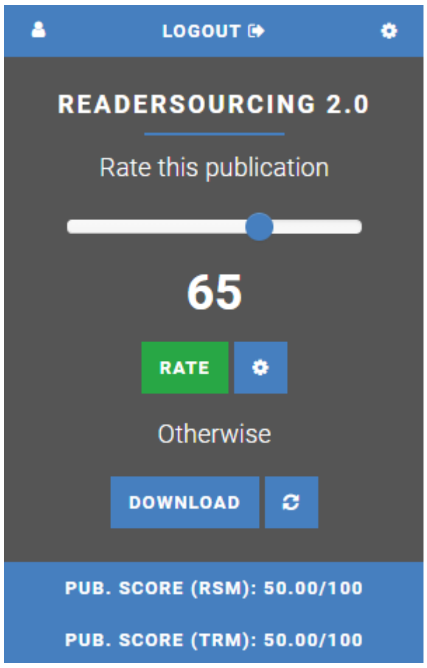
Password

Did you forget your password?

Do you want to create a new account?

LOGIN

Figure 9: The login page of RS_Rate.


LOGOUT

READERSOURCING 2.0

Rate this publication

65

RATE

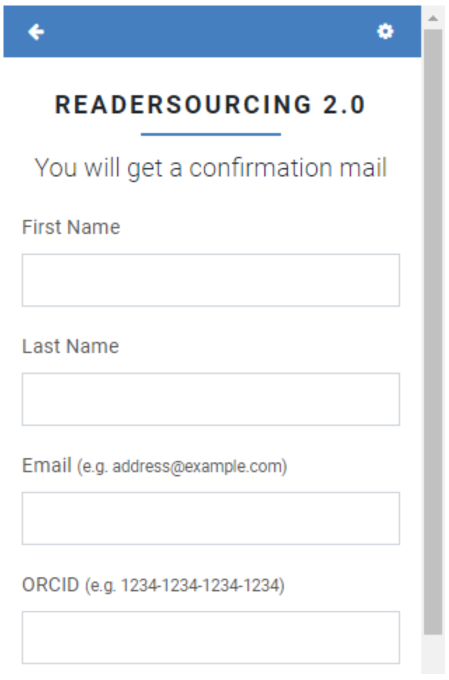
Otherwise

DOWNLOAD

PUB. SCORE (RSM): 50.00/100

PUB. SCORE (TRM): 50.00/100

Figure 10: The rating page of RS_Rate.


READERSOURCING 2.0

You will get a confirmation mail

First Name

Last Name

Email (e.g. address@example.com)

ORCID (e.g. 1234-1234-1234-1234)

Figure 11: The user registration page of RS_Rate.

inside the PDF file they are viewing. As soon as such the editing procedure is completed (usually just a few seconds), the *Save for later* button transforms into a *Download* button, as shown in Figure 12.

The reader can finally download the link-annotated publication by clicking on it. Furthermore, they can also use the refresh button (located to the right of the *Download* button) to, as it says, refresh the link-annotated publication. This means that a new copy of the publication file will be downloaded, annotated, and made available to the reader. This feature is useful since a publication could be updated at a later time by its author.

As soon as the link-annotated publication is downloaded, the reader will find a PDF containing a new final page with the URL. In Figure 13, an example of such a link-annotated publication can be seen; in that case, the reader has chosen to open it with their favorite PDF reader.

Once the reader clicks on the reference, which is a special link to RS_Server, they will be taken to the server-side application itself. The interface presented allows them to express their rating independently of the browser extension used to store the reference. Therefore, if they send their link-annotated publication to a tablet-like device, for example, they can take advantage of the built-in browser to express their rating. Figure 14 shows the interface that the reader sees after clicking on the stored reference. The reader is required to authenticate themselves again as a form of security. Without this step, the stored reference could be used by anyone who gets a copy of the link-annotated publication.

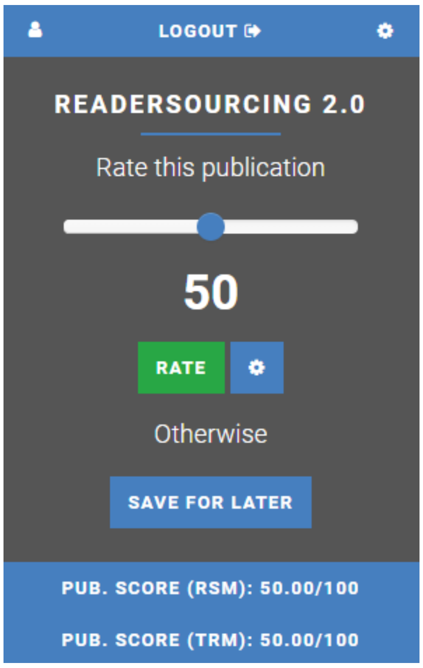
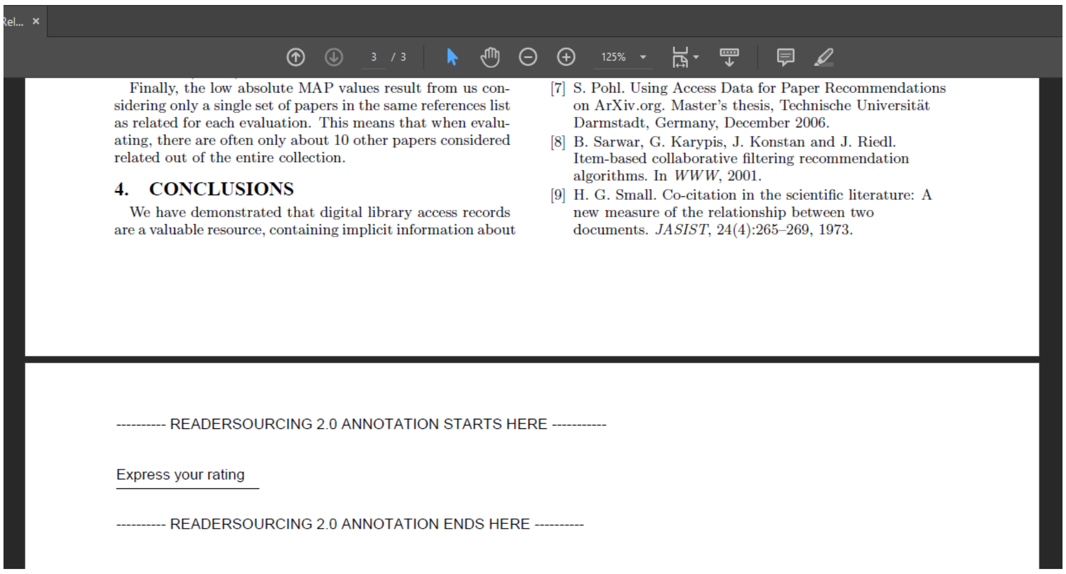


Figure 12: The rating page of RS_Rate after a “save for later” request.



Finally, the low absolute MAP values result from us considering only a single set of papers in the same references list as related for each evaluation. This means that when evaluating, there are often only about 10 other papers considered related out of the entire collection.

4. CONCLUSIONS

We have demonstrated that digital library access records are a valuable resource, containing implicit information about

[7] S. Pohl. Using Access Data for Paper Recommendations on ArXiv.org. Master's thesis, Technische Universität Darmstadt, Germany, December 2006.
[8] B. Sarwar, G. Karypis, J. Konstan and J. Riedl. Item-based collaborative filtering recommendation algorithms. In *WWW*, 2001.
[9] H. G. Small. Co-citation in the scientific literature: A new measure of the relationship between two documents. *JASIST*, 24(4):265–269, 1973.

---------- READERSOURCING 2.0 ANNOTATION STARTS HERE -----------

Express your rating

---------- READERSOURCING 2.0 ANNOTATION ENDS HERE ----------

Figure 13: A publication link-annotated through RS_Rate.

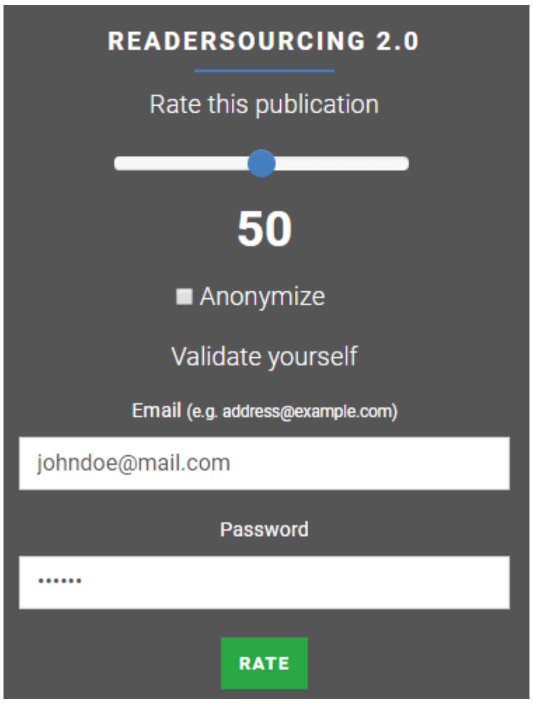


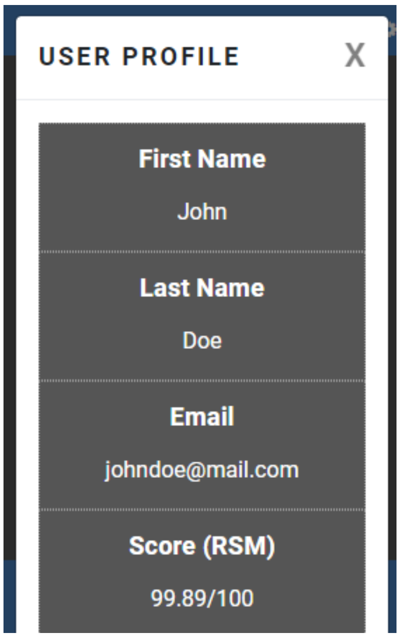
Figure 14: The server-side interface to rate a publication.

Every time a reader rates a publication, every score is updated according to both RSM and TRM models, and each reader can see the result through RS_Rate. In the bottom section of the rating page, the score of the current publication can be seen (one for each model), as shown in Figures 10 and 12. To view their score as a reader (once again, one for each model), a user must click the profile button in the upper right corner. Once they do that, they will see the interface shown in Figure 15. From there, they can also edit their password since that interface acts as a profile page.

6 RS_Py

RS_Py [7] is an additional component of the Readersourcing 2.0 ecosystem, providing a fully working implementation of the models presented by Soprano et al. [5]. These models are encapsulated by the server-side application of Readersourcing 2.0, as described in Section 3.

Developers with a background in the Python programming language can leverage RS_Py to generate and test new simulations of ratings given by readers to a set of publications. They are allowed to alter the internal logic of the models to test new approaches without the need to fork and edit the full implementation of Readersourcing 2.0.


USER PROFILE

X

First Name

John

Last Name

Doe

Email

johndoe@mail.com

Score (RSM)

99.89/100

Figure 15: The profile page of RS_Rate.

6.1 Implementation and Technology

RS_Py is a collection of *Jupyter Notebooks*. Jupyter[28] is a Python-powered[29] open-source web application that allows the creation and sharing of documents containing live code, equations, visualizations, and narrative text. Notebooks can be shared with others and are composed of cells that can be run independently, providing a step-by-step overview of the implemented computations.

6.2 Installation

To use RS_Py notebooks, it is sufficient to clone its repository[30] and place it somewhere on the filesystem. Ensure that the required Python packages are installed by leveraging a distribution such as Anaconda[31]. If a lightweight installation is preferred, an instance of Python 3.7.3 or higher is needed to install the required packages, such as Jupyter, Pandas, and others.

6.3 Usage

RS_Py is organized into five main folders on the filesystem:

[28]`https://jupyter.org/`
[29]`https://www.python.org/`
[30]`https://github.com/Miccighel/Readersourcing-2.0-RS_Py`
[31]`https://www.anaconda.com/distribution/`

- The `data` folder is used to store the dataset exploited to test the models presented by Soprano et al. [5].
- The `models` folder is used to store the output of these models.
- The `notebooks` folder contains Jupyter notebooks used to generate new datasets and implement the models presented by Soprano et al. [5].
- The `scripts` folder contains implementations of the Jupyter notebooks as pure Python scripts.
- The `src` folder contains a Python script which converts Jupyter notebooks into pure Python scripts.

Within the `notebooks` folder, three Jupyter notebooks are available:

- `Readersourcing.ipynb` provides an implementation of the RSM model presented by Soprano et al. [5].
- `TrueReview.ipynb` offers an implementation of the TRM model.
- The `Seeder.ipynb` notebook allows the generation of new datasets, which will be stored inside the `data` folder.

Inside the `src` folder, the `Convert.py` script enables the conversion of notebooks into Python scripts, and these are then stored inside the `scripts` folder.

The behavior of `Seeder.ipynb` and `Readersourcing.ipynb` notebooks can be customized by modifying the parameter settings found in the initial rows of both notebooks. Table 4 outlines the parameters available for the former, while Table 5 presents the parameters for the latter.

To run and use the Jupyter notebooks, navigate to the main directory of RS_Py using a command-line prompt (you should see folders such as `data`, `models`, `notebooks`, etc.) and type `jupyter notebook`.[32] This command will start the Jupyter server, and you can access the *Notebook Dashboard* in your browser at the web application's URL (typically, `http://localhost:8888`).

7 Overview

An overview of Readersourcing 2.0 capabilities is shown in Figure 16. Let us suppose that there are four readers using RS_Rate to rate a publication, `P1`, namely `RD1`, `RD2`, `RD3`, and `RD4`. Both `RD1`, `RD2`, and `RD3` utilize the *Save for later* functionality of RS_Rate to express their ratings at a later time. By doing this, they receive a link-annotated version of `P1`, namely `P1+Link`.

[32] `https://jupyter.readthedocs.io/en/latest/running.html#running`

Parameter	Description	Values
`dataset_name`	Name of the dataset to simulate	String
`papers_number`	Number of papers to simulate	Positive integer
`readers_number`	Number of readers to simulate	Positive integer
`authors_number`	Number of authors to simulate	Positive integer
`months_number`	Number of months of activity to simulate	Positive integer
`paper_frequencies`	Amount of papers rated by each reader group	Array of positive integers
`readers_percent`	Percentage of readers to assign to a single group	Positive integer

Table 4: Parameters available for the Seeder Jupyter notebook.

Parameter	Description	Values
`dataset_name`	Name of the dataset to simulate	String
`day_serialization`	Activate serialization of data on a per-day basis	True, False
`day_serialization_threshold`	Serialize data every X days	Positive integer
`days_number`	Amount of days simulated in the input dataset	Positive integer

Table 5: Parameters available for the Readersourcing Jupyter notebook.

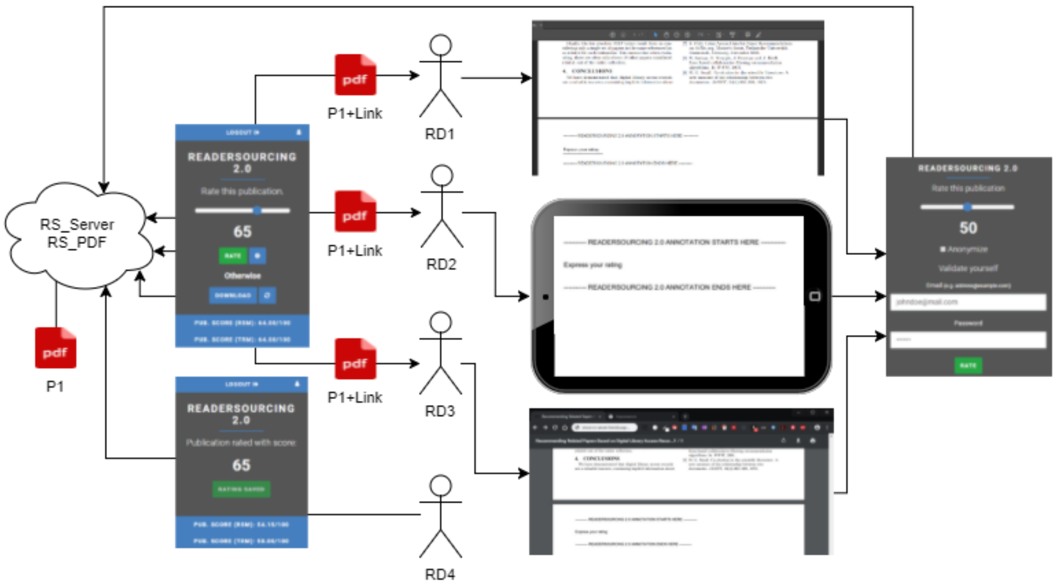


Figure 16: Readers' interaction modalities with the Readersourcing 2.0 ecosystem.

After some time, `RD1` chooses to open `P1+Link` with his favorite PDF reader. `RD2`, instead, sends it to his tablet, while `RD3` simply opens it with his instance of Google Chrome. When they click on the URL added by RS_Server, they are taken to the special page provided by RS_Server where they provide their ratings. On the contrary, `RD4` simply chooses to give his rating as soon as he finishes reading `P1` directly through the RS_Rate interface.

References

[1] Luca De Alfaro and Marco Faella. TrueReview: A Platform for Post-Publication Peer Review. In: *CoRR* abs/1608.07878 (2016). arXiv: `1608.07878`. URL: `http://arxiv.org/abs/1608.07878`.

[2] 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.

[3] Erich Gamma et al. *Design Patterns: Elementi per il Riuso di Software ad Oggetti*. Pearson, Jan. 2002.

[4] Stefano Mizzaro. Quality control in scholarly publishing: A new proposal. In: *Journal of the American Society for Information Science and Technology* 54.11 (2003), pp. 989–1005. DOI: `10.1002/asi.10296`. URL: `https://onlinelibrary.wiley.com/doi/abs/10.1002/asi.10296`.

[5] M. Soprano and S. Mizzaro. Crowdsourcing Peer Review: As We May Do. In: *Communications in Computer and Information Science* 988 (2019), pp. 259–273.

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

[7] Michael Soprano and Stefano Mizzaro. *Readersourcing 2.0: RS_Py*. DOI: `10.5281/zenodo.3245208`. URL: `https://doi.org/10.5281/zenodo.3245208`.

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

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