Renewal Recsystems

E. Madison Bray

Sep 20, 2021

CONTENTS

This documentation provides the necessary information for implementing a recommendation system ("recsystem" for short) compatible with the Renewal competition platform, as well as documentation for the sample implementation in Python which can be extended to implement your own recsystem with a minimal amount of boilerplate code.

Table of Contents

CHAPTER

ONE

INTRODUCTION

You can implement your recsystem in several ways:

1. Use the high-level interface of the reference implementation. This allows you to write your recsystem by creating a Python file containing a few functions that follow a pre-defined format. The reference system does the rest of the heavy lifting while you focus on your recommendation algorithm. This is the easiest way to get started. See the Quickstart Guide.
2. More experienced Python coders who find the high-level interface too limiting may wish to explore the lower-level reference implementation provided by the `renewal_recsystem.RenewalRecsystem` base class. This provides a lot of boilerplate functionality such as handling of the WebSocket connection and JSON-RPC details, so you can focus on just the details of your recommendation algorithms. However, it requires more experience with object-oriented programming and asyncio.
3. You may implement a recsystem completely from scratch, either in Python or any other language, as the protocol for recsystems is based on open standards. The full documentation provides the details needed to do this. Reference implementations in new languages would be a welcome contribution to the project, but may be outside the scope of a single competition (depending on the length of the competition).

A recommendation system can be written in any language: It connects to the Renewal Backend over HTTP(S) and communicates with it using standard web technologies, namely WebSockets using the JSON-RPC protocol.

The Backend also provides a RESTful API against which recsystems can make additional calls for data lookups (e.g. to fetch articles and user histories).

The REST and JSON-RPC interfaces, and all other details of implementing a recsystem are documented in detail in the full documentation.

Sample implementations in additional languages (e.g. JavaScript) may be added in the future.

1.1 Installation

Prerequisites: Python 3.7 or greater.

Using a Python "virtual environment" to install the package is highly recommended, as this ensures it will be installed in an isolated environment that will not conflict with the rest of your Python installation. See Creating a virtual environment.

To install the `renewal_recsystem` package, first clone this git repository:

```
$ git clone https://gitlri.lri.fr/renewal/recsystems.git
$ cd recsystems/
```

Then from the root of the repository, install it with:

```
$ pip install .
```

Alternatively, this can be done in a single command like:

```
$ pip install git+https://gitlri.lri.fr/renewal/recsystems.git
```

For installing to do development on the package itself, install in "editable" mode like:

```
$ pip install -e .
```

The package is not currently published on a package index like PyPI, but may be in the future.

You can make a quick check that the installation worked by running:

```
$ python -m renewal_recsystem --help
```

1.1.1 Additional steps

The following steps are good to check if you plan to do development on this package; if you only intend to use it for implementing your own recsystem it is not necessary to run the tests or build the documentation.

Running the tests

To run the tests, first install the test dependencies by changing directories to the root of the repository and running:

```
$ pip install -e .[tests]
```

Then simply run the tests with `pytest` like:

```
$ pytest
```

Building the documentation

To build the documentation for this repository, first install the documentation dependencies by changing directories to the root of the repository and running:

```
$ pip install -e .[docs]
```

Then run

```
$ cd docs/
$ make html
```

The HTML docs will be output to `_build/html`.

1.2 Baseline recommendation system

This package also implements the baseline recsystem service used by the renewal backend to provide recommendations as a baseline against which contest participant recsystems can be compared, and which provides backup recommendations when participant recsystems become unreachable.

1.2.1 Running the baseline recsystem

To run the baseline recsystem, install the package, then run:

```
$ python -m renewal_recsystem --token=<token> renewal_recsystem.baseline.popularity
```

or

```
$ python -m renewal_recsystem --token=<token> renewal_recsystem.baseline.random
```

The only required flag is `--token` to provide the authentication token for the recsystem (this is provided by an administrator when registering this recsystem with the backend). The token is a JSON Web Token (JWT) and can be provided either directly as a string, or as the path to a file containing the JWT and nothing else (recommended).

The baseline takes some other optional command line parameters which can be listed by running:

```
$ python -m renewal_recsystem --help
```

The positional parameter is the name of the Python module which implements the recommendation system itself:

- random: articles are simply returned at random
- popularity: for every batch of articles requested, the articles are returned prioritized by "popularity" as measured by their number of clicks and their overall rating by users

Each command-line flag can also be passed as an environment variable instead. The associated environment variable is the same as the flag but all uppercase and prefixed with `RENEWAL_`. For example, instead of passing `--token=<token>` you can set the environment variable `RENEWAL_TOKEN=<token>`.

New recsystems can be implemented by providing your own Python module defining some basic callback functions; see full documentation for more details.

CHAPTER

TWO

FULL DOCUMENTATION

2.1 Quickstart Guide

This provides a quick introduction to implementing a recsystem using the high-level interface to the reference implementation.

It demonstrates how a trivial recommendation system can be written in a single function, and that the `renewal_recsystem` Python package provides a simple command-line interface for running your recsystem.

2.1.1 Prerequisites

You must install the `renewal_recsystem` package for Python. Please refer to the *installation instructions*.

In order to connect your recsystem to the Renewal Backend, you must also have obtained an authentication/authorization token. Currently, this is provided directly to you by the administrator of your contest. In the future it will be provided through a registration site.

2.1.2 Bare minimal "working" recsystem

To get started on your recsystem you will create a `.py` file containing at a minimum a function named `recommend()`. This function is called every time recommendations for a user are requested from your recsystem.

It may import any other modules as needed, whether they're your own modules (if you have decided to split your code among multiple files) or third-party packages like Numpy and Pandas. But at a minimum this file is the entry-point to your recsystem.

For this tutorial we'll call it `my_recsystem.py`. Create a file with that name and containing the following code:

```
def recommend(state, user, articles, min_articles, max_articles):
    return []
```

The `recommend()` function must have exactly the same signature as given here.

This is the bare minimum "working" recsystem insofar as that the recsystem will run and connect to the backend. You can start it by running:

```
$ python -m renewal_recsystem -t <path-to-your-token> my_recsystem.py
```

It should output some log messages that look like:

```
2021-06-01 17:03:45 MyComputer my_recsystem.py[16167] INFO starting up basic recsystem␣
↪on https://api.renewal-research.com/v1/
2021-06-01 17:03:45 MyComputer my_recsystem.py[16167] INFO initializing articles cache␣
↪with 1000 articles
2021-06-01 17:03:45 MyComputer my_recsystem.py[16167] INFO initializing websocket␣
↪connection to wss://api.renewal-research.com/v1/event_stream
2021-06-01 17:03:45 MyComputer my_recsystem.py[16167] INFO ping() -> 'pong'
```

But beyond that it will never provide useful recommendations because it simply returns an empty list. We want it to return a list of articles to recommend to the user. In particular, our `recommend()` function should return a list of *article IDs* of the best articles we want to recommend to the user.

2.1.3 Testing the recsystem

In the previous section we defined a function called `recommend()` which takes some arguments. But how do we actually *call* that function in order to test it? What arguments does it take?

Actually, we never call this function directly. It is called for us whenever our recsystem receives a request from the backend for recommendations for a user.

Under normal operation, this means we would have to wait around for our recsystem to be assigned some users, and for those users to generate activity (i.e. fetching news recommendations in the mobile app).

However, for testing and development of our recsystem, there is a separate utility that allows our recsystem to be called "on demand" with test data. The results of these test calls are not used by the backend, and do not in any way impact the performance of our recsystem in a contest.

To test remote calls to your recsystem, while your recsystem is running open a separate terminal and use the test utility like:

```
$ python -m renewal_recsystem.test -t <token> <command>
```

We pass this command the same `<token>` as when running the actual recsystem, in order to authenticate to the backend. Then `<command>` is the name of any method we want the backend to call on our recsystem. For example:

```
$ python -m renewal_recsystem.test -t <token> recommend
[]
```

This prints `[]` which is the return value of the `recommend()` function we just implemented. If we look at the logs of our recsystem we also see something like:

```
2021-06-03 17:52:55 MyComputer my_recsystem.py[29595] INFO recommend(user_id='fake-user',
↪ max_articles=200, min_articles=15) -> []
```

2.1.4 The candidate articles

When you wrote the stub for your `recommend()` function it took a number of arguments: `state`, `user`, etc. Let's take a look at what those look like by augmenting the function to log their values:

```
import logging

log = logging.getLogger(__name__)

```

(continues on next page)

(continued from previous page)

```
def recommend(state, user, articles, min_articles, max_articles):
    log.info(f'state: {state}')
    log.info(f'user: {user}')
    log.info(f'articles:\n{articles}')
    log.info(f'min_articles: {min_articles}')
    log.info(f'max_articles: {max_articles}')
    return []
```

Restart your recsystem (hit Ctrl-C if it's still running) and try making another test call:

```
$ python -m renewal_recsystem.test -t <token> recommend
```

In the logs we should see something like:

```
2021-06-03 18:02:37 MyComputer my_recsystem.py[30460] INFO state: {}
2021-06-03 18:02:37 MyComputer my_recsystem.py[30460] INFO user: User(uid='fake-user',␣
↪interactions=defaultdict(<class 'dict'>, {}))
2021-06-03 18:02:37 MyComputer my_recsystem.py[30460] INFO articles:
                                                authors                 date  ...        ␣
↪                                      title                                             ␣
↪   url
article_id                                                                    ...
48573                                  [Par, La Rédaction]  2021-06-02T19:56:29  ...  France-
↪Galles : les Bleus mènent à la pause gr...  https://sport24.lefigaro.fr/football/euro-
↪2020...
48572                            [Par Le Figaro Avec Afp]  2021-06-02T20:00:38  ... ␣
↪Israël: le parti arabe Raam formalise son appu...  https://www.lefigaro.fr/
↪international/israel-l...
48571                                    [Kenneth Chang]  2021-06-02T20:04:00  ...  New␣
↪NASA Missions Will Study Venus, a World Ov...  https://www.nytimes.com/2021/06/02/
↪science/nas...
48570         [Stéphany Gardier, Par Stéphany Gardier]  2021-06-02T19:10:32  ...  Un␣
↪retour d'expérience rassurant sur des milli...  https://www.lefigaro.fr/sciences/un-
↪retour-d-e...
48569       [Vincent Bordenave, Par Vincent Bordenave]  2021-06-02T19:11:14  ...  Covid-
↪19: protéger les plus jeunes pour attein...  https://www.lefigaro.fr/sciences/covid-19-
↪prot...
...                                                  ...                 ...  ...        ␣
↪                                    ...                                               ␣
↪   ...
47578               [Paul Carcenac, Par Paul Carcenac]  2021-05-25T21:01:28  ...  Breton,
↪ belge, californien... Le cercle des va...  https://www.lefigaro.fr/sciences/breton-
↪belge-...
47577                            [Par Le Figaro Avec Afp]  2021-05-26T06:01:27  ... ␣
↪Livraisons de vaccins : l'UE et AstraZeneca s'...  https://www.lefigaro.fr/societes/
↪livraisons-de...
47576                            [Par Le Figaro Avec Afp]  2021-05-25T17:12:15  ...  Covid-
↪19 : l'Académie de médecine préconise de...  https://www.lefigaro.fr/sciences/covid-19-
↪l-ac...
47575               [Elsa Bembaron, Par Elsa Bembaron]  2021-05-25T16:27:21  ...  Le␣
↪carnet de rappel sera obligatoire à l'entré...  https://www.lefigaro.fr/secteur/high-
↪tech/le-c...
```

(continues on next page)

(continued from previous page)

```
47574                          [Par Le Figaro Avec Afp]  2021-05-26T17:13:00  ...  Covid-
↪19 : le variant indien présent dans 53 t...  https://www.lefigaro.fr/sciences/covid-19-
↪le-v...

[1000 rows x 11 columns]
2021-06-03 18:02:37 MyComputer my_recsystem.py[30460] INFO min_articles: 15
2021-06-03 18:02:37 MyComputer my_recsystem.py[30460] INFO max_articles: 200
```

For this section we are focusing on the last 3 arguments: `articles`, `min_articles`, and `max_articles`.

Of these, the last two are simply integers giving hints as to how many articles the backend wants your recsystem to return. Usually these will be the same values each time, but they may change as they are adjustable parameters. Your recsystem should return a minimum of `min_articles` recommendations on each call to `recommend()` in order for your recommendations to be considered.

The `articles` argument, on the other hand, is a Pandas DataFrame containing a collection of candidate articles to recommend to the user. This includes a backlog of past articles and new articles sent to your recsystem when it started running.

It also has pre-filtered out articles that user has already been recommended.

Note: The pre-filtering of already recommended articles is not always perfect as there may be race conditions. However, they should be mostly unique. As long as your recsystem returns a good number of results (well above `min_articles` but below `max_articles`) it should have more than enough recommendations to be considered for the user.

Further exploring the articles data

The exact format of the articles `DataFrame` is not fully documented here.

In order to more easily explore it, you could add something like:

```
articles.to_csv('articles.csv')
```

to your `recommend()` function to save the articles to a file. Then in a separate Python prompt or Jupyter Notebook open it like:

```
>>> import pandas
>>> articles = pandas.read_csv('articles.csv')
```

Note: In order debug your function, it might be a good idea to insert `breakpoint()` at the beginning of your `recommend()` function, then call `python -m renewal_recsystem.test recommend`. This will drop you into `PDB: the Python debugger` in which you can explore the value of `articles` interactively. However, in order for this to work you must also change the function definition from `async def recommend(...):`. This will be explained in a future chapter.

One of the more interesting columns is `articles.metrics`:

```
>>> articles.metrics
article_id
48573    {'bookmarks': 0, 'clicks': 0, 'dislikes': 0, '...
```

(continues on next page)

(continued from previous page)

```
48572    {'bookmarks': 0, 'clicks': 0, 'dislikes': 0, '...
48571    {'bookmarks': 0, 'clicks': 0, 'dislikes': 0, '...
48570    {'bookmarks': 0, 'clicks': 0, 'dislikes': 0, '...
48569    {'bookmarks': 0, 'clicks': 0, 'dislikes': 0, '...
                              ...
47578    {'bookmarks': 0, 'clicks': 0, 'dislikes': 0, '...
47577    {'bookmarks': 0, 'clicks': 0, 'dislikes': 0, '...
47576    {'bookmarks': 0, 'clicks': 0, 'dislikes': 0, '...
47575    {'bookmarks': 0, 'clicks': 0, 'dislikes': 0, '...
47574    {'bookmarks': 0, 'clicks': 0, 'dislikes': 0, '...
Name: metrics, Length: 1000, dtype: object
```

For each article this gives a tally of all user interactions with that article, how many users have clicked on it, liked it, etc.

We will use this for the example in the next section.

2.1.5 Simple popularity-based recsystem

What we've learned so far is enough to build a recsystem that actually makes some recommendations. For starters we'll add to `my_recsystem.py` a very simple function that measures the "popularity" of a single article given its `metrics` dict, using a very naïve metric (which you can take your own time to enhance):

```python
def popularity(metrics):
    """
    Returns a measure of an article's popularity.

    The formula is ``max(clicks, 1) * ((likes - dislikes) or 1)``.

    You could replace this with a more sophisticated measure of popularity.
    """
    clicks = metrics.get('clicks', 0)
    likes = metrics.get('likes', 0)
    dislikes = metrics.get('dislikes', 0)

    return max(1, clicks) * ((likes - dislikes) or 1)
```

Basically the articles with the most clicks are the most "popular", though it is "weighted" by the number of likes minus the number of dislikes.

Now we can sort our candidate articles from greatest to least popularity like:

```python
def recommend(state, user, articles, min_articles, max_articles):
    # Drop articles that don't have a 'metrics' dict
    articles = articles.dropna(subset=['metrics'])

    # Sort articles by most to least popular according to the
    # `popularity` function applied to their metrics dicts.
    articles = articles.sort_values(
        'metrics',
        key=lambda m: m.apply(popularity),  # type: ignore
        ascending=False)
```

(continues on next page)

(continued from previous page)

```
    # Take the top `max_articles` most popular
    articles = articles.iloc[:max_articles]
    return list(articles.index)
```

Here `articles.sort_values` sorts the articles according to their `metrics`. It takes as a sort key a function that applies the `popularity()` function to each article.

At the end we return `list(articles.index)`. The articles tables is indexed by their `article_id`, so this results in a list of the article IDs of the articles we want to recommend. Let's test it:

```
$ python -m renewal_recsystem.test -t <token> recommend
[48573, 47902, 47915, 47914, 47913, 47912, 47911, 47910, 47909, ...]
```

This should return a long list of article IDs. If you look at the logs for your recsystem you should see something similar logged.

Note: Make sure you've restarted your recsystem after making the code changes. Hot reloading isn't implemented yet!

The example we've seen here is actually used for one of the baseline recsystems: renewal_recsystem.baseline.popularity

2.2 Recsystem Interface Documentation

If you've worked though the *Quickstart Guide* you've seen the basics of how to write and run a recsystem for Renewal competitions.

That guide introduced the main function all recsystems have to implement: `recommend()`. But it did not delve into all the details that might be needed to implement a non-trivial recsystem.

This guide lists all of the other special "hook" functions you can write to make your recsystem respond to user activity from the mobile apps, and also explains how you can use the *state* dict to hold data specific to your recommendation algorithm.

2.2.1 The hook module

This is the main entry-point to your recommendation system. The *renewal_recsystem* package handles all the heavy-lifting of managing the network protocols and concurrency, so that you can just focus on writing a few functions that instruct the package on how you want to provide recommendations to users, as well as collect data on users' activity (as well as any additional background work you want the recsystem to perform).

The hook module is the Python file you pass as the argument to the `python -m renewal_recsystem` script. If your code is complex enough that it needs to grow beyond one file, your hook module is free to import code from other files, as well as from any Python packages installed on your system.

See the next section for a complete list of the functions understood by *renewal_recsystem* that you can include in your hook module.

2.2.2 Available hook functions

The following hook functions are pre-defined by the system. Some of them have exact names and some of them have naming patterns you can follow. All of them should be implemented with the exact call signatures defined here.

- *recommend*
- *article_interaction*
- *initialize and shutdown*
- *background_**
- *every_<second|minute|hour|day>*

recommend

```
def recommend(state, user, articles, min_articles, max_articles):
    ...
    return [<list of recommendations>]
```

This is the only function that is *required* to be implemented in your hook module. It is called every time one of the users assigned to your recsystem requests a list of news recommendations.

In a future version it also may be called periodically by the Backend in order to pre-queue recommendations for users, but from the perspective of your recsystem the two cases are no different (except that you should make sure to return unique sets of recommendations on each call, for each user.

- The `state` argument is explained in *Recsystem state*.
- The `user` argument is a `User` object representing the user your are making recommendations for.
- The `articles` argument is a `pandas.DataFrame` representing candidate articles to recommend to the user. To the extent possible (outside race conditions) this contains articles not already recommended to that user by any recsystem.
- The `min_articles` and `max_articles` arguments are *hints* specifying how many recommendations the function should return. If less than `min_articles` are returned, this recommendation list will be discarded by the backend, as there are not enough recommendations to make a meaningful head-to-head contest with the other recsystem(s) assigned to the user. Returning more than `max_articles` does not currently disqualify your recommendations, but recommendations beyond the first `max_articles` returned will be discarded.

article_interaction

```
def article_interaction(state, user, articles, article_id, interaction):
    ...
```

This *optional* hook function is called every time one of your recsystem's *assigned* users interacts with an article in any way. This can be used to make real-time updates to your model of the user, e.g. any statistics you are keeping of the user's preferences. The `User.interactions` attribute also keeps a tally of all the user's interactions with all articles the user has seen. This is updated automatically whenever a user interacts with an article, before your `article_interaction` function is called. For example, if a user clicked on article `1234`, then the following will be true:

```
user.interactions[1234]['clicked'] == True
```

So you don't have to keep track of these basic metrics yourself.

You can see an example implementation of `article_interaction` in the keywords-based example recsystem. This recsystem keeps scores for keywords found in articles that each user interacts with.

- The `state` argument is explained in *Recsystem state*. In the case of `article_interaction` you might use the `state` dict to track running statistics of the user's preferences, such as similarity scores.
- The `user` argument is the `User` object representing the user.
- The `articles` argument is the `pandas.DataFrame` containing the corpus of articles available to your recsystem. This is similar to the `articles` argument to *recommend* except it also contains articles the user has not interacted with yet.
- The `article_id` argument is the ID of the article the user interacted with. Thus, you can look up the full record for that article by using:

```
article = articles.loc[article_id]
```

- The `interaction` argument is a `dict` specifying the type of interaction that took place. Typically it has one or two keywords specifying the type of interaction. Here are the current possibilities:
  - `{'recommended': True}` this is a special case that just means the user recently refreshed the app and received this article as recommendation (but has not yet clicked on it or rated it).
  - `{'clicked': True}` the user clicked on the article to read it.
  - `{'rating': 1, 'prev_rating': 0}` the user "rated" the article's interest to them (whether or not they read it). The rating can be either `-1` (the article is not interesting), `1` (the article is interesting), or `0` (no opinion). You will only ever see `{'rating': 0}` if a user previously rated the article and then changed their mind. The `'prev_rating'` is the user's previous rating of the article. This can be used to recalculate scores in case a user rates an article, but then later change their minds (for example they might rate it `1`, but then read the article, decide it wasn't interesting, and change their rating to `-1`).
  - `{'bookmarked': True}` the user added the article to their bookmarks.

  More interaction details will be added in a future version, including the percent-read of the article, and geolocation details (if the user has allowed geolocation).

`initialize` and `shutdown`

```
def initialize(state, users, articles):
    ...
```

```
def shutdown(state):
    ...
```

These are lifecycle hooks that are called shortly after your recsystem starts up, and before it exits cleanly (where "cleanly" means it is not terminated forcefully such as with `kill -9`).

This can be used for any additional steps you want to perform at the startup of your recsystem, such as initialize the *state* or save the state at shutdown.

See *State persistence* and the keywords recsystem for an example of how you can load and save your state dict from a `pickle` file. Though in the future state persistence will be handled automatially (see issue #15).

background_*

```
def background_<name>(state, users, articles):
    ...
```

If you define *any* function whose name begins with `background_` (the rest of the name is up to you) that function is run repeatedly in the background in an infinite loop. For example if you have a function named `background_work` it is run (schematically) like:

```
while True:
    update = background_work(state, users, articles)
    state = apply_state_update(state, update)
```

This can be used for example to perform intensive calculations that take a long time, and that would otherwise introduce too much latency into functions like `recommend()`. For example, it could be performing running updates of similarity calculations between articles.

Warning: Be careful to use `background_` functions for work that is performed very "fast" (e.g. less than a few milliseconds). See How to Profile Your Code in Python for tips on how to measure the execution speed of your functions.

This is because every `background_` function is called repeatedly in an infinite loop, and could create a bottleneck if it is being called too often. For tasks that might be short but that you still want to call periodically, see *every_<second|minute|hour|day>*.

every_<second|minute|hour|day>

```
def every_<second|minute|hour|day>(state, users, articles):
    ...
```

or

```
def every_<n>_<seconds|minutes|hours|days>(state, users, articles):
    ...
```

These are like `background_` but allow you to define hook functions that are scheduled periodically. For example, if you write a function named `every_minute_calculate_scores` that function will be called once every minute.

Alternatively, you can use a name scheme like `every_30_seconds_calculate_scores` to run the function every 30 seconds.

The time units "seconds", "minutes", "hours", and "days" are available.

The function is re-scheduled after its last call completes. So for example if you have a function that is called every second, but it takes more than a second to complete, its next call will be one second after it completed.

In other words, you won't have multiple calls of the same periodic hook running simultaneously. So you might choose a period that represents an upper bound on the time *performance* of the hook function.

2.2.3 Recsystem state

Here we explain the use of the `state` argument that is passed as the first argument to all hook functions.

The `state` argument is a Python `dict` which may contain any number of nested dicts. It's your recsystem's own work area where it can store any data specific to your recsystem's functionality. For example, say you are performing sentiment analysis on articles. You would like to peridically compute sentiment scores for articles, and you will need a place to save these scores (in order to avoid recomputing them).

You could add a key to your `state` named `"article_sentiments"` containing a dictionary mapping article IDs to the sentiment analysis results. In this case the state (or this portion of it) could look something like:

```
{
    "article_sentiments": {
        12345: "happy",
        12346: "sad",
        12347: "neutral"
    }
}
```

Note: Technical note for the curious: You may ask "Why do I need to pass this `state` argument around? Why can't I just use a global variable?"

In many cases using a global variable will not work, because in order to keep your recsystem able to handle many events concurrently, your hook functions may be run in some separate processes. If you use a global variable for this, changes you make to its value will not be propagated correctly to the whole system.

This is also why your hook functions should return *State updates*.

The keys in the `state` dict may be any type that can be used as a dictionary key in Python (strings, integers, tuples, etc). However, keys and values must be able to be `pickled`. Fortunately, this is true for most types you will likely encounter in Python data science, such as Numpy arrays and Pandas DataFrames, etc.

State updates

Most of the *hook functions* defined may `return` a value referred to as a "state update" performed by that function. It informs the system which parts of the state you want your hook function to modify.

The state update is also a `dict`, but you *should not* simply modify the original `state` dict and return it. This could result in your hook functions overstepping each other and clobbering each other's results. Instead, each call to a hook function should only return a `dict` representing the parts of the state changed by that call. This update will be automatically merged into the "real" state that will be passed to future hook calls.

Returning to the previous example, if you have a function `every_10_seconds_perform_sentiment_analysis` to update the sentiment analysis for new articles, and it finds a new article with ID `12456`, the hook function should return a state update like:

```
{
    "article_sentiments": {
        12456: "mixed"
    }
}
```

This informs the system that there is a new key/value pair to add to `"article_sentiments"` and that no other part of the state needs to be touched.

Special case: `recommend()`

With one exception, the return value of every hook function is a state update (or no return value if you have a hook function that does not update the state). The `shutdown()` hook is also a corner case since any state update it returns will be ignored, as the system is shutting down.

However, the `recommend()` function normally returns a `list` of article IDs, not a state update. If you have a `recommend()` function in which you also want it to update the state (e.g. maybe to keep some statistics on how many recommendations it's made to each user) it can return a tuple: `(recommendations, state_update)`.

State persistence

Currently (though this might change in the future) the `state` is not *persisted* automatically. That is, when you shut down your recsystem and start it up again, it will always start with an empty state (`{}`).

Naturally, you will probably want to be able to keep your recsystem's data over the course of a competition. Currently, the best way to do that is to define some *initialize and shutdown* hooks like:

```
import logging
import os
import pickle

log = logging.getLogger(__name__)

STATE_FILENAME = __name__ + '.pickle'

def initialize(state, users, articles):
    # This runs every time your recsystem starts up
    if os.path.isfile(STATE_FILENAME):
        with open(STATE_FILENAME, 'rb') as fobj:
            state = pickle.load(fobj)

        log.info(f'loaded previous state from {STATE_FILENAME}')

    return state

def shutdown(state):
    # This runs every time you stop the recsystem cleanly and in most
    # cases if it crashes
    save_state(state)

def save_state(state):
    pickled_state = pickle.dumps(state)

    with open(STATE_FILENAME, 'wb') as fobj:
        fobj.write(pickled_state)

    log.info(f'saved updated state to {STATE_FILENAME}')
```

To protect against unfortunate catastrophes (e.g. your computer crashes) you might also want to periodically save state updates:

```
def every_30_seconds(state, users, articles):
    save_state(state)
```

Alternatives

The `state` dict is provided as a quick and convenient space to store your recsystem's data at runtime. Its use is purely optional. For example, some contestants might choose instead to use an external storage method for their recsystem's data, such as a database (SQLite, MongoDB, Redis, etc.). This is perfectly allowed.

A combination of the two can also be used, such as using `state` as a cache, but using a database for longer-term persistence. The choice is yours!

2.2.4 Async hooks functions

Todo: Explain how to write hook functions with `async def` instead of `def`, what this means, and when and why to use it.

2.3 How Recsystems Work

Contents

The ultimate goal of a recsystem is recommend news articles to a user, with an aim towards providing that user the articles that will be of the most interest to them.

How you, as a contest participant, accomplish this is entirely up to you. You can use any algorithm, such as,

Todo: List some examples of how a user might implement a recommendation system.

You can also write your recommendation system in any programming language or language(s), as well as any auxiliary tools e.g. for training your algorithm on data provided by the Renewal backend.

However, plugging your recommendation algorithm into the rest of the Renewal platform will require a minimal understanding of how the Renewal platform works, and how it connects to and interacts with your recsystem.

Your recsystem must include a "real-time" component: a software service that is always running during the duration of the contest, in order to provide news recommendations to users of the Renewal app. When users of the app refresh their news feed, their app makes a request to the backend. The backend in turn assigns the user to two more recsystems that will be placed head-to-head in competition for the user's eyeballs. At this moment each recsystem assigned to the

user is requested a batch of recommendations for that user, which are returned to the backend, and from the backend to the user's app.

Todo: Add a simplified sequence diagram showing what happens when a user refreshes their news feed. It can be based on the one at https://gitlri.lri.fr/renewal/Renewal#appendix-sequence-diagrams but simplified to omit most of the backend details.

Again, this "real-time" service can be written in most any programming language since it communicates with the Renewal backend using standard Web technologies such as WebSockets and JSON-RPC which have implementations for most popular languages, including Python, JavaScript, R, Go, etc.

The `renewal_recsystem` Python package also provides a base recsystem implementation in Python which takes care of all the boilerplate programming such as handling the WebSocket connection, so that you can focus on the parts that matter to your recsystem, simply by providing implementations for a few stub functions. However, whether you use the `renewal_recsystem` package or roll your own is entirely your choice. For details on how to use this package to implement your recsystem, see the *Quickstart Guide*.

Note: Contributions either to the `renewal_recsystem` package or of new boilerplate recsystems in other languages are also highly encouraged.

The rest of this documentation focuses on implementation of the "real-time" component of your recommendation system; that is, the software that responds in real time to recommendation requests. It may be the only component of your recsystem, or you may have various additional code run "offline" for training your models.

2.3.1 Lifecycle of a recsystem

The primary purpose of each recsystem is to respond to requests for news by users of the mobile app. Every time a user refreshes the app, the app (via communication with the backend) will request news recommendations for that user from two or more recsystems. Each recsystem connected to the backend (including yours!) is assigned one or more users for which they are currently providing recommendations. The user assignments are rotated on an occasional basis (e.g. once per week).

Responses to requests for recommendations should be *fast*–typically under one second–in order to not keep the user waiting. How this is done is up to you: For example, you can build a model of each users' preferences in the background, and then use that model to decide which news articles to send the user when they make a request.

In order to accomplish this, your recsystem is responsible for managing a few things:

- A set of users currently assigned to your recsystem.
  - If you wish, you may also build models around users *not* currently assigned to your recsystem, in case they are later assigned to you.
- A corpus of news articles to use in building your models. These are news articles provided to you by the backend, including metadata such as the article's news source (i.e. what website/newspaper it came from) title, text, keywords, etc.
- User interactions: For example, when a user clicks on and reads or rates an article, you will want to know about that in order to build your model of that user's preferences.

As such, a typical lifecycle for a recsystem is as follows:

1. Initialization

When first starting up, your recsystem will want to know:

a) To which users am I currently assigned.

b) What are some articles I can work with.

This can be accomplished by making a couple requests to the Renewal backend using its *API*. For example, to request your assigned users, make an HTTP `GET` request to `https://api.renewal-recsystems.com/v1/user_assignments`.

Currently this just returns a list of opaque user IDs like:

```
["Mhkc4xuaFPWnmbFomIv8drAtsn13","ct4LvjwHDOXdIGH1kJUtAvVQgmv1"]
```

This will be updated in the near future to return other details about the user that they have opted in to sharing with the app such as their age, location, gender, etc. It will not contain other personally identifying information such as their names or e-mail addresses.

You will also want to have some news articles that you can recommend to users. You can fetch a list of recently crawled news articles from the backend by making an HTTP `GET` request to `https://api.renewal-recsystems.com/v1/articles`. This returns a list of *article documents* that look something like:

```
{
  "article_id": 10999,
  "authors": [
    "Brooks Barnes"
  ],
  "date": "2020-09-30T01:14:41",
  "image_url": "https://static01.nyt.com/images/2020/09/25/business/25virus-disneyparks-
↪3/25virus-disneyparks-3-facebookJumbo.jpg",
  "keywords": [
    "workers",
    "unionized",
    "world",
    "newsom",
    "disneyland",
    "mr",
    "theme",
    "quarter",
    "lays",
    "florida",
    "park",
    "disney",
    "restrictions"
  ],
  "lang": "en",
  "metrics": {
    "bookmarks": 0,
    "clicks": 0,
    "dislikes": 0,
    "likes": 0
  },
  "site": {
    "icon_url": "http://localhost:8080/v1/images/icons/5f68e3404b19bc8dd873ef25",
```

(continues on next page)

(continued from previous page)

```
    "name": "NYTimes",
    "url": "www.nytimes.com"
  },
  "summary": "In Florida, where government officials have ...",
  "text": "Disneyland in California has remained closed ...",
  "title": "Disney Lays Off a Quarter of U.S. Theme Park Workers",
  "url": "https://www.nytimes.com/2020/09/29/business/disney-theme-park-workers-layoffs.
↪html"
}
```

where every article has a unique integer `article_id`. You may store these article documents however you like, whether in memory, or your own database of articles.

While the recsystem is running it is *not* necessary to make frequent requests for more articles. Instead, every time the backend scrapes a new article it will be sent to your recsystem. See the next step in the lifecycle.

To view a working example of initializing a recsystem see the source code for `BasicRecsystem.initialize`.

2. Event loop

After your recsystem is initialized it will enter an *event loop*, in which it listens for and in some cases responds to events sent to it by the backend (using *JSON-RPC*).

The most important such event will be `recommend` requests: This happens when a user assigned to your recsystem requests new articles through the app. Your recsystem will respond to this request by returning a list of `article_id`s based on your recommendation model for that user.

Most other events do not require a response, and are merely to notify your recsystem of something interesting. In particular:

- `article_interaction`: received when a user interacts with an article in any way, such as clicking on it or rating it. This notification will be in the form of an *interaction* record. You can use this event to tune your recommendation model for that user.
- `new_article`: received every time the backend crawls a new news article. You can add this to your existing database of articles in order to always return the freshest news to users.
- `assigned_user`: received every time the backend assigns a new user to your recsystem; similarly `unassigned_user`.

The full list of events your recsystem should be able to handle are documented in the *JSON-RPC API*.

For the rest of the time it is running, your recsystem is simply waiting for and responding to new events.

2.3.2 How it works

Hosting requirements

The basic networking requirements for running a recsystem are minimal. The recsystem does not act as a server; rather it only makes outgoing connections to the Renewal backend over the standard port 443 used for secure Web connections. What this means is that your recsystem can run on most any computer with an internet connection. There is no need to open a firewall for incoming connections–if your recsystem is running on a computer that can connect to websites, it can connect to Renewal.

Once connected, bi-directional communication between your recsystem and the backend is achieved using WebSockets.

Thus, the main requirement is to run your recsystem on a computer that can be expected to have reliable up-time and internet connection, since when your recsystem is down it cannot respond to recommendation requests, and its overall ranking will diminish.

Todo: Maybe provide a list of some hosting options, either at the university or publicly available.

Note: In the future the Renewal project may provide hosting for recsystems, but presently does not have the infrastructure set up.

Connecting to the backend

Your recsystem has two methods of communicating with the Renewal backend: At any time, whether while running in real-time, or for "offline" data analysis and model training, it can access the *HTTP API* to download data on articles and users from the database.

The current URL for the backend API is:

```
https://api.renewal-research.com/v1/
```

So all connection to the backend will start with HTTPS requests to endpoints under that URL.

The majority of communication your recsystem will have is via the "event stream", over which your recsystem will receive notification about events on the system–when new articles become available, when users click on articles, assignments of your recsystem to users, etc. as well as respond to requests for article recommendations for users. Your recsystem will connect to the WebSocket interface via the URI:

```
wss://api.renewal-research.com/v1/event_stream
```

All messages sent by the backend to the recsystem over WebSockets are in the form of JSON-RPC requests, and all messages sent by your backend will be in response to certain JSON-RPC requests. Your recsystem must implement the full *JSON-RPC API* documented here.

See *the next section* for an introduction to WebSockets and JSON-RPC if you are unfamiliar with these technologies. However, if you build a recsystem in Python on top of the `renewal_recsystem.RenewalRecsystem` class provided by this package, it is not necessary to fully understand how to use these protocols, as it implements all the details, and all you need to provide are implementations of some of the functions called on your recsystem via RPC.

Recsystem as WebSockets client

Because your recsystem initiates requests to the Renewal backend server, including when making a WebSockets connection, it acts as a *client* to the backend's WebSockets server. See the *WebSockets primer* for more details.

Recsystem as JSON-RPC server

WebSockets are merely a transport mechanism which can carry any type of message. For effective communication between two ends of a WebSocket connection an additional protocol is needed. The Renewal backend uses JSON-RPC for this.

However, in the JSON-RPC context your recsystem acts as a JSON-RPC *server*. That is, it provides implementations of set of methods or "procedures" which the Renewal backend calls remotely on your recsystem, and to which your recsystem returns responses. So once the WebSocket connection is established, all communications between the two ends are initiated by the backend in the form of RPC calls, and the only messages your recsystem sends are responses to (some of) these RPC calls.

Any messages sent by your recsystem that are not RPC responses are ignored. See the *JSON-RPC primer* for more details.

Authentication

Almost all requests made by your recsystem to the backend must be authenticated, including when connecting to the WebSocket interface.

Authentication is performed by passing an authentication/authorization token in the form of a JSON Web Token (JWT) which you will be provided by an administrator when registering your recsystem.

The token should be provided along with each request in the Authorization HTTP header using the `Bearer` scheme. That is, each request must send a header in the form:

```
Authorization: Bearer eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.e30.uJKHM4XyWv1bC_-
↪rpkjK19GUy0Fgrkm_pGHi8XghjWM
```

where the string after `Bearer` in this case is an example JWT to be replaced with your actual token.

Warning: The JWT token acts both to identify and authenticate your recsystem. Treat it as you would any password: Take every step to keep it private, as having this token will allow anyone to identify themselves as your recsystem.

If working in a team, you should strongly consider using a secure password manager for teams in order to share the token.

If the token is lost or revoked, contact the Renewal administrator who registered you to have the token revoked and to obtain a new one.

Todo: In the future it will be possible to revoke/regenerate recsystem tokens through the website.

2.4 Renewal Backend API

2.4.1 Introduction

2.4.2 HTTP API

These are the RESTful interfaces on the Renewal backend currently available to recsystems, e.g. to download details of articles, fetch currently assigned users, etc.

All URIs are currently relative to `/v1` as there is currently only one API version.

GET /v1

Returns the current API version.

Used just to check that the API is up.

Example request:

```
GET /v1 HTTP/1.1
Host: api.renewal-research.com
Accept: application/json
```

Example response:

```
HTTP/1.1 200
Content-Type: application/json
```

```
{"version": 1}
```

Status Codes

- 200 OK – success

GET /v1/articles

Fetch a list of article documents from the backend.

Articles are formatted according to the *Article* data type.

Although the WebSocket interface will continuously provide your recsystem with new articles as they are crawled, this allows your recsystem to fetch all the details about past articles, e.g. for training purposes, or for pre-fetching a list of recent articles when the recsystem first comes online (e.g. in `initialize`).

Example request:

```
GET /v1/articles?max_id=11000&limit=1 HTTP/1.1
Host: api.renewal-research.com
Accept: application/json
Authorization: Bearer <token>
```

Example response:

Note: The real reponse contains the full article summary and text, but they are truncated in this example.

```
HTTP/1.1 200
Content-Type: application/json
```

```
[
 {
   "article_id": 10999,
   "authors": [
     "Brooks Barnes"
   ],
   "publish_date": "2020-09-30T01:14:41",
   "image_url": "https://static01.nyt.com/images/2020/09/25/business/25virus-
↪disneyparks-3/25virus-disneyparks-3-facebookJumbo.jpg",
   "keywords": [
     "workers",
     "unionized",
     "world",
     "newsom",
     "disneyland",
     "mr",
     "theme",
     "quarter",
     "lays",
     "florida",
     "park",
     "disney",
     "restrictions"
   ],
   "lang": "en",
   "metrics": {
     "bookmarks": 0,
     "clicks": 0,
     "dislikes": 0,
     "likes": 0
   },
   "site": {
     "icon_url": "http://localhost:8080/v1/images/icons/5f68e3404b19bc8dd873ef25",
     "name": "NYTimes",
     "url": "www.nytimes.com"
   },
   "summary": "In Florida, where government officials have ...",
   "text": "Disneyland in California has remained closed ...",
   "title": "Disney Lays Off a Quarter of U.S. Theme Park Workers",
   "url": "https://www.nytimes.com/2020/09/29/business/disney-theme-park-workers-
↪layoffs.html"
 }
]
```


Query Parameters

- **limit** – the maximum number of articles to return; note that this may be limited to an internal upper-limit which is not currently specified
- **max_id** – the maximum (exclusive) `article_id` to return; all returned articles will have `article_id` less than this
- **since_id** – the minimum (exclusive) `article_id` to return; all returned articles will have `article_id` greater than this

Request Headers

- Authorization – authentication token in `Bearer <token>` format

Status Codes

- 200 OK – success
- 401 Unauthorized – unauthorized; missing or invalid authentication token

`GET /v1/articles/(int: article_id)`

Fetch a single of article document from the backend.

This is like `/v1/articles/` but just returns a single article by `article_id`.

Articles are formatted according to the *Article* data type.

Example request:

```
GET /v1/articles/10999 HTTP/1.1
Host: api.renewal-research.com
Accept: application/json
Authorization: Bearer <token>
```

Example response:

```
HTTP/1.1 200
Content-Type: application/json
```

```
{
  "article_id": 10999,
  "summary": "In Florida, where government officials have ...",
  "text": "Disneyland in California has remained closed ...",
  "title": "Disney Lays Off a Quarter of U.S. Theme Park Workers",
  "url": "https://www.nytimes.com/2020/09/29/business/disney-theme-park-workers-
↪layoffs.html"
  ...
}
```

Request Headers

- Authorization – authentication token in `Bearer <token>` format

Status Codes

- 200 OK – success
- 401 Unauthorized – unauthorized; missing or invalid authentication token
- 404 Not Found – article does not exist

`GET /v1/articles/interactions/(int: article_id)`

Return all user interactions on the article with the specified `article_id`. May optionally filter by `user_id` to get a single result (as an `object` instead of `array`) for that user's interactions with the article.

Example request:

```
GET /v1/articles/interactions/10999 HTTP/1.1
Host: api.renewal-research.com
```

(continues on next page)

(continued from previous page)

```
Accept: application/json
Authorization: Bearer <token>
```

Example response:

```
HTTP/1.1 200
Content-Type: application/json
```

```
[
  {
    "article_id": 1452,
    "prev_rating": 1,
    "rating": 1,
    "user_id": "Vf7tIKw9uMQRiZ40v6wnNBcVI2G3",
    "when": "2020-09-08T13:46:47.119Z"
  },
  {
    "article_id": 1452,
    "bookmarked": true,
    "clicked": true,
    "prev_rating": 0,
    "rating": 1,
    "user_id": "Mhkc4xuaFPWnmbFomIv8drAtsn13"
  }
]
```

Query Parameters

- **user_id** – optionally filter by `user_id`, in this case only one result `object` is returned if found

Request Headers

- Authorization – authentication token in `Bearer <token>` format

Status Codes

- 200 OK – success
- 401 Unauthorized – unauthorized; missing or invalid authentication token
- 404 Not Found – no article interactions for the given article and/or user were found

`GET /v1/user_assignments`

Get the user IDs of all users currently assigned to your recsystem, as well as their past article interaction history.

Example request:

```
GET /v1/user_assignments HTTP/1.1
Host: api.renewal-research.com
Accept: application/json
Authorization: Bearer <token>
```

Example response:

```
HTTP/1.1 200
Content-Type: application/json
```

```
[
  {
    "user_id": "Mhkc4xuaFPWnmbFomIv8drAtsn13",
    "interactions": [{"article_id": 1234, "recommended": true}]
  },
  {
    "user_id": "ct4LvjwHDOXdIGH1kJUtAvVQgmv1",
    "interactions": []
  }
]
```

Request Headers

- Authorization – authentication token in `Bearer <token>` format

Status Codes

- 200 OK – success
- 401 Unauthorized – unauthorized; missing or invalid authentication token

2.4.3 JSON-RPC API

This documents the JSON-RPC methods that the Renewal backend may make to your recsystem, and which must be implemented by your recsystem.

They are documented here as though they were implemented as Python functions but the function signatures and meanings of the parameters are transferable to any implementation language. The types of parameters and return values are given in their corresponding Pythons types. To determine what they would be in a different implementation language, you can map the Python types to the corresponding JSON types. Then look up what those types correspond to in your implementation language:

JSON	Python
`string`	`str`
`integer`	`int`
`number`	`float`
`array`	`list`
`object`	`dict`
`null`	`None`

Notification methods are indicated in the description, and do not return any value.

—

Notifications

2.4.4 Data Structures

This documentation describes the formats of data structures sent to your recsystem by the HTTP and JSON-RPC APIs used by the Renewal backend.

The data structures described first in JSON Schema format, and each is followed by an example.

Article

Article Interaction

2.5 `renewal_recsystem` API Documentation

This page contains the full generated API documentation for objects in the *`renewal_recsystem`* package.

For documentation on how recsystems interact with the Renewal backend, see the *Renewal Backend API*. For more narrative documentation on the high-level interface for writing recsystems, see *Recsystem Interface Documentation*.

`types`	Define names for built-in types that aren't directly accessible as a builtin.
`articles`	Implements the *`ArticleCollection`* class, for maintaining a size-limited cache of articles known by the system.

2.5.1 `renewal_recsystem.basic`

2.5.2 `renewal_recsystem.types`

2.5.3 `renewal_recsystem.recystem`

2.5.4 `renewal_recsystem.articles`

Implements the *`ArticleCollection`* class, for maintaining a size-limited cache of articles known by the system.

Recsystems can use this if they like, but may also replace it with a more sophisticated data store, such as a database, for maintaining a collection of articles sent to the system.

class ArticleCollection(*initial=None*, *max_size=None*)

Maintain a list of article objects sorted by article_id (ascending).

Articles are currently just represented as `dict`s and `'article_id'` is their only required key.

> **Note:** It might be a good idea to also feature validation of articles against a public schema.

MAX_ARTICLES = 10000

Maximum number of articles to keep cached in memory.

push(*item*)

Push a new article to the collection while maintaining the sort invariant.

If the new article is already than the lowest article ID and the collection is already at capacity, it is discarded.

Examples

```
>>> from renewal_recsystem.articles import ArticleCollection
>>> articles = ArticleCollection(max_size=2)
>>> articles.push({'article_id': 2})
>>> len(articles)
1
>>> articles[2]
{'article_id': 2}
```

If the article_id already exists this push is ignored (it does not merge with or overwrite the existing article):

```
>>> articles.push({'article_id': 2, 'extra': True})
>>> len(articles)
1
>>> articles[2]
{'article_id': 2}
```

If the collection is at capacity, the article with the lowest article_id is dropped on the next push:

```
>>> articles.push({'article_id': 3})
>>> articles.push({'article_id': 4})
>>> len(articles)
2
>>> 2 in articles
False
```

Unless the new article is already below the lowest article_id, then the push is ignored:

```
>>> articles.push({'article_id': 1})
>>> len(articles)
2
>>> 1 in articles
False
>>> articles.article_ids
[3, 4]
```

2.5.5 renewal_recsystem.server

2.5.6 renewal_recsystem.utils

2.5.7 renewal_recsystem.utils.testing

2.6 Primer on WebSockets and JSON-RPC

This section provides a brief introduction to the WebSocket and JSON-RPC protocols used for communication between recsystems and the Renewal backend, as well as to asynchronous I/O programming techniques (in particular on Python using asyncio, though the general concepts are transferrable to other programming languages which support asynchronous I/O). If you are already familiar with `asyncio`, you can skip straight to the *WebSockets primer*.

2.6.1 Asynchronous I/O with `asyncio`

While it is not strictly necessary to understand or even use asynchronous I/O to implement a recsystem, in practice you will find that due to the event-driven nature of WebSockets many modern programming language interfaces for dealing with WebSockets, such as on Python and JavaScript, use asynchronous I/O techniques. In particular, the *`renewal_recsystem`* package itself uses Python's asyncio library throughout, so understanding how it works requires a minimum of background learning, especially if you have never seen the async/await syntax before.

In particular, all of the below examples of WebSockets and JSON-RPC use Python and `asyncio`, so it useful to provide this additional primer.

A full introduction to asynchronous I/O with coroutines is beyond the scope of this documentation, but Async IO in Python: A Complete Walkthrough is a good, thorough explainer.

At a bare minimum you need to keep in mind the following points:

- A function defined using the `async` keyword is a *coroutine* function. For example:

```
>>> import asyncio
>>> async def ticker(ticks=5, interval=1):
...     for n in range(ticks):
...         print(f'tick {n}')
...         await asyncio.sleep(interval)
...
```

- Coroutines look like normal functions (except the `async` keyword) but they are not called like normal functions. If you try to call a coroutine like a normal function, it won't run, and you'll just get back the resulting `coroutine` object:

```
>>> ticker()
<coroutine object ticker at 0x7f5f780559e0>
```

- To run a coroutine function, you have to prefix its call with the `await` keyword, like `await ticker()`. However, you can't use `await` just anywhere, like at the command prompt, or in a normal function. For example, if you try to do this in the Python interactive prompt you'll just get:

```
>>> await ticker()
  File "<stdin>", line 1
SyntaxError: 'await' outside function
```

Note: Unless you are using recent versions of IPython, which has a special feature allowing you to use `await` in interactive prompts.

- In order to use the `await` keyword, you have to be inside another coroutine function defined with `async`. For example:

```
>>> async def run_ticker():
...     print('starting ticker coroutine')
...     await ticker()
...     print('finished ticker coroutine')
...
```

- In order to call an `async` function from *synchronous* code, i.e. from outside an `async` function, you have to pass the coroutine to the *event loop* which runs the coroutine until completion. The easiest way to do this as of Python 3.7 is to call:

```
>>> import asyncio
>>> asyncio.run(run_ticker())
starting ticker coroutine
tick 0
tick 1
tick 2
tick 3
tick 4
finished ticker coroutine
```

This `asyncio.run` call is roughly equivalent to:

```
>>> loop = asyncio.get_event_loop()
>>> loop.run_until_complete(run_ticker())
>>> loop.close()
```

Note that in both cases we did *not* put `await` in front of `run_ticker()`. Instead, this function is just passed the `coroutine` object that is returned when it is called without `await`. This is the one case where you would not run an `async` function without `await`–to kick off the event loop which runs all the coroutines.

To summarize:

- `async`/`await` always go together: If a function is defined with `async` you must call it with `await` (unless passing it directly to an event loop). Conversely, to use the `await` keyword you must be in an `async` function.
- An *event loop* is responsible for running `async` functions, i.e. *coroutines*. To kick off the process of running `async` functions you will typically wrap them in a "main" `async` function which is passed to the event loop.

Note: In some languages, such as JavaScript, coroutines are implicitly scheduled on the event loop. That is, the event loop is always running, and if call an `async` function without `await` it will be scheduled to run on the event loop, which can lead to confusing and hard to debug errors. On Python, however, you must explicitly run a coroutine on the event loop.

Common mistakes

Here are some common mistakes in programming with `async`/`await` in Python and their symptoms.

Forgetting to `await` an `async` function:

```
>>> async def run_ticker():
...     print('starting ticker coroutine')
...     ticker()
...     print('finished ticker coroutine')
...
>>> asyncio.run(run_ticker())
starting ticker coroutine
finished ticker coroutine
```

In this case you get the warning `RuntimeWarning: coroutine 'ticker' was never awaited` and you can see there is no output from `ticker()`.

Forgetting to use `await` inside an `async` function:

```
>>> def run_ticker():
...     print('starting ticker coroutine')
...     await ticker()
...     print('finished ticker coroutine')
...
  File "<stdin>", line 3
SyntaxError: 'await' outside async function
```

Trying to run a non-async function on the event loop:

```
>>> def run_ticker():
...     print('starting ticker coroutine')
...     ticker()
...     print('finished ticker coroutine')
...
>>> asyncio.run(run_ticker())
Traceback (most recent call last):
...
ValueError: a coroutine was expected, got None
```

2.6.2 WebSockets

Traditionally, communication between a Web server and a client connecting to it is stateless and mostly one-directional: A client connects to the Web server, requests a resource (e.g. an HTML page or a RESTful API), and is returned a response.

WebSockets allow a traditional HTTP request to be "upgraded" to a long-running bi-directional communication channel, where both the client and server can send messages to each other and receive responses until one side closes the connection. The contents of the messages sent over WebSockets can contain anything, so it is up to the application to determine a protocol over WebSockets that the client and server will use.

Typically you will connect to a server supporting WebSockets using a software library which supports it, using a URI with the protocol prefix `ws://` or `wss://` (for secure connections). A successful connection will return an object representing that connection, on which you can `send()` messages to the server and `recv()` (receive) responses. The exact APIs vary, but they typically follow this design.

Websockets example

For example, the Python websockets package provides a simple WebSocket client interface, which can be used roughly like:

```
import websockets
websocket = websockets.connect('ws://example.com/websocket')
# send a greeting to the server
websocket.send('Hello')
# receive and print the response from the server
print(websocket.recv())
# close the connection
websocket.close()
```

In fact, the `websockets` package uses `asyncio` so the real usage requires `await` on all these calls. So let's try a real-world example using both a server and a client. The `websockets` package also includes a simple WebSockets server.

The easiest way to run these examples is probably to open two terminals side-by-side, one for the server and one for the client. We will create a simple echo server in which everything we say to the server will be echoed back to the client.

First, make sure you have the `websockets` package installed:

```
$ pip install websockets
```

Now the **server** code. To implement the server we define a "handler" `async` function. This function is run every time a client connects to our server and defines how the server communicates to each client over the WebSocket. It takes as its sole argument a `websocket` object which is passed to it when the client connects. It runs a loop until the client disconnects:

```
>>> import websockets
>>> async def handler(websocket):
...     while True:
...         # receive a message from the client
...         message = await websocket.recv()
...         # echo the message back to the client
...         await websocket.send(message)
...
```

To start the server we create a simple wrapper that starts the server, on a given port, and then waits for the server to finish (which should be never unless an error occurs). If you pass `port=0` it will automatically pick a free port on your system:

```
>>> async def run_server(handler, host='localhost', port=0):
...     server = await websockets.server(handler, host=host, port=port)
...     # if port==0 we need to find out what port it's actually
...     # serving on as shown below:
...     port = server.sockets[0].getsockname()[1]
...     print(f'server running on ws://{host}:{port}')
...     await server.wait_closed()
...
```

Finally, start the server like so, optionally providing a port like `port=9090`:

```
>>> import asyncio
>>> asyncio.run(run_server(handler, port=9090))
server running on ws://localhost:9090
```

Next on the **client** side, we can simply `connect()` to the server, send some messages and receive their echoes, and exit:

```
>>> import websockets, asyncio
>>> async def client(uri):
...     websocket = await websockets.connect(uri)
...     async def send_recv(msg):
...         print(f'-> {msg}')
...         await websocket.send(msg)
...         resp = await websocket.recv()
...         print(f'<- {resp}')
...
...     await send_recv("Hello!")
...     await send_recv("Goodbye!")
```

(continues on next page)

(continued from previous page)

```
...     await websocket.close()
...
```

Now run the `client()` function passing it the `port` used for the server, for example:

```
>>> asyncio.run(client('ws://localhost:9090'))
-> Hello!
<- Hello!
-> Goodbye!
<- Goodbye!
```

WebSockets programming for real applications proceed more-or-less in the same fashion, though for complex applications it is necessary to establish a protocol over which the client and server communicate. Typically one side opens with an initial message to which the other side responds. Then they take turns sending messages back and forth, the next message often determined by the contents of the previous message, like any conversation.

2.6.3 JSON-RPC

JSON-RPC is a simple protocol for making remote procedure calls (RPC) using JSON-encoded messages. JSON-RPC is not specific to WebSockets, and can be used over any transport mechanism. Renewal uses JSON-RPC to provide *structure* to the WebSocket communications between the Renewal backend and your recsystem.

With JSON-RPC there is a "server" side which provides a number of functions or "methods" which are executed by the server, and which may produce a result. And there is a "client" side which makes remote procedure calls of the methods provided by the server.

JSON-RPC has two types of methods that a server can implement: "requests" are methods that return a result to the client, whereas "notifications" are just for the client to send some notification to the server, and they do not return a response.

Say, for example, our JSON-RPC server implements a `square(x)` method which can be called via RPC:

```
def square(x):
    return x * x
```

Then in order to call this method, a client will send a message to the server like:

```
{"jsonrpc": "2.0", "method": "square", "params": [4], "id": 3}
```

The server will execute `square(4)` and upon completion return the following result to the server:

```
{"jsonrpc": "2.0", "result": 16, "id": 3}
```

Each request and response come with a unique "id" which allows responses to be matched up with the corresponding request (this allows the client to send many requests to the server, which does not necessarily have to respond to requests in the same order it received them).

While JSON-RPC is relatively easy to implement by hand, there are libraries that help converting function calls to correctly-formatted JSON-RPC requests and responses. For example, the *renewal_recsystem* package uses the jsonrpcserver package for Python to implement the base recsystem, and the Renewal backend uses its sister package jsonrpcclient to make RPC calls.

JSON-RPC example

Here's an example of how JSON-RPC can be used over WebSockets, building on our previous example from the *WebSockets primer*.

In this case the WebSocket *client* will act as the JSON-RPC *server* (it provides the functions to run), and the WebSocket *server* will act as the JSON-RPC *client* (it will make the RPC calls). This may seem counter-intuitive but in fact models how communication between the Renewal backend and recsystems works.

As in the *WebSockets primer*, these examples are easiest to run in two separate terminals side-by-side. One for the WebSocket server (JSON-RPC client) side, and one for the WebSocket client (JSON-RPC server) side.

First make sure you have the `websockets` package installed, as well as the JSON-RPC client for `websockets` and the `jsonrpcserver` package:

```
$ pip install websockets jsonrpcclient[websockets] jsonrpcserver
```

WebSocket server side

As before, we must create a handler function which describes what the WebSocket server will do when a client connects to it. In this case it will simply greet the client by sending the `greeting()` notification RPC, and then it will request the square of 42 by calling the `square()` RPC and print the result, then close the connection:

```
>>> import websockets
>>> from jsonrpcclient.clients.websockets_client import WebSocketsClient
>>> async def handler(websocket):
...     # create a WebSocketsClient wrapping the websocket connection
...     rpc_client = WebSocketsClient(websocket)
...
...     # use the notify() method by passing it the name of the
...     # notification RPC and any arguments it takes
...     await rpc_client.notify("greeting", "Hello, friend!")
...
...     # use the request() method the same way, but it returns a
...     # a response object
...     response = await rpc_client.request("square", 42)
...
...     # print the result of the call
...     print(f"got square(42) = {response.data.result}")
...
```

Start the WebSockets server as before (e.g. on port 9090):

```
>>> async def run_server(handler, host='localhost', port=0):
...     server = await websockets.server(handler, host=host, port=port)
...     # if port==0 we need to find out what port it's actually
...     # serving on as shown below:
...     port = server.sockets[0].getsockname()[1]
...     print(f'server running on ws://{host}:{port}')
...     await server.wait_closed()
...
>>> import asyncio
>>> asyncio.run(run_server(handler, port=9090))
server running on ws://localhost:9090
```

WebSocket client side

The WebSocket client acts as a JSON-RPC server: It provides a few methods that can be called via RPC. When it connects to the server, in this case, the server will immediately call those methods and then close the connection (in the case of the actual Renewal backend it keeps the connection open indefinitely and continues to send notifications and requests to your recsystem as long as both ends are running).

The `jsonrpcserver` package provides a `@method` decorator that we can put on top of the definition of any function that we want to be callable via RPC. In this case we define `greeting()` and `square()`. Note in this case we are using the "async dispatcher", so all functions must be defined with `async` even if they don't use `await`:

```
>>> from jsonrpcserver import method
>>> @method
... async def greeting(message):
...     print(f'Received a greeting from the client: {message}')
...
>>> @method
... async def square(x):
...     result = x * x
...     print(f'Squaring {x} for the client -> {result}')
...     return result
...
```

Now define a function to connect to the WebSockets server. It waits to receive RPC calls from the server, and uses the `dispatch()` function which handles the RPC calls by passing them to the appropriate function from the ones we registered above:

```
>>> import websockets, asyncio
>>> from jsonrpcserver import async_dispatch as dispatch
>>> async def client(uri):
...     websocket = await websockets.connect(uri)
...     while True:
...         try:
...             message = await websocket.recv()
...         except websockets.ConnectionClosedOK:
...             # We will receive this exception when trying to receive
...             # more messages from the WebSocket after the server
...             # has closed the connection; so we just exit the loop
...             break
...         response = await dispatch(message)
...         # If response.wanted is False, the message contained a
...         # notification call, in which case we do
...         # not send a response to the other side.
...         if response.wanted:
...             await websocket.send(str(response))
...
```

Now run the client function. On the client side you should see the output as follows:

```
>>> asyncio.run(client('ws://localhost:9090'))
Received a greeting from the client: Hello, friend!
Squaring 42 for the client -> 1764
```

While on the WebSocket server side you should see:

```
got square(42) = 1764
```

The above examples demonstrate in simplified form how your recsystem will communicate with the Renewal backend. In fact it does not require much more than that, though in practical application it can get a little more complicated; see the source code for `renewal_recsystem.server` for example. Its extra complexity arises from the fact that it can handle *multiple simultaneous* RPC calls. In the example above our RPC server just takes once RPC at a time and sends a result in serial. Whereas the implementation in `renewal_recsystem.server` allows it to handle many RPC calls simultaneously and send their results as the RPC handler functions complete.

All you have to do is implement the functions described in *JSON-RPC API* and the rest of the framework will take care of registering them as RPC methods.

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

r

HTTP ROUTING TABLE

/v1

INDEX