# PRISONER Documentation

*Release 1.1.1*

Luke Hutton

Jun 18, 2017

# Contents

PRISONER is a framework for running ethical and reproducible social network experiments.

PRISONER is actively under development, and has been released to help steer its design, and to improve the consideration of these concerns in the community.

# CHAPTER 1

# Features

- A single consistent API to collect and publish data from supported social network sites
- Built-in support for Facebook, Twitter, and Last.fm with simple interfaces to add support for additional services
- Simplified API for sensitively storing data collected from social network sites along with responses to experimental interventions, with support for any database engine with SQLAlchemy bindings
- Declarative syntax for expressing the data collection requirements of an experiment to ensure only the data needed for an experiment can be collected.
- Built-in support for common sanitisations of sensitive data which can be invoked declaratively
- Includes tools to simplify the creation of Docker containers wrapping experiments along with an instance of PRISONER to support reproducibility of experiments

# CHAPTER 2

# Currently in development

The following features are not yet ready for distribution, but will be available in future releases. Please track progress on GitHub if you are interested in contributing to these features.

- Automatic generation of consent forms based on the data-handling requirements of an experiment
- Improve the longevity of code by automatically mapping older API calls to newer API versions, gracefully degrading where individual calls can no longer be satisfied.
- Support the archiving of social network data and PRISONER workflows by generating metadata designed for ingest by research information systems.

CHAPTER 3

# What PRISONER is not

PRISONER is not:

- a crawler. PRISONER is designed to support the execution of user studies which handle social network data, and is not designed for crawling or scraping data from these services where there is no direct intervention from a participant. As a rule of thumb, if your experiment would not require individual participants to authenticate your experiment to access data via their social network account, it is probably not the kind of experiment PRISONER can support.
- a tool for "anonymising" social network data. Guaranteeing the anonymity of identifiable data while maintaining utility is not a trivial problem. PRISONER's support for sanitisations is to coarsen sensitive data as they are collected, and is not intended for anonymising data before release.

If you have any issues deploying or using PRISONER, or have suggestions for how to improve the framework, please raise an issue on GitHub. We would be delighted if you would like to contribute code or improved documentation to PRISONER, and we will accept pull requests with test coverage.

This documentation includes tutorials to help you run a PRISONER instance, build experiments which use social network data, and to package your experiments such that others can reproduce them. A full API reference is available, but familiarity with this is not required to use PRISONER.

Contents:

## Tutorials

These tutorials help you install PRISONER, build a simple experiment, and package your experiment so it can be reproduced by others. We strongly recommend that you follow the first three tutorials.

### Running the PRISONER demo

This tutorial helps you launch a Docker container with a self-contained instance of PRISONER and a demo web application which demonstrates how to use its basic functionality, connecting to Facebook and Twitter.

### Prerequisites

The PRISONER demo uses data from Facebook and Twitter. To run this demo, you will need to have Facebook and Twitter accounts to test the experiment with, be a registered Facebook and Twitter developer, and create an app for both services which PRISONER can use to make authenticated requests to the appropriate API.

### Facebook app

Please follow these steps to create a Facebook app:

- Visit https://developers.facebook.com and follow the steps to register as a developer, if you have not done so before.
- From the navigation bar, click "My Apps > Add a new app > Website".
- Provide a name for your experiment, such as "PRISONER Demo". The name you choose here is not significant.
- Click "Skip quick start", then go to the "Settings" page. Enter "localhost" as the Site URL and App Domains.
- At the top of the screen, note the App ID and App Secret for this app. You will need to provide these to PRISONER later.

### Twitter app

Please follow these steps to create a Twitter app:

- Visit https://dev.twitter.com and click "Manage Your Apps" in the footer.
- Click "Create new app" and provide the required details.
- The "Callback URL" most be a non-empty value. As PRISONER dynamically provides

a callback, the callback given here is irrelevant. For example, you can supplyyour homepage or http://prisoner.cs.st-andrews.ac.uk * Click "Create your Twitter application". * Go to the "Keys and access tokens" tab and make a note of the API key and secret, which you will need later.

### Docker

You will need to be running Docker to run the demo container. The Docker site provides guides to getting started for your platform.

### Start the Docker container

If you are running macOS, you should run the following commands to correctly map the ports to the VirtualBox VM (where "default" is the name of your Docker VM). If you are running Docker natively on Linux, you do not need to do this:

```
VBoxManage controlvm default natpf1 "prisoner,tcp,127.0.0.1,5000,,5000"
VBoxManage controlvm default natpf1 "demo,tcp,127.0.0.1,9000,,9000"
```

To avoid port conflicts after using the Docker container, you should run the following when you're done:

```
VBoxManage controlvm default natpf1 delete prisoner
VBoxManage controlvm default natpf1 delete demo
```

From the command line, run the following to download the Docker image for the PRISONER demo and start the container:

```
docker run -i -t -p 9000:9000 -p 5000:5000 --name prisoner-demo lhutton/prisoner-demo
```

If your /etc/resolv.conf points to 127.0.0.1 (default on Ubuntu installs since 12.04) Docker will try to use public DNS to resolve domains. In some configurations, this might not work (if you receive "Name or service not known" errors when trying to use this experiment, this is probably the cause), in which case you will need to manually provide a nameserver by running the container as follows:

```
docker run -i -t -p 9000:9000 -p 5000:5000 --dns=[YOUR_NAMESERVER_HERE] --name␣
↪prisoner-demo lhutton/prisoner-demo
```

### Running the demo

When the container starts, you will be prompted to enter the Facebook and Twitter App IDs and secrets you noted earlier. Then, you will be given a URL to visit to start testing the experiment.

This demo initially shows the workflow of a trivial experiment which collects some data from your Facebook profile, and displays it in the browser. You can run this experiment to make sure that the PRISONER instance is working. You will see how PRISONER provides the bootstrapping interface to the experiment, showing some basic information about how the experiment works, and the process of authenticating with Facebook.

To view and edit the underlying files, you will need to open a shell on the Docker container:

```
docker exec -it prisoner-demo /bin/bash
```

You can look at how the demo is implemented by visiting /usr/bin/prisoner-demo. "demo.py" implements the server for the web experiment, and shows how the PRISONER session is instantiated, and how Facebook data are collected and displayed. In "static/policy/design.xml" you can see the privacy policy which constrains this experiment. If you are not familiar with the role of the policy, consider reviewing the "Writing your first experiment" tutorial. This container includes vim for editing text files (TODO: include a simpler text editor too)

### Modifying the demo

We can see how trivial modifications to the policy affect the execution of the experiment. For example, when you tried this experiment, you will have seen that your name was displayed, but not your politics and religion, even if you have provided this in your Facebook profile. In demo.py we make a request to the PRISONER API for a "Person" object in our on_get() method, which retrieves a user's biographical attributes, so why are these missing? If we turn to policy.xml, we can see why. Note that in the policy element, we enumerate the gender, first name, and last name attributes, which we have "retrieve" policies for. This provides a whitelist of the data we can collect, so let's add the following religion and politics clauses after the "last name" attribute policy:

```xml
<attribute type="religion">
<attribute-policy allow="retrieve" />
</attribute>

<attribute type="politicalViews">
<attribute-policy allow="retrieve" />
</attribute>
```

If you now revisit the website for the demo experiment, and continue through the PRISONER bootstrap process, you will note that PRISONER automatically detects the changes to the policy and requests the appropriate additional Facebook permissions. Now, the missing attributes will be visible on the experimental results page.

Similarly, you can modify any other aspect of this demo to see how you can request different types of data. To understand the data you can collect from Facebook using PRISONER, consult the documentation for the Facebook Service Gateway.

So far, we have shown we can collect different types of data from Facebook. Now, let's change the experiment completely to collect data from Twitter instead. This might sound like an arduous task, but we can do this by changing a single line of code. Return to /usr/bin/prisoner-demo/demo.py and find line 28, which currently indicates Facebook is our social network of choice. Change this to read "Twitter" and save the file. Return to the URL for the experiment and run through it one more time. Note that PRISONER now authenticates you with Twitter instead, and instead of seeing Facebook's status updates, you see a list of your recent tweets. How is this possible? PRISONER provides a consistent API for requesting equivalent types of data from different services. Therefore, just by changing the name of the provider, we can collect data from a completely different service, while maintaining all other parameters of the experiment.

If you return to the policy.xml we've edited already, you might notice we don't even have a policy for Twitter. While we have explicit Facebook policies to collect attributes such as "gender" or "likes" which are Facebook-specific, we have "base" policies which only refer to the common attributes in all base social objects. Instead of matching the author on the Facebook session ID, we use a special object, "session:Service.id" which allows us to authenticate with whatever the current data provider is, allowing us to re-use a policy for any service, including ones which don't exist yet. Only if we required Twitter-specific attributes would we need to write an explicit Twitter policy.

## Saving data

When running an experiment, we usually want to save some data, which might take the form of some data we collected from a social network site, coupled with data provided by a participant, such as questionnaire responses. PRISONER provides a mechanism for saving data that works similarly to retrieving data from services. It ensures we can only store the data that we absolutely need for our experiment, and can help us apply any sanitisations to remove unnecessarily sensitive data before they are stored, while maintaining as association with additional data provided by participants during the course of an experiment.

We can test this by clicking the "Store this user profile" button, which will save the user profile object we summarise at the top of the screen to the database which PRISONER initialised when we started the experiment.

However, when we click this, we get an error. Why? Just like retrieving data, our policy needs to enable storing social objects on a per-object, and per-attribute basis. Let's quickly amend our policy.xml file to let us save the name attributes of our user object, but not religion and politics. Within both the firstName and lastName elements, where we already have a "retrieve" attribute-policy, add the following:

```
<attribute-policy allow="store" />
```

Then, after the "retrieve" object-policy, add the following:

```
 <object-policy allow="store">
 <object-criteria>
  <attribute-match match="author.id" on_object="session:Facebook.id" />
 </object-criteria>
</object-policy>
```

What did this do? The "store" object-policy tells PRISONER we can now store objects of the type Facebook:Person, so long as it matches the current participant, while the two "store" attribute-policies only allow us to store these attributes.

Let's reload the experiment, and try to save the object again. This time, you should be told this was successful. But what can we do with these data? Let's go back to our shell on the Docker container and run the following:

```
sqlite3
.open /tmp/prisoner_demo.db
SELECT * from response;
```

Here you will see a JSON representation of the Person object we just saved. Note that the attributes, such as religion and gender, have been nullified, while the name is still visible. From here, we can run our own analyses on these results, or share the SQLite database with others.

### Packaging the modified demo

Now that we've made these changes, perhaps we want to package up the changes we've made, including our now-populated database, so others can reproduce our version of the experiment or run analyses with our results. Docker allows us to commit the changes we've made within a container and build a new image from that, which we can use to restore the state of this container at any time, or share with others. To do this, run the following:

```
docker commit prisoner-demo [YOUR_NAME]/prisoner-demo-mod
```

Now, if you run:

```
docker images
```

You will see prisoner-demo-mod among your cached images. From here, you could publish this to Docker Hub to make it publicly visible:

```
docker push [YOUR_NAME]/prisoner-demo-mod
```

Then, anyone else can pull and run your image, or you can simply run this container later as above, by running:

```
docker run -i -t -p 9000:9000 -p 5000:5000 --name prisoner-demo-mod lhutton/prisoner-
↪demo-mod
```

## Installing PRISONER

This tutorial helps you get up and running with a PRISONER instance.

### Installing PRISONER for local development

For developing experiments, you will probably want to run a PRISONER server locally on your development machine to quickly iterate. There are two ways to do this: using our pre-prepared Docker VM image, or cloning the latest release from GitHub.

#### Spin-up a Docker VM

If you have familiarity with the Docker environment, using the PRISONER Docker container is probably the easiest way to get up and running without having to worry about your environment and resolving dependencies. We have an image in Docker Hub which includes the latest release from our GitHub repository.

This guide assumes you have installed Docker and are familiar with using it.

To spin-up a PRISONER instance, run the following at the command line:

```
docker run -p 5000:5000 --name prisoner lhutton/prisoner
```

This will pull the prisoner image from the DockerHub registry, and its prerequisites, which may take several minutes then start an instance of the container.

Now, PRISONER's development server has started on port 5000. Test that everything is working, and that Docker has correctly mapped the port by visiting localhost:<mapped_port>, which should display a "Welcome to PRISONER" message. Depending on your Docker configuration, you may have to access the underlying VM via an alternative IP.

### Clone from GitHub

#### Prerequisites

PRISONER should work on any platform which supports Python 2.7. PRISONER is not compatible with Python 3.

#### Installing PRISONER

PRISONER is developed openly, with all active development pushed to GitHub. We recommend cloning the latest release rather than pulling from head for stability. From the directory where you cloned the repository, run the following at the command line to install any dependencies:

```
pip install -r requirements.txt
```

We strongly recommend running PRISONER from within a virtualenv to isolate dependencies and avoid conflicts with your system Python configuration. See this guide for more information about setting up virtual environments.

From the PRISONER directory, run the following to start the local development server:

```
python server/prisoner.wsgi
```

Make sure everything is working by visiting localhost:5000, where you should see a "Welcome to PRISONER" message.

In the next tutorial, we cover writing your first PRISONER experiment.

## Writing your first experiment

Please note, this tutorial is a work-in-progress and not complete. In the meantime, we recommend reviewing the *PRISONER demo* for an understanding of how to build a simple experiment.

Now that you have a PRISONER development server up and running, we are going to write a simple experiment which collects some data from a participant's Facebook account, sanitises it, and displays it in the browser.

### Prerequisites

This tutorial shows an experiment being written in Python, but as this is to only demonstrate how to use the PRISONER web service, this can be easily adapted to any other environment. This guide assumes a working understanding of XML files.

This example requires a Facebook account to test, and assumes you are registered as a Facebook developer. You will need to create a Facebook app, and make a note of its App ID and Secret. A short guide to doing this is available in our *demo tutorial*.

### In this tutorial

PRISONER experiments consist of three elements, which we will introduce and develop during this tutorial:

- Your experimental application
- An XML privacy policy
- An XML experimental design

### Privacy policies

Privacy policies are XML documents which outline the types of data your experiment needs to collect or publish to social network sites. They contain rules that place constraints on how your experiment handles data. For a given type of data (for example, a tweet, or a Facebook user profile), the policy answers the following questions:

- Is my experiment able to handle this data type?
- Can my experiment retrieve, store, or publish this data type, or a combination of these?
- Under which conditions can I retrieve, store, or publish this data type?
- Which attributes of this data type can my experiment retrieve, store, or publish?
- Which attributes need to be sanitised as they are retrieved, stored, or published?

Encoding this information in a policy file yields some advantages from both ethical and reproducibility perspectives:

- Policies can be written "offline" before you write any code. This allows you to iterate on the appropriate data-handling strategy for your experiment, including engagement with IRB or ethics boards, until you arrive at a final set of constraints for your experiment.
- PRISONER enforces this policy at runtime, so that if the experimental code you are writing attempts to violate its constraints, you cannot inadvertently collect more data than needed for your experiment.
- The standardised representation of the policy allows other documents to be automatically and consistently generated, such as consent forms for participants which reflect the actual data-handling practices of a study, or human-readable summaries of the study's design for review by IRB or ethics boards.
- Privacy policies are effectively a workflow standard for social network experiments, and allow the protocols for studies to be shared. While ideally coupled with the underlying experimental code to support full reproducibility of experiments, the platform-agnostic nature of the privacy policy allows other researchers to replicate a study under the same constraints, even if they are not using PRISONER.

### Writing the privacy policy

Outside of the PRISONER directory, create a directory to store your experiment application. In there, create a new file called policy.xml. Populate it with the following:

```xml
<?xml version="1.0" encoding="UTF-8"?>

<p:privacy-policy
        xmlns:p="http://prisoner.cs.st-andrews.ac.uk/prisoner/privacy-policy"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://prisoner.cs.st-andrews.ac.uk/prisoner/privacy-
↪policy privacy_policy.xsd">

        <policy for="Facebook:Person">
                <attributes>
                        <attribute type="id">
```

```
                                <attribute-policy allow="retrieve"></attribute-policy>
                        </attribute>
                        <attribute type="displayName">
                                <attribute-policy allow="retrieve"></attribute-policy>
                        </attribute>
                        <attribute type="username">
                                <attribute-policy allow="retrieve"></attribute-policy>
                        </attribute>
                        <attribute type="image">
                                <attribute-policy allow="retrieve"></attribute-policy>
                        </attribute>
                        <attribute type="firstName">
                                <attribute-policy allow="retrieve"></attribute-policy>
                        </attribute>
                        <attribute type="middleName">
                                <attribute-policy allow="retrieve"></attribute-policy>
                        </attribute>
                        <attribute type="lastName">
                                <attribute-policy allow="retrieve"></attribute-policy>
                        </attribute>
                        <attribute type="birthday">
                                <attribute-policy allow="retrieve"></attribute-policy>
                        </attribute>
                        <attribute type="gender">
                                <attribute-policy allow="retrieve"></attribute-policy>
                        </attribute>
                </attributes>

                <object-policy allow="retrieve">
                        <object-criteria>
                                        <attribute-match match="id" on_object=
↪"session:Facebook.id" />
                        </object-criteria>
                </object-policy>
        </policy>
</p:privacy-policy>
```

So, what does this policy do? Simply, it enumerates the objects we can collect, and the attributes of those objects we can collect. The policy file includes a clear hierarchy where we define policy elements for each data type, which contains a collection of attributes we can process, and an object-policy which describes the criteria under which we can collect objects of this type. In this experiment, we want to collect some biographical information about the participant in this experiment, so our policy is for the User object provided by Facebook. The Facebook prefix defines the namespace, which means we are explicitly requesting Facebook's representation of a User, and is not generalisable to the other social network sites that PRISONER supports. This means we can access Facebook-specific attributes, but at the cost of making our experiment harder to adapt for other services. Because we only want to collect data about the current participant, we provide an object-policy which dictates that we can only collect a User object if it matches the ID of the participant. This ensures our experiment can not inadvertently collect more sensitive data than we need, such as the identitfy of the participant's friends. Although we now have criteria for collecting the objects themselves, the objects PRISONER returns will have no attributes. Therefore, we must specifically enumerate the attributes we need in the attributes collection of this policy. Each policy element enables us to retrieve that attribute. We could add additional attribute-policy elements for each attribute to also enable us to store those attributes if we later wish to persist these data, but this is not necessary for this experiment.

Later, when we write the experimental application, we will provide PRISONER with this policy to initialise the experiment and allow these constraints to be enforced.

### Experimental designs

Experimental design files provide PRISONER with basic metadata about your experiment, such as its name, properties for specific services such as API keys, and the structure of any data you wish to store so PRISONER can manage the database appropriately.

### Writing the experimental design

In the same directory where you wrote your privacy policy, add another file called design.xml, and populate it with the following:

```
<p:experimental-design xmlns:p="http://prisoner.cs.st-andrews.ac.uk/prisoner/
↪experimental_design"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://
↪prisoner.cs.st-andrews.ac.uk/prisoner/experimental_design.xsd">

        <experiment name="PRISONER Tutorial">
                <tables>
                        <table name="participant" type="participant">
                                <column name="email" type="string"/>
                        </table>

                        <table name="response" type="response">
                                <column name="participant_id" type="string" />
                                <column name="user" mapTo="Facebook:Person" />
                        </table>
                </tables>
                <props>
                        <prop for="Facebook" key="app_id" value="$APP_ID" />
                        <prop for="Facebook" key="app_secret" value="$APP_SECRET" />
                        <prop for="Facebook" key="api_version" value="2.0" />
                        <prop for="PRISONER" key="secret" value=
↪"prisonerTutorialSecret" />
                </props>
        </experiment>
</p:experimental-design>
```

What does this design do? First of all, note that in the tables element, we specify two tables. The first is marked as a “participant” table, which indicates to PRISONER that the table stores metadata about individual participants. This allows PRISONER’s internal record of an individual participant, including their service-specific session identifiers, to be related to the metadata that is specific to your experiment. In this case, we identify participants by their email address.

Our second table is marked as a “response” table, which lets PRISONER know that data collected during the course of an experiment can be stored here. For the purposes of this tutorial, we will store the participant’s Facebook profile along with our identifier for that participant.

The schema we have provided here does not directly translate to the underlying database which PRISONER will instantiate on our behalf. The “mapTo” syntax in our response table means PRISONER will store the representation of a Facebook profile in a metatable, which individual response records will be related to. Rather than directly accessing the database, PRISONER recommends you use its persistence API to store responses and retrieve them, with fully-formed social objects returned as part of the response, where appropriate.

Finally, we provide some properties, or “props”, which provide miscellaneous metadata PRISONER needs to provide your experiment. Because we are using Facebook, we must provide the App ID and secret for our app which we noted earlier, so make sure you edit the file with these values as appropriate. Finally, we provide a PRISONER secret. This is a passphrase which you will provide to PRISONER whenever you make administrative commands, such as initialising an experiment, to make sure you are authorised to do this.

#### Writing the experiment app

We're two thirds of the way there! These two policy files do a lot of the work in telling PRISONER what your experiment needs to do. From these two files, PRISONER can create an entire pre-briefing structure and authentication flow, and is now able to enforce constraints on our experiment's data collection.

Let's put this into action by writing a small web app. In this example, we're using Python and the WSGI library Werkzeug, but you can choose to follow along in an alternative environment of your choosing. As PRISONER is exposed as a web service, your application just consists of your experiment-specific logic, along with some simple HTTP requests to PRISONER to initialise an experiment and retrieve the appropriate data.

Our experimental application consists of four functions: * Initialise the PRISONER experiment with the policy files we have prepared, and allow PRISONER to authenticate our participant with Facebook

- Collect some data from the participant's Facebook profile
- Present the collected data to the participant
- Store a response from the participant along with a sanitised subset of the

collected data in the PRISONER database.

## Deploying a PRISONER server

This guide will explain how to deploy a PRISONER server.

## Best practices for distributing reproducible PRISONER experiments

PRISONER aims to help make social network studies more reproducible, but what does that mean in practice, and what do you need to do to make your experiments reproducible?

First of all, when we talk about reproducibility, we mean *someone else can reproduce the methodology of your experiment*. This is distinctive from *replicating* or *recomputing* a result, where you might want to verify that an algorithm produces a certain result given a certain input. This distinction is important, because it means we need to make sure that others have access to all of the resources needed to reproduce your experiment. This probably doesn't just mean the source code for your experimental software, or the scripts you used to perform stats, but all manner of details, including how participants were recruited, what types of data were collected about them, and how they were briefed before taking part in the study.

We suggest that reproducibility hinges on three components: the availability of **code**, **methodology**, and **data**. In this guide we discuss how you can work towards adequate sharing of your code and methodology. While making available the source code for your PRISONER-based experiments achieves the former, it may only make a limited contribution to the latter. Sharing your PRISONER policy files, however, can be very helpful, as it encodes useful information about how you collect and process data in your experiment, which can aid others reproducing your experiments, even if they choose not to use PRISONER themselves.

Placing your code in an online archive or public version control repository, via GitHub for example, is a good way of letting others examine and run your code. This approach, however, has some limitations. Distributing software in this way does not make it easy to resolve package dependencies, and if others are not running the same operating system as you, or other environmental variables differ, it may be difficult or impossible to execute your experiment.

Specifically, if you are developing a PRISONER experiment, you will need to distribute your PRISONER policies, and someone hoping to execute your experiment needs to be able to understand how to setup and run an instance of PRISONER to get things working.

In this guide we cover some best practices for distributing PRISONER experiments, outlining a few scenarios which involve packaging your experiment in different ways. Please note this guidance is not final and may not cover all scenarios. We welcome suggestions or improvements as GitHub issues or pull requests.

### 1) Sharing code and PRISONER policies

Making the source code for your experiment available online is a great first step to allowing others to see how your experiment works, and reproduce it. The Software Sustainability Institute provides lots of guidance on this subject.

We recommend using GitHub, as you can assign a DOI to individual releases of your repository using Zenodo, which can make it easier for others to cite your experiment software.

Within your repository, you should include the two PRISONER policy files your experiment requires: the privacy policy and experimental design. While this will not be sufficient for others to execute the experiment without access to an instance of PRISONER itself, these policy files are an excellent way of distributing a list of constraints about the data-handling requirements of your experiment, which may not be obvious from the source code alone.

Before distributing your experiment, please make sure your experimental design file does not include any hard-coded API keys. In your documentation, make sure you point out that users must supply their own API keys as necessary.

In the documentation for your project, linking to the PRISONER website lets others find out more about the framework, so they can choose whether to download it themselves in order to execute the experiment.

Finally, we ask you to let us know that you're sharing a PRISONER experiment, which you can do by email or Twitter. It's useful for us to be able to monitor how widely used our tools are, and we can also give your experiment a shout-out to help people find it!

### 2) Forking PRISONER on GitHub

While the above is clearly better than nothing, it falls a little short of our reproducibility goal as it falls on anyone wishing to replicate your study to manage their own instance of PRISONER.

One way to simplify this is to distribute your experiment as a fork of PRISONER on GitHub, with your experiment-specific code added to the repository. This has a number of advantages:

- Your experiment is clearly bound to a specific release of PRISONER avoiding issues with mismatched versions
- If you have made any modifications to PRISONER, their relationship to the canonical version is easier to track, and you can also push changes upstream to help make PRISONER better!
- The visibility of your experiment will be helped by its direct relationship to the base PRISONER repository, and we can help promote interesting uses of PRISONER to achieve better impact.
- It helps us monitor usage of PRISONER with no additional effort on your part.
- As your repository includes a full release of PRISONER, people don't need to go to any further effort to get your experiment running.

### 3) Release a virtual machine image

If your experiment has complex software or environmental dependencies which can impede distribution, you may wish to consider a virtual machine image, either as a full VM (recomputation.org has guidance on this) or as a Docker image.

We recommend using Docker as you can distribute a relatively lightweight image of your experimental code and PRISONER policies, while expressing any other environmental dependencies. Anyone else running Docker can then pull your image and instantiate a container with an executable version of your experiment and PRISONER server.

A guide to using Docker is beyond the scope of this document, but to help you get started, we provide PRISONER itself, and a separate working example, as Docker images via Docker Hub. This tutorial explains how to run our example Docker experiment. To see how we build this Docker image, derived from a base PRISONER image see the source for the demo.

We recommend distributing both a PRISONER fork as above, and a Docker image (either via Docker Hub or a private Docker registry). This approach has some further advantages:

- Maximises the sustainability of your experiment, as most environmental dependencies have been abstracted from the user.
- Consistency for the end-user. While each GitHub repository may have its own dependencies and installation procedures, once someone has learned how to pull and run one Docker image, they can run any experiment in the same manner.

### Final thoughts

In this guide, we've introduced a few ways you can distribute your PRISONER experiments, with trade-offs between upfront complexity and the ease with which others can reproduce your experiment. The scenarios we discuss here are based on our own experience in conducting and distributing experiments, and should not be considered the final word. Ultimately, you should choose whichever workflow suits you, and please share your own recommendations with us and the community via GitHub. We will update this document with alternative distribution strategies which emerge.

## prisoner package

### Subpackages

#### prisoner.gateway package

#### Subpackages

##### prisoner.gateway.tests package

##### Submodules

##### prisoner.gateway.tests.FacebookGatewayTests module

**class** `prisoner.gateway.tests.FacebookGatewayTests.`**`BaseFacebookGatewayTestCase`**(*methodName='runTes*

Bases: `unittest2.case.TestCase`

**`create_user_all_permissions`**()

Creates a test user who has every relevant FB permission enabled

**`create_user_no_permissions`**()

Creates a test user who has no FB permission enabled

**`get_bad_processor`**()

**`get_empty_processor`**()

**`get_good_processor`**()

**`get_good_props`**()

**post_graph_data**(*query*, *params*)
Internal Function. Post the params dictionary to the given query path on the Graph API Use for creating, deleting, updating content All calls must be authenticated

**Parameters**

- **query** (*str*) – Graph API query to perform
- **params** (*dict*) – Dictionary of data to publish to this endpoint

**setUp**()
### These values must be set before running these tests!

**set_test_user_attributes**()
sets some attributes for testing common to all test objects

**class** prisoner.gateway.tests.FacebookGatewayTests.**GetPermissionsForPolicyTestCase**(*methodName='*
Bases: *prisoner.gateway.tests.FacebookGatewayTests.BaseFacebookGatewayTestCase*

**test_bad_policy**()

**test_good_policy**()

**class** prisoner.gateway.tests.FacebookGatewayTests.**InitialiseTestCase**(*methodName='runTest'*)
Bases: *prisoner.gateway.tests.FacebookGatewayTests.BaseFacebookGatewayTestCase*

Test handling of init parameters: tokens, props, and policies

**test_bad_policy**()

**test_bad_props**()

**test_bad_token**()

**test_good_policy**()

**test_good_props**()

**test_good_token**()

**test_no_policy**()

**test_no_props**()

**test_no_token**()

**class** prisoner.gateway.tests.FacebookGatewayTests.**StatusTestCase**(*methodName='runTest'*)
Bases: *prisoner.gateway.tests.FacebookGatewayTests.BaseFacebookGatewayTestCase*

**test_post**()

**class** prisoner.gateway.tests.FacebookGatewayTests.**UserTestCase**(*methodName='runTest'*)
Bases: *prisoner.gateway.tests.FacebookGatewayTests.BaseFacebookGatewayTestCase*

**test_good_get**()

**test_post**(**arg*, ***kw*)

### prisoner.gateway.tests.LastfmGatewayTests module

**class** prisoner.gateway.tests.LastfmGatewayTests.**BaseLastfmServiceGatewayTestCase**(*methodName='ru*
Bases: unittest2.case.TestCase

**setUp**()

class prisoner.gateway.tests.LastfmGatewayTests.**ImageTestCase**(*methodName='runTest'*)
Bases: *prisoner.gateway.tests.LastfmGatewayTests.BaseLastfmServiceGatewayTestCase*

**test_get_failure**()

**test_get_success**()

### prisoner.gateway.tests.SocialGatewayTests module

### Module contents

## Submodules

### prisoner.gateway.FacebookGateway module

class prisoner.gateway.FacebookGateway.**Album**
Bases: *prisoner.SocialObjects.SocialObject*

Representation of an album object on Facebook. Albums are created by users or apps and have a number of key attributes such as privacy and count. Albums also have a cover photo and a type. Once you have an album's ID, you can then use Photo() to retreive the photos it contains.

**albumType**
The album's type. (Eg: Wall, Mobile)

**comments**
The comments on this photo album.

**count**
The number of photos in this album.

**coverPhoto**
This album's cover photo.

**likes**
The people who've liked this album.

**photos**
The images in the album.

**privacy**
The privacy setting for this album.

class prisoner.gateway.FacebookGateway.**Albums**
Bases: *prisoner.SocialObjects.Collection*

Lightweight collection class for representing collections of albums.

class prisoner.gateway.FacebookGateway.**Book**
Bases: *prisoner.gateway.FacebookGateway.Page*

Stub for representing music.

class prisoner.gateway.FacebookGateway.**Checkin**
Bases: *prisoner.SocialObjects.SocialObject*

Representation of a check-in. A Facebook user can be determined to have been somewhere if they explicitly said they were there in a status, or have been tagged in a photo that is also tagged with that location. As well as containing basic information such as where the check-in is for and who the user was with, a check-in object also contains a "Type" attribute that specifies how the check-in was determined. (Eg: Status, Photo...)

**`checkinType`**
This check-in's type. (Eg: Status, Photo)

**`image`**

**class** `prisoner.gateway.FacebookGateway.`**`Checkins`**
Bases: *`prisoner.SocialObjects.Collection`*

Lightweight collection class for representing collections of check-ins.

**class** `prisoner.gateway.FacebookGateway.`**`Comment`**
Bases: *`prisoner.SocialObjects.Note`*

Representation of a comment object on Facebook. Comments are typically short replies / notes on objects such as statuses, photos, check-ins or just about any other Facebook object. Comments consist of their content, an author a published date and a permalink.

**class** `prisoner.gateway.FacebookGateway.`**`Comments`**
Bases: *`prisoner.SocialObjects.Collection`*

Lightweight collection class for representing collections of comments.

**class** `prisoner.gateway.FacebookGateway.`**`FacebookServiceGateway`**(*access_token=None*, *props={}*, *policy=None*)
Bases: *`prisoner.gateway.ServiceGateway.ServiceGateway`*

Service gateway for Facebook. This gateway interacts with Facebook directly by making calls via the network's Social Graph API.

The Facebook Service Gateway allows you to access Facebook from PRISONER experiments. In order to use Facebook, you must register an app with the Facebook Developers portal and provide three additional props in your experimental design file. The app_id and app_secret props correspond to the values for your app, and the api_version prop dictates which version of the Facebook API your experiment targets. At this time, only "2.0" is an acceptable API version. See the documentation on key concepts for guidance on using props in experimental designs.

**`Album`**(*operation*, *payload*)
Performs operations on a user's photo albums. Currently only supports GET operations. This lets us retrieve a list of photo albums associated with the supplied payload ID.

**Parameters**

- **`operation`** (*`str`*) – The operation to perform. (GET)
- **`payload`** (`Person`) – A Person() whose ID is either a Facebook UID or username.

**Returns** A collection representing this person / object's photo albums.

**`Book`**(*operation*, *payload*)
Performs operations relating to people's taste in books and literature. Currently only supports GET operations. This lets us get the books / authors people are into.

**Parameters**

- **`operation`** (*`str`*) – The operation to perform. (GET)
- **`payload`** (`SocialObject`) – A Person() whose ID is either a Facebook UID or username.

**Returns** A list of the books this person likes.

**`Checkin`**(*operation*, *payload*)
Performs operations on check-ins / objects with location. Currently only supports GET operations. This lets us retrieve a list of places the supplied User() or Person() has been.

**Parameters**

- **`operation`** (`str`) – The operation to perform. (GET)
- **`payload`** (`SocialObject`) – The Person() object to retrieve check-in information for.

**Returns** A collection of objects representing check-ins.

**`Friends`**(*operation*, *payload*)
Performs operations on a user's friends. Only supports GET operations. This lets us retrieve someone's entire friends list.

**Parameters**

- **`operation`** (`str`) – The operation to perform. (GET)
- **`payload`** (`Person`) – A Person() whose ID is either a Facebook UID or username.

**Returns** A collection representing this person's friends list.

**`Like`**(*operation*, *payload*)
Returns a user's liked pages. Only supports GET operations.

**Parameters**

- **`operation`** (`str`) – The operation to perform. (GET)
- **`payload`** (`SocialObject`) – A Person() whose ID is either a Facebook UID or username.

**Returns** A list of pages this person likes.

**`Movie`**(*operation*, *payload*)
Performs operations relating to people's taste in films. Currently only supports GET operations. This lets us retrieve the movies / films people like.

**Parameters**

- **`operation`** (`str`) – The operation to perform. (GET)
- **`payload`** (`SocialObject`) – A Person() whose ID is either a Facebook UID or username.

**Returns** A list of the movies this person likes.

**`Music`**(*operation*, *payload*)
Performs operations relating to people's musical tastes. Currently only supports GET operations, so we can just get the bands a person / user likes.

**Parameters**

- **`operation`** (`str`) – The operation to perform. (GET)
- **`payload`** (`SocialObject`) – A Person() whose ID is either a Facebook UID or username.

**Returns** A list of the bands this person likes.

**`Note`**(*operation*, *payload*)
Performs operations on a user's status updates. Currently only supports GET operations. This lets us retrieve a user's entire backlog of status updates.

**Parameters**

- **operation** (*str*) – The operation to perform. (GET)
- **payload** (SocialObject) – A Person() whose ID is either a Facebook UID or username.

**Returns** A collection representing this person's backlog of status updates.

**Person**(*operation*, *payload*)
Performs operations relating to people's profile information. Currently only supports GET operations. This allows us to, given a suitable payload such as a Person() object, retrieve the information they have added to Facebook. (Eg: Full name, education, religion...)

**Parameters**

- **operation** (*str*) – The operation to perform. (GET)
- **payload** (SocialObject) – A Person() object whose ID is either a Facebook UID or username.

**Returns** A Person() object with all available attributes populated.

**Photo**(*operation*, *payload*)
Performs operations on images. Currently only supports GET operations. This lets us retrieve the photos associated with the supplied payload's ID. This will commonly be an Album() to get the photos in said album, or a User() / Person() to get any photos they're tagged in.

**Parameters**

- **operation** (*str*) – The operation to perform. (GET)
- **payload** (SocialObject) – The Facebook object to retrieve associated photos for.

**Returns** A collection representing photos associated with the supplied object.

**Session**()
The Facebook session exposes the authenticated user as an instance of User(). Can also be accessed in the same way as Person() as this class simply extends it.

**Returns** session object

**complete_authentication**(*request*)
Completes authentication. Extracts the "code" param that Facebook provided and exchanges it for an access token so we can make authenticated calls on behalf of the user.

**Parameters** **request** (*HTTPRequest*) – Response from the first stage of authentication.

**Returns** Unique access token that should persist for this user.

**generate_permissions_list**()
Generates a list of permissions based on the experiment's privacy policy.

**Returns** List of permissions

**get_comments**(*object_id*)
Internal function. Takes a JSON Facebook object and returns a list of the comments on it.

**Parameters** **facebook_obj** (*Dict*) – The Facebook object to get comments on.

**Returns** A list representing the comments on this object.

**get_graph_data**(*query*)
Internal function. Queries Facebook's Graph API and returns the result as a dict.

**Parameters** **query** (*str*) – The Graph API query to perform. (Eg: /me/picture?access_token=...)

**Returns** A Dict containing the parsed JSON response from Facebook. Attributes are accessed through their name.

**get_likes** (*object_id*)
Internal function. Takes a JSON Facebook object and returns a list of the people who've liked it.

**Parameters** **facebook_obj** (*Dict*) – The Facebook object to get likes for.

**Returns** A list representing the people / users that have liked this object.

**get_value** (*haystack*, *needle*)
Internal function. Attempts to get the value corresponding to the supplied key. If no key exists, None is returned.

**Parameters**

- **haystack** – The Dictionary object to look at.
- **needle** – The key we're looking for.

**Returns** If the key exists, its corresponding value is returned. Otherwise None is returned.

**parse_comments** (*facebook_obj*)
Internal function. Takes a JSON Facebook object and returns a list of the comments on it. Note that this function just PARSES. It does not attempt to retrieve all the comments on the given object. This means it has a limit of around 25 comments.

**Parameters** **facebook_obj** (*Dict*) – The Facebook object to get comments on.

**Returns** A list representing the comments on this object.

**parse_json** (*json_obj*)
Internal function. Takes a JSON object as returned by Facebook and returns the Dict representation of it. Avoids having to call json.loads(?) everywhere, and allows for potential improvements in the future.

**Parameters** **json_obj** (*str, list*) – The JSON object to parse.

**Returns** A Dict object representing the supplied JSON.

**parse_likes** (*facebook_obj*)
Internal function. Takes a JSON Facebook object and returns a list of the people who've liked it. Note that this function just PARSES. It does not attempt to retrieve all the likes for the given object. This means it has a limit of around 25 likes.

**Parameters** **facebook_obj** (*Dict*) – The Facebook object to get likes for.

**Returns** A list representing the people / users that have liked this object.

**parse_location** (*facebook_obj*)
Internal function. Takes a JSON Facebook object and returns a Place object representing its location.

**Parameters** **facebook_obj** (*Dict*) – The Facebook object to get the location of.

**Returns** A Place() object representing the location of the supplied object.

**parse_tags** (*facebook_obj*)
Internal function. Takes a JSON Facebook object and returns a list of the objects that have been tagged in it. (Usually people)

**Parameters** **facebook_obj** (*Dict*) – The Facebook object to get tags for.

**Returns** A list representing the people / objects that were tagged in the supplied object.

**post_graph_data**(*query*, *params*)
Internal Function. Post the params dictionary to the given query path on the Graph API Use for creating, deleting, updating content All calls must be authenticated

**Parameters**

- **query** (*str*) – Graph API query to perform
- **params** (*dict*) – Dictionary of data to publish to this endpoint

**request_authentication**(*callback*)
Initiates Facebook's authentication process. Returns a URI at which the user can confirm access to their profile by the application.

**Parameters** **callback** – PRISONER's authentication flow URL. User must be redirected here after registering with Facebook

in order to continue the flow. :type callback: str :returns: URI the user must visit in order to authenticate.

**request_handler**(*request*, *operation*, *payload*, *extra_args=None*)
Wrapper around object requests. Used to inject any necessary debug headers.

**Parameters**

- **request** (*method*) – A method instance on this service gateway
- **operation** (*str*) – A HTTP method of this request (ie. GET or POST)
- **payload** – The criteria for this request, ie. which objects to retrieve,

or data to publish :param extra_args: A dictionary of arguments to further filter this query (eg. limit) :type extra_args: dict :returns: A WrappedResponse with any additional headers injected

**restore_authentication**(*access_token*)
Provides a mechanism to restore a session. (Essentially refresh an access token) Facebook does not allow access tokens to be refreshed. However, if the user is forced to go through the authentication process again, it will be done transparently so long as the PRISONER app has not requested additional permissions.

**Parameters** **access_token** (*str*) – The current access token held for this user.

**Returns** False, thus forcing the authentication process to take place again. (Transparently)

**str_to_time**(*time*)
Internal function. Used to convert Facebook's ISO-8601 date/time into a Date/Time object. Also converts Facebook's MM/DD/YYYY format used for birthdays.

**Parameters** **time** (*str*) – The string to parse.

**Returns** A Date/Time object.

**class** prisoner.gateway.FacebookGateway.**Friends**
Bases: *prisoner.SocialObjects.Collection*

Lightweight collection class for representing collections of users / friends.

**class** prisoner.gateway.FacebookGateway.**Image**
Bases: *prisoner.SocialObjects.Image*

Representation of a photo object on Facebook. Photos are images uploaded by users or applications. As well as the standard attributes inherited from SocialObject, a photo also has additional specialised attributes such as position, width and height. A photo also contains Image() objects to represent both the full-size image as well as thumbnails.

**comments**
The comments on this photo.

**height**
The height of this photo. (Pixels)

**image**
The full size version of this photo.

**likes**
The people who've liked this photo.

**position**
Position of this photo in its album.

**tags**
The people who are tagged in this photo.

**thumbnail**
The thumbnail image for this photo.

**width**
The width of this photo. (Pixels)

**class** `prisoner.gateway.FacebookGateway.`**`Images`**
Bases: *`prisoner.SocialObjects.Collection`*

Lightweight collection class for representing collections of photos.

**class** `prisoner.gateway.FacebookGateway.`**`Like`**
Bases: *`prisoner.gateway.FacebookGateway.Page`*

A Like is just a representation of a Page

**class** `prisoner.gateway.FacebookGateway.`**`Likes`**
Bases: *`prisoner.SocialObjects.Collection`*

Lightweight collection class for representing collections of likes.

**class** `prisoner.gateway.FacebookGateway.`**`Movie`**
Bases: *`prisoner.gateway.FacebookGateway.Page`*

Stub for representing music.

**class** `prisoner.gateway.FacebookGateway.`**`Music`**
Bases: *`prisoner.gateway.FacebookGateway.Page`*

Stub for representing music.

**class** `prisoner.gateway.FacebookGateway.`**`Note`**
Bases: *`prisoner.SocialObjects.Note`*

Representation of a status object on Facebook. Status updates are short posts by Facebook users. They can either be entirely textual or contain a link or a photo. As well as the basic attributes, status updates also contain a privacy setting as well as a collection of likes and comments.

**comments**
The comments on this status update.

**likes**
The people who liked this status update.

**link**
A link to an external resource embedded in this status update

**privacy**
The privacy setting for this status update. (Eg: Friends)

**class** `prisoner.gateway.FacebookGateway.`**`Page`**

Bases: *`prisoner.SocialObjects.SocialObject`*

Representation of a generic Facebook page / object. Pages are used to represent entities like bands, books, films and so on.

**`category`**

**`image`**

**class** `prisoner.gateway.FacebookGateway.`**`Person`**

Bases: *`prisoner.SocialObjects.Person`*

Representation of a user object on Facebook. Users are essentially the backbone of the Facebook service and such objects can contain a great deal of information. User objects will not always have values for all their attributes, as Facebook does not require users to provide allthis information.

**`bio`**
This person's short biography.

**`birthday`**
This person's birthday.

**`education`**
This person's education history.

**`email`**
This person's email address.

**`firstName`**
This person's first name.

**`gender`**
This person's gender.

**`hometown`**
This person's hometown.

**`interestedIn`**
This person's sexual orientation.

**`languages`**
Languages this person can speak.

**`lastName`**
This person's last name.

**`location`**
This person's current location.

**`middleName`**
This person's middle name.

**`politicalViews`**
This person's political preferences.

**`relationshipStatus`**
This person's relationship status.

**`religion`**
This person's religion.

**`significantOther`**
This person's significant other.

**timezone**
This person's timezone. (Offset from UTC)

**updatedTime**
When this person last updated their Facebook profile.

**username**
This person's Facebook username.

**work**
This person's work history.

**class** prisoner.gateway.FacebookGateway.**StatusList**
Bases: *prisoner.SocialObjects.Collection*

Lightweight collection class for representing collections of statuses.

**class** prisoner.gateway.FacebookGateway.**Tags**
Bases: *prisoner.SocialObjects.Collection*

Lightweight collection class for representing collections of tags. Tags are simply User() objects that have been tagged in a photo or status.

prisoner.gateway.FacebookGateway.**check_none**(*value*)
Internal function. Used to check to see whether or not a value is None. If so, it replaces it with N/A. Mainly used for testing and creating string representations.

## prisoner.gateway.LastfmGateway module

**class** prisoner.gateway.LastfmGateway.**LastfmServiceGateway**(*access_token=None*)
Bases: *prisoner.gateway.ServiceGateway.ServiceGateway*

ServiceGateway for Last.fm. This is a concrete implementation to demonstrate how to build experimental applications which consume data from, and publish data to, Last.fm

This ServiceGateway supports a number of core Social Objects, and introduces a range of its own to represent site-specific constructs such as Tracks and Playlists, etc.

This gateway uses a modified version of pylast to interact with Last.fm API

**Comment**(*operation*, *payload*)
Performs operations on Comment objects. Supports GET and POST operations.

**Parameters**

- **operation** (*str*) – The operation to perform (GET, POST)
- **payload** (SocialObject) – Provide a Comment object. Will be posted as a shout to the profile of the inReplyTo attribute.

**Image**(*operation*, *payload*)
Performs operations on Image objects. Only supports GET operations.

**Parameters**

- **operation** (*str*) – The operation to perform (GET)
- **payload** (SocialObject) – Provide a Person object, to return that user's profile image

:returns Image – image of requested object

**Playlist**(*operation*, *payload*)

**Session**()
The Last.fm session exposes the authenticated user as a Person instance

**Track**(*operation*, *payload*)
Performs operations on Track objects. Only supports the GET operation (you can get a user's tracks, you can't create them).

Returns a set of Tracks depending on the payload.

**Parameters**

- **operation** (*str*) – The operation to perform (GET)
- **payload** (SocialObject) – Provide a Person (whose id is username) to return a set of that user's Loved Tracks

**Returns** list[track] - set of tracks matching criteria

**complete_authentication**(*request*)
Completes authentication. Request passed via authentication flow must contain a token argument as returned by Last.fm. We pass this to Last.fm to return a session key (lasts indefinitely) for making authenticated calls on this user.

**Parameters** **request** (*HTTPRequest*) – Request from first stage of authentication

**Returns** Session key to persist for this user

**request_authentication**(*callback*)
Instigates first of Last.fm's two-stage authentication. Returns a URL for the participant to confirm access to their profile by this application.

**Parameters** **callback** (*str*) – PRISONER's authentication flow URL. The participant must go here after authenticating with Last.fm to continue the flow

**restore_authentication**(*access_token*)
Restores previously authenticated session. Last.fm session keys last indefinitely so this just provides pylast with the old session key and hope it works

**Parameters** **access_token** (*str*) – Last.fm session key received from previous auth attempt

**Returns** boolean - was auth successful?

**class** prisoner.gateway.LastfmGateway.**Playlist**
Bases: *prisoner.SocialObjects.Collection*

**class** prisoner.gateway.LastfmGateway.**Track**
Bases: *prisoner.SocialObjects.SocialObject*

**artist**
String identifying artist of this track.

**get_friendly_name**(*attribute*)

**tag**
Set of tags associated with this track

**title**
The title of this track.

**transform_artist**(*transformation*, *level*)
Applies anonymising transformation to the artist attribute. Uses the base_transform_name transformation

### prisoner.gateway.ServiceGateway module

**class** `prisoner.gateway.ServiceGateway.`**`SARHeaders`**(*operation*, *provider*, *object_type*, *payload*)

Bases: `object`

SARHeaders contain information about the request for a SocialObject. They are used within the validation/sanitisation process as part of a SocialActivityResponse. They may also be used to audit the requests made for objects.

**`object_type`**

The name of the SocialObject type to use. This must be a core SocialObject or provided by the ServiceGateway indicated in the provider attribute.

**`operation`**

The operation to be performed by this request. These map to HTTP methods (GET, POST etc.)

**`payload`**

The criteria for a request, or object to publish.

**`provider`**

The provider this request is intended for. Must map to a ServiceGateway

**`wrapped_headers`**

The header component of a WrappedResponse. Allows service-specific headers to be surfaced

**class** `prisoner.gateway.ServiceGateway.`**`ServiceGateway`**(*props={}*, *policy=None*)

Bases: `object`

Service Gateways make external providers of social data accessible to the rest of PRISONER. They accept sanitised requests for Social Objects, and make the appropriate API calls to return well-formed social objects. They also accept requests to publish social objects to services, converting these to the representations expected by that API.

This is an abstract interface - concrete implementations subclass this and provide methods corresponding to each Social Object they implement, with the signature def ObjectType(self, operation, payload).

See examples of concrete implementations for examples of this.

ServiceGateways must adhere to the following conventions:

- exist as a package <GatewayName>ServiceGateway.py within the gateway module
- contain a class called <GatewayName>ServiceGateway which subclasses ServiceGateway

NEW in 0.2: ServiceGateways should expect a props dict in __init__, populated by an experiment's design policy, and a policy parameter with an instance of PolicyProcessor, allowing the experiment's privacy policy to be interrogated.

NEW in 0.2: All gateways must implement a request_handler() - or use the superclass-provided implementation. This allows a dictionary of response headers to be added by the service gateway, provided to a WrappedResponse object.

**`Image`**(*operation*, *payload*)

Perform operations on ServiceGateway to publish and retrieve Image objects

**Parameters**

- **`operation`** (*`str`*) – The operation to perform (eg. GET, POST)
- **`payload`** (`SocialObject`) – Object to perform operation with, eg. GET objects matching criteria, or POST this object to service

**Session**()

Each ServiceGateway can maintain a Session object, which contains limited information that is needed to persist throughout the session with the service. This should *not* be used as a way of caching social objects to circumvent the usual GetObject interface. The session is intended to store, for example, metadata about the authenticated participant so that it is possible to relate the participant to their service username, etc.

**complete_authentication**(*request*)

Second stage of authentication. Request contains the response from the client-side authentication, which should contain access tokens required to complete the authentication process.

Do not implement this method if the service does not perform authenticated requests.

**Parameters** **request** (*HTTPRequest*) – Request with clientside authentication tokens

**request_authentication**(*callback=None*)

First stage of two-stage authentication. Participation client has requested that participant is authenticated with this service.

For most authentication schemes, return a URL the participant can visit to authenticate themselves with the service. After completing this stage, the user must be redirected to the callback URL.

Do not implement this method if the service does not perform authenticated requests.

**Parameters** **callback** (*str*) – URL that PRISONER has determined participant must visit after authenticating with service.

**request_handler**(*request*, *operation*, *payload*)

**restore_authentication**(*access_token*)

Similar to complete_authentication(), except directly providing the access token needed by the gateway to restore an existing session. We can't guarantee the token is still valid, so the gateway should return a boolean value to indicate whether the attempt was successful. If not, it may be necessary to complete the clientside request/complete flow.

**Parameters** **access_token** (*object*) – Object needed by service gateway to restore authentication

**Returns** boolean - was authentication attempt succesful?

**class** prisoner.gateway.ServiceGateway.**SocialActivityResponse**(*content*, *headers*)

Bases: object

SocialActivityResponse wraps a SocialObject received from a service gateway. It provides the original object alongside headers relating to the request from the participation clients. These headers are used to validate the request, and sanitise the response object.

**content**

**headers**

**class** prisoner.gateway.ServiceGateway.**WrappedResponse**(*social_object*, *headers*)

Bases: object

A Social Object returned by a service gateway is wrapped in this object which allows the gateway to inject additional metadata to be handled elsewhere

**headers**

**social_object**

### prisoner.gateway.TwitterGateway module

**class** `prisoner.gateway.TwitterGateway.`**`Note`**

Bases: *`prisoner.SocialObjects.Note`*

A tweet is a single post shared to Twitter, derived from the base Note object.

**`favorites`**

**`retweets`**

**class** `prisoner.gateway.TwitterGateway.`**`Person`**

Bases: *`prisoner.SocialObjects.Person`*

A Twitter User

**class** `prisoner.gateway.TwitterGateway.`**`Timeline`**

Bases: *`prisoner.SocialObjects.Collection`*

A collection of Tweets

**class** `prisoner.gateway.TwitterGateway.`**`TwitterServiceGateway`**(*policy=None*, *props=None*)

Bases: *`prisoner.gateway.ServiceGateway.ServiceGateway`*

Service Gateway for Twitter.

This gateway supports reading a user's timeline and publishing tweets on their behalf, with support for geo-tagged content.

**`Note`**(*operation*, *payload*)

Requests all tweets by a given user.

**Parameters**

- **`operation`** (*`str`*) – (GET) tweets
- **`payload`** (`Person`) – A Person whose ID is a Twitter ID

**Returns** A list of Tweet objects

**`Person`**(*operation*, *payload*)

Gets the user profile of a user.

**Parameters**

- **`operation`** (*`str`*) – (GET) user
- **`payload`** (`Person`) – A Person or User whose ID is a Twitter user ID

**Returns** User object populated by profile

**`Session`**()

The Twitter session exposes the authenticated user as an instance of Person().

**`complete_authentication`**(*request*)

Final stage of authentication flow.

**Parameters** **`request`** (*`HTTPRequest`*) – Response from the first stage of authentication.

**Returns** Unique access token that should persist for this user.

**`request_authentication`**(*callback*)

Initiates Twitter's authentication process. Returns a URI at which the user can confirm access to their profile by the application.

**Parameters** **`callback`** – PRISONER's authentication flow URL. User must be redirected here after registering with Twitter in order to continue the flow.

**Returns** URI the user must visit in order to authenticate.

**`request_handler`**(*request*, *operation*, *payload*, *extra_args=None*)
Wrapper around object requests. Used to inject any necessary debug headers.

**Parameters**

- **`request`** (`method`) – A method instance on this service gateway
- **`operation`** (`str`) – A HTTP method of this request (ie. GET or POST)
- **`payload`** – The criteria for this request, ie. which objects to retrieve,

or data to publish :param extra_args: A dictionary of arguments to further filter this query (eg. limit) :type extra_args: dict :returns: A WrappedResponse with any additional headers injected

**`restore_authentication`**(*access_token*)
Provides a mechanism to restore a session. (Essentially refresh an access token) Twitter does not allow access tokens to be refreshed. However, if the user is forced to go through the authentication process again, it will be done transparently so long as the PRISONER app has not requested additional permissions.

**Parameters** **`access_token`** (`str`) – The current access token held for this user.

**Returns** False, thus forcing the authentication process to take place again. (Transparently)

### Module contents

## prisoner.persistence package

### Submodules

### prisoner.persistence.PersistenceManager module

**class** `prisoner.persistence.PersistenceManager.`**`PersistenceManager`**(*exp_design=None*, *policy_processor=None*, *connection_string=None*)
Bases: `object`

The PersistenceManager manages the storage of all data, including participant metadata, experimental responses, and persistence of Social Objects. Storage of data is subject to the privacy policy for the experiment. Generally, participation clients do not directly instantiate a PersistenceManager, and where possible, friendlier interfaces are available through the ExperimentBuilder and SocialObjectsGateway.

**`close_connection`**()

**`do_build_schema`**(*drop_first=False*)

**`experimental_design`**

**`get_existing_provider_auth`**(*participant_id*, *provider*)
Checks if the given participant has been previously authenticated with the named provider. If so, returns the credentials stored. Otherwise, returns None, so participant should continue with consent flow. If credentials are found, a ServiceGateway should be provided with them. It should signal whether it is able to successfully authenticate with them. If not, it should attempt to reauthenticate server-side if possible (eg. request new session from API). If clientside intervention is required, it should signal this.

**Parameters**

- **participant_id** (*int*) – ID of participant to authenticate
- **provider** (*str*) – Name of provider to return authentication for

**Returns** Stored credentials or None

**get_participant** (*schema*, *participant_id*)
Retrieve the participant with the given ID from the given schema.

**Parameters**

- **schema** (*str*) – name of scema to get participant from
- **participant_id** (*int*) – ID of participant to return

**Returns** tuple - participant data from database

**get_props** ()
Parses the props collection in the experimental design and makes these available as a dict of dicts

**Returns** dict of dicts of props

**get_table** (*table_type*, *table_name*)
Returns an active connection to the requested table. Used internally for data access. Do not use this from participation clients. Instead, use managed data access interfaces where possible to ensure data are sanitised appropriately.

**Parameters**

- **table_type** (*str*) – Type of table [response, participant, object]
- **table_name** – Name of table to return

**Returns** Table - requested table

**post_response** (*schema*, *response*)
Writes response to given schema.

**Parameters**

- **schema** (*str*) – Schema to write to
- **response** (*dict*) – Response data to write

**post_response_json** (*sog*, *schema*, *response*)
Wrapper to post_response for use by web services.

**Parameters**

- **sog** (SocialObjectsGateway) – Current instance of SocialObjectsGateway
- **schema** (*str*) – Name of response schema to write to
- **response** (*JSON object as str.*) – JSON object corresponding to the response schema. References to SocialObjects must consist of its prisoner_id (as originally received)

**props**

**rebuild_engine** (*connection_string*)

**register_participant** (*schema*, *participant*)
Add the participant data in given dictionary to the participant table in this database and return the ID

**Parameters** **schema** – name of participant table (must be of type 'participant') :type schema: str. :param participant: dictionary of data about participant :type participant: dict :returns: int – inserted row ID

**register_participant_with_provider**(*participant_id*, *provider*, *token*)

Store access credentials with this provider for the given participant. Access token can be any object, so long as the relevant service gateway is able to interpet it.

**Parameters**

- **participant_id** (`int`) – Participant to register access token with
- **provider** (`str`) – Name of provider to register access token with
- **token** (`object`) – Access token used in authenticated calls

**Returns** row as inserted in meta_table

**validate_design**(*design*)

Tests that the given experimental design validates against the XML schema

**Parameters** **design** (`str`) – Path to experimental design

**Raises** IOError

**Returns** ElementTree - parsed experimental design object

### Module contents

## prisoner.server package

### Submodules

### prisoner.server.webservice module

**class** `prisoner.server.webservice.`**PRISONER**

Bases: `object`

PRISONER Web Service Exposes the functionality of PRISONER through a RESTful API. Participation clients should use this API to manage social objects.

The PRISONER web service requires the following flow:

1) Call / to handshake with PRISONER. Returns a PRISession header whose value must be passed to all future requests as a PRISession argument

2.**Call /begin with the following POST payload:** 'policy': URL to your experiment's privacy policy 'design': URL to your experimental design 'participant': The ID of the current participant 'providers': A comma-seperated list of services the participant must be authenticated with

3) PRISONER will return a URL your participant must visit to complete their consent and authentication flow. Call this and append an (escaped) argument "callback" - this is the URL your participant should be returned to, to begin using your experiment

4.**From this point onwards, use PRISONER to request objects:** /get/<provider>/<object_name>/<payload>/<criteria>

eg. to get a participant's favourite tracks by Pixies on Last.fm we query: /get/Lastfm/Track/session:Lastfm.id/x.artist=="Pixies"

(for readability we have not escaped this query string - this must be safely encoded before making requests!)

Append a '?async' parameter to perform this request asynchronously. Call the same URL, but with a '?isready' parameter to get the result (if it's not ready yet, expect blank response)

**To publish objects:** /publish/<provider>/<object_name> with a form-encoded payload of the data to publish.

eg. to publish a comment to my own Last.fm profile, we query: /publish/Lastfm/Comment {'author': session:Lastfm.id, 'inReplyTo': session:Lastfm.id, 'content': "Test comment" }

**To store experimental responses:** /post with a form-encoded payload matching the response schema in your experimental design.

eg. to publish a response to a question about a favourite track, we query: /response {'track': 5343gt32-g43519500-223f, 'answer': "My response", }

5) PRISONER provides a simple session layer for *temporarily* storing state information (eg. one set of responses by a participant in a multi-step form). To write to the session store call:

/session/write/ with a form-encoded 'key' and 'data' (any arbitrary data can be stored)

**Later, to retrieve session data, call:** /session/read/<key>

Note that the session store is *not* persistent, and there are no guarantees how long this data will be accessible. For permanent data, use the experiment response interface.

Note that the PRISONER Web Service returns JSON objects corresponding to instances of Social Objects. Each object in a JSON response includes a "prisoner_id" attribute. Use this to subsequently relate a request to a previous object you received. PRISONER will lookup the original object based on this identifier. For example, in our experimentntal response above, we provided a track ID. This allows requests to be lightweight while PRISONER temporarily stores the complete version of that object.

**`dispatch_request`**(*request*)
Internal handler to get from URL mapping to the right response handler :param request: Current HTTP request

**`find_nth`**(*haystack*, *needle*, *n*)
Utility method for fallback endpoint.

**Parameters**

- **`haystack`** – search for nth item in here
- **`needle`** – search nth this in haystack
- **`n`** – this is n!

**Returns** found needle

**`get_builder_reference`**(*request*)
Each session has its own instance of PRISONER's internals, keyed on the session cookie.

**Parameters** **`request`** – Current HTTP request

**Returns** The ExperimentBuilder for this session

**`on_begin`**(*request*)
Initialises the flow of an experiment. This endpoint must be provided with the following arguments:

policy: the URL to the privacy policy XML file

design: the URL to the experimental design XML file title: the name of the experiment

contact: the email address of the researcher

db: a connection string (must have SQLAlchemy bindings to be supported) for PRISONER to store data to participant: form data to register the current participant

providers: a comma-delimited list of all social network sites this experiment connects to

callback: which URL for your experiment to redirect the participant to after successful authentication

**Returns** Response for participant to be redirected to

`on_cancel`(*request*)
Call if the participant does not provide consent and revokes participation. Should also invalidate and remove any session identifiers.

**Parameters** **`request`** – Current HTTP request

`on_complete`(*request*)
Called at the end of the authentication flow. Redirects participant to the callback provided at the start of the experiment. Request must provide cbprovider and PRISession arguments to identify this session and provider flow.

**Parameters** **`request`** – The current HTTP request

**Returns** redirect to experiment callback or /cancel if participant

invalidates entry

`on_confirm`(*request*)
Provides the authentication flow to redirect participant through authentication for each requested provider. Request must provide the following arguments:

pctoken: the authentication token provided earlier in the flow provider: the name of the service participant is being authenticated against PRISession: current session identifier from cookie

**Parameters** **`request`** – HTTP request with above arguments provided

**Returns** Redirect response to service authentication or to complete

authentication flow

`on_consent`(*request*)

`on_fallback`(*request*, *wildcard*)
If an invalid URL is provided, try to rewrite and redirect it in case something malformed it.

**Parameters**

- **`request`** – current HTTP request
- **`wildcard`** – Not used

**Returns** redirect to rewritten URL

`on_get_object`(*request*, *provider*, *object_name*, *payload*, *criteria=None*)
Returns a SocialObject of given type (object_name) from a given provider. The payload is the primary criteria for evaluating a request for the object, and must be interpretable by the receiving ServiceGateway. For example, providing a user ID may return instances of objects created by that user. Provide a lambda expression (criteria) to filter this request further (eg. only return objects matching a certain attribute value).

For larger requests, an asynchronous request pattern is also provided (for AJAX calls). Make your request as usual, but append the argument 'async'. This will immediately return if your request was valid. Periodically, call your request URL again, instead with the additional argument 'isready'. This will return an empty response if the request has not been completed, or the full response object when it is.

**Parameters** **`request`** – Current HTTP request. If a limit argument is provided this

will be pushed to an extra_args dictionary for filtering in gateways. Provide an async parameter to perform request asynchronously, or an isready parameter to check if a previous async request for the same data is ready. :param provider: The service to retrieve data from :type provider: str :param object_name: The class name of object being retrieved :param payload: Query argument of object to be retrieved, ie. object ID :param criteria: Lambda function for filtering objects before being returned :returns: A JSON response of the returned object, or an empty JSON object if request is happening asynchronously, or existing async request is not ready

**`on_handshake`**(*request*)
This initial call provides the client with their session token. If response is good, call /begin providing the given PRISession value.

**Parameters** **`request`** – Current HTTP response

**Returns** HTTP response to confirm handshake.

**`on_invalidate`**(*request*, *session*)
Invalidate the current session, removing it from memory. Call this at the end of the experiment to remove its footprint, or in the event of an irrecoverable error, from which you do not want the participant to recover without restarting the experiment flow

**Parameters**

- **`request`** – Current HTTP request
- **`session`** (`str`) – Session ID to invalidate

**`on_post_response`**(*request*)
Writes response data to the given response schema. Provide a form with: schema: the name of the schema to write to response: json data of response to write

**Returns** HTTP Response with the written data

**`on_publish_object`**(*request*, *provider*, *object_name*)
Publishes the given data as a social object to the given service.

**Parameters**

- **`request`** – HTTP request with the required payload as a HTTP form
- **`provider`** (`str`) – the name of the service to publish to
- **`object_name`** (`str`) – The class of object being published

**`on_register`**(*request*)
Register a participant. Requires a URL for the experimental design and privacy policy, and a form of columns to insert about this participant.

**Parameters** **`request`** – Current HTTP request

**Returns** HTTP response to confirm the ID of the registered participant

**`on_schema`**(*request*)
Builds the database schema matching this experimental design.

**Parameters** **`request`** – Current HTTP request

**Returns** Response to confirm the schema was built successfully.

**`on_session_read`**(*request*, *session*)
Read the session data corresponding to the given key parameter. Session data is bound to the active PRISession.

**Parameters**

- **`request`** – Current HTTP request with a key argument of which data to read
- **`session`** – Session we're reading data from

**Returns** HTTP response with JSON object of returned data

**`on_session_timeout`**(*request*)

Participant is redirected here if their session key is not found

**Parameters** **`request`** – Current HTTP request

**Returns** Response rendering the expired session template

**`on_session_write`**(*request*, *session*)

Writes arbitrary data to a temporary session. A session is bound to a PRISession, and is intended to retain state data during an experiment before committing to database. There is no guarantee how long the session will be valid for, so gracefully handle instances where expected data cannot be retrieved.

To write session data, provide a POST form with a "key" value (used to retrieve the data later) and "data" (the arbitrary session data to store).

**Parameters**

- **`request`** – Current HTTP request including session data to write
- **`session`** – The session to write data to

**Returns** Empty response if successful

**`render_template`**(*template_name*, ***context*)

Return the given template populated with content :param template_name: Name of the template file to render :type template_name: str :param context: Additional context :returns: Response

**`set_builder_reference`**(*request*, *builder*)

Attach this ExperimentBuilder to the current session.

**Parameters**

- **`request`** – Current HTTP request
- **`builder`** (`ExperimentBuilder`) – Instance of ExperimentBuilder to Attach

**Returns** ExperimentBuilder

**`threaded_get_object`**(*request*, *provider*, *object_name*, *payload*, *criteria=None*, *extra_args=None*)

Wrapper around the SocialObjectGateway GetObjectJSON method to retrieve social objects as JSON, then return as response. This should not be called directly, but is intended to be called by the on_get_object handler.

**Parameters**

- **`request`** – Current HTTP request
- **`provider`** (`str`) – The service to retreive data from
- **`object_name`** (`str`) – The class of object being retrieved
- **`payload`** – The criteria to retrieve objects by
- **`criteria`** – Optional lambda function to filter request by
- **`extra_args`** – Dictionary of generic arguments to filter on. Currently

only limit is (partially) supported :type extra_args: dict :returns: Response with a JSON object of requested data

**wsgi_app**(*environ*, *start_response*)
Exposes the server as a WSGI application. Handles session injection and request dispatch.

":param environ: The environment for this request :param start_response: Initial response for app :returns: Response

prisoner.server.webservice.**create_app**()
Instantiates server instance.

### Module contents

### prisoner.tests package

#### Submodules

#### prisoner.tests.tests module

PRISONER Unit Test Runner

If it runs, test it.

Don't worry too much about where your tests live, so long as they live. By convention, each module contains a tests directory - segregate your tests logically as you see fit, and complement the base test suite with your own tests for additional service gateways, etc.

**class** prisoner.tests.tests.**SocialObjectsTestCase**(*methodName='runTest'*)
Bases: unittest.case.TestCase

**setUp**()

#### Module contents

### prisoner.workflow package

#### Subpackages

#### prisoner.workflow.tests package

#### Submodules

#### prisoner.workflow.tests.PolicyProcessorTests module

**class** prisoner.workflow.tests.PolicyProcessorTests.**BasePolicyProcessorTestCase**(*methodName='runT*
Bases: unittest2.case.TestCase

**get_bad_processor**()

**get_disallow_processor**()

**get_good_processor**()

**setUp**()

**class** `prisoner.workflow.tests.PolicyProcessorTests.`**`InferAttributesTestCase`**`(`*methodName='runTest'*`)`
Bases: *`prisoner.workflow.tests.PolicyProcessorTests.BasePolicyProcessorTestCase`*

**`test_bad_attribute`**`()`

**`test_bad_format`**`()`

**`test_bad_nested_obj`**`()`

**`test_bad_obj`**`()`

**`test_good_nested_obj`**`()`

**`test_good_obj`**`()`

**class** `prisoner.workflow.tests.PolicyProcessorTests.`**`InferObjectTestCase`**`(`*methodName='runTest'*`)`
Bases: *`prisoner.workflow.tests.PolicyProcessorTests.BasePolicyProcessorTestCase`*

**`test_good_literal`**`()`

**`test_invalid_base`**`()`

**`test_invalid_literal`**`()`

**`test_invalid_social_gateway`**`()`

**`test_missing_base`**`()`

**`test_valid_base`**`()`

**`test_valid_social_gateway`**`()`

**class** `prisoner.workflow.tests.PolicyProcessorTests.`**`SanitiseObjectRequestTestCase`**`(`*methodName='ru*
Bases: *`prisoner.workflow.tests.PolicyProcessorTests.BasePolicyProcessorTestCase`*

**`test_good_response`**`()`

**`test_logic_failOnAnd`**`()`

**`test_logic_failOnImplicitAnd`**`()`

**`test_logic_failOnNested`**`()`

**`test_logic_failOnOr`**`()`

**`test_malformed_headers`**`()`

**`test_malformed_response`**`()`

**`test_missing_headers`**`()`

**`test_no_allow_attribute`**`()`

**class** `prisoner.workflow.tests.PolicyProcessorTests.`**`ValidateObjectRequestTestCase`**`(`*methodName='ru*
Bases: *`prisoner.workflow.tests.PolicyProcessorTests.BasePolicyProcessorTestCase`*

**`test_bad_request_badObject`**`()`

**`test_bad_request_badOperation`**`()`

**`test_fail_validation`**`()`

**`test_good_validation`**`()`

**class** `prisoner.workflow.tests.PolicyProcessorTests.`**`ValidatePolicyTestCase`**(*methodName='runTest'*)
Bases: *`prisoner.workflow.tests.PolicyProcessorTests.BasePolicyProcessorTestCase`*

**`test_bad_policy`**()

**`test_good_policy`**()

**`test_no_policy`**()

### prisoner.workflow.tests.SocialObjectGatewayTests module

**class** `prisoner.workflow.tests.SocialObjectGatewayTests.`**`BaseSocialObjectGatewayTestCase`**(*methodN*
Bases: `unittest2.case.TestCase`

This test suite ensures: - caching behaves as expected - service gateway authentication flows are delegated correctly - object requests are delegated correctly

**`GetObject_returns_object`**(**args*, ***keywargs*)

**`ProcessorInferObject_returns_Person`**(**args*, ***keywargs*)

**`setUp`**()

**class** `prisoner.workflow.tests.SocialObjectGatewayTests.`**`CacheObjectTestCase`**(*methodName='runTest'*)
Bases: *`prisoner.workflow.tests.SocialObjectGatewayTests.BaseSocialObjectGatewayTestCase`*

**`test_cache_hit`**()

**`test_cache_miss`**(**arg*, ***kw*)

**class** `prisoner.workflow.tests.SocialObjectGatewayTests.`**`GetObjectJSONTestCase`**(*methodName='runTest*
Bases: *`prisoner.workflow.tests.SocialObjectGatewayTests.BaseSocialObjectGatewayTestCase`*

**`test_bad_get`**(**args*, ***keywargs*)

**`test_good_get`**()

**class** `prisoner.workflow.tests.SocialObjectGatewayTests.`**`ProvidePoliciesTestCase`**(*methodName='runT*
Bases: *`prisoner.workflow.tests.SocialObjectGatewayTests.BaseSocialObjectGatewayTestCase`*

**`test_provide_good_exp_design`**()

**`test_provide_good_privacy_policy`**()

**`test_provide_invalid_exp_design`**()

**`test_provide_invalid_privacy_policy`**(**arg*, ***kw*)

### Module contents

## Submodules

### prisoner.workflow.Exceptions module

**exception** `prisoner.workflow.Exceptions.`**`DisallowedByPrivacyPolicyError`**(*error*)
Bases: `exceptions.Exception`

Raised if a method attempts to perform an action not allowed by the current privacy policy.

**exception** `prisoner.workflow.Exceptions.`**`IncorrectSecretError`**
Bases: `exceptions.Exception`

Raised if the secret is not correct for this experiment

**exception** `prisoner.workflow.Exceptions.`**`InvalidPolicyProvidedError`**(*error*)
Bases: `exceptions.Exception`

**exception** `prisoner.workflow.Exceptions.`**`NoPrivacyPolicyProvidedError`**
Bases: `exceptions.Exception`

Raised if a privacy policy is required before the operation can be completed. See the SocialObjectGateway or ExperimentBuilder to provide a privacy policy.

A Privacy policy is required to read or write data to/from service gateways.

**exception** `prisoner.workflow.Exceptions.`**`OperationNotImplementedError`**(*operation*)
Bases: `exceptions.Exception`

Raised if a service gateway does not implement a request operation (GET, POST, PUT etc.)

**exception** `prisoner.workflow.Exceptions.`**`RuntimePrivacyPolicyParserError`**(*error*)
Bases: `exceptions.Exception`

Raised if a privacy policy which passed schema validation fails complex validation - eg. an invalid object reference is provided, or logical criteria is incorrectly expressed

**exception** `prisoner.workflow.Exceptions.`**`ServiceGatewayNotFoundError`**(*gateway*)
Bases: `exceptions.Exception`

Raised if a participation client attempts to connect to a service without a corresponding ServiceGateway class in the gateway module.

**exception** `prisoner.workflow.Exceptions.`**`SocialObjectNotSupportedError`**(*gateway*, *object*)
Bases: `exceptions.Exception`

Raised if a service gateway doesn't know how to handle the given social object.

## prisoner.workflow.ExperimentBuilder module

**class** `prisoner.workflow.ExperimentBuilder.`**`CallbackHandler`**(*application*, *request*, ***kwargs*)
Bases: `tornado.web.RequestHandler`

Takes a parameter (callback), and calls the unescaped version of that URL (useful for baking nested params in a callback URL)

**`get`**()

**class** `prisoner.workflow.ExperimentBuilder.`**`CompleteConsentHandler`**(*application*, *request*, ***kwargs*)
Bases: `tornado.web.RequestHandler`

Called when the user has authenticated themselves with the last provider necessary. This completes the authentication flow and allows the experimental application to begin.

**`get`**()

**class** `prisoner.workflow.ExperimentBuilder.`**`ConsentFlowHandler`**(*application*, *request*, ***kwargs*)

Bases: `tornado.web.RequestHandler`

This renders the human-readable representation of the privacy policy and ensures the participant understands the data requirements of the experimental application before providing consent.

**`get`**()

**class** `prisoner.workflow.ExperimentBuilder.`**`ExperimentBuilder`**

Bases: `object`

The ExperimentBuilder is the interface for bootstrapping an experiment with PRISONER. After instantiating an ExperimentBuilder, complete the following steps:

- call provide_privacy_policy() with the path to your privacy policy XML file
- call provide_experimental_design() with the path to your experimental design XML file
- call authenticate_participant() with the id of the participant in this session
- call authenticate_providers() with a list of services which the participant must authenticate with to participate
- call build() to generate a pre-experiment flow, which allows participants to review a human-readable version of your privacy policy, and to authenticate themselves with providers as needed.

**`authenticate_participant`**(*schema*, *participant_id*)

Provide the ID of the participant in this experiment. This participant must exist in the participant table for this experiment.

**Parameters** **`participant_id`** (*`int`*) – ID of participant

**`authenticate_providers`**(*providers*)

Provide a list of provider names this participant needs to be authenticated with to participate (eg. if they are only using a subset of providers all participants will be using, only include that subset in this list). When the experiment is built, each gateway will inject its own authentication logic.

**Parameters** **`providers`** (*`list[str]`*) – List of providers to authenticate with

**`build`**(*callback_url*)

Using the information provided by the participation client, instigate the experiment consent process. This does the following:

- parse the experimental design and privacy policy and generate a human-readable document, relevant to the participant, which also lists which providers the participant will be asked to authenticate with
- creates a temporary web server - the participation client must access the returned URL using the cookie provided when the ExperimentBuilder was instantiated
- when the user consents to the policies, each service gateway for which authentication is needed provides a URL to authenticate with which the participant is asked to visit in

turn (decorated by additional context from PRISONER for participants' confidence). Note, this URL must contain the entire authentication flow, so you may need to host this yourself, particularly if this involves two (or more) factor authentication as users are bounced between URLs (many authentication flows expect a URL callback). This flow must return a token to persist alongside the Participant.

: param callback_url: A callable to be invoked only when consent is confirmed - ie. the entrypoint for the participation client :type callback_url: callable : returns: URL participant must visit to begin consent flow

**`build_schema`**()

Constructs the database schema (destroying whatever data might already exist). This places the database in a state in which participants may be registered, and experiments run, but does not return usable interfaces to the rest of the workflow (such as the SocialObjectGateway)

**consent_confirmed**(*cookie*)
Called when user with given cookie accepts consent. If cookie is valid, continue the authentication flow for that participant.

**get_props**(*who_for=None*)
Retrieve the props for a given target (eg. PRISONER or a provider)

**Parameters** **who_for** (`str`) – the target to retrieve props for

**provide_contact**(*contact*)
How to contact someone in connection with this experiment, eg. an email address. This should be provided in a form that fits the following sentence construction:

“Contact the researcher at <contact>.”

**Parameters** **contact** (`str`) – Contact information

**provide_db_string**(*db_string*)
Set connection string for this experiment.

**Parameters** **db_string** (`str`) – conncetion string

**provide_experimental_design**(*exp_design*)
Provide the experimental design for this experiment.

**Parameters** **exp_design** (`str`) – Path to experimental design file

**provide_privacy_policy**(*policy*)
Provide the privacy policy for this experiment.

**Parameters** **policy** (`str`) – Path to privacy policy file

**provide_title**(*title*)
The title of the experiment as presented to your participants. :param title: Friendly experiment title :type title: str

**class** `prisoner.workflow.ExperimentBuilder.`**ProviderAuthentHandler**(*application*, *request*, ***kwargs*)

Bases: `tornado.web.RequestHandler`

Called during the authentication flow for each provider. Informs the participant about the service they are about to authenticate themselves with, then redirects to the appropriate URL for that service.

**get**()

## prisoner.workflow.PolicyDocumentGenerator module

**class** `prisoner.workflow.PolicyDocumentGenerator.`**PolicyDocumentGenerator**(*policy*, *design*, *format*)

Bases: `object`

The PolicyDocumentGenerator generates human-readable versions of PRISONER policy documents - the privacy policy and experimental design. It returns natural language documents so that participants and other stakeholders know exactly how an experimental application will collect, store, and generate data about them. This exposes generators for HTML, LaTeX, among others, with the ability to define your own generators for specific output formats.

This is in-development, concept stuff. Do not use in production!

**`do_print_policy`**()

**`html`**()

**`latex`**()

**`print_policy`**(*tree*)

## prisoner.workflow.PolicyProcessor module

**class** `prisoner.workflow.PolicyProcessor.`**`PolicyProcessor`**(*policy=None*, *sog=None*)

Bases: `object`

The Policy Processor is responsible for validating and sanitising all requests to retrieve and publish Social Objects.

It requires a well-formed privacy policy XML file to be supplied. If this is missing or invalid, all requests will fail.

The PolicyProcessor is an internal object. Service gateways and participation clients do not need to directly interact with it. See the SocialObjectGateway for a friendly interface to these innards.

PolicyProcessor needs an instance of SocialObjectGateway so it can evaluate the current session scope of service gateways.

**`privacy_policy`**

Get the privacy policy bound to this PolicyProcessor.

**`validate_policy`**(*policy*)

Validates a privacy policy against the XML Schema.

**Parameters** **`policy`** (`str.`) – Path to privacy policy XML file.

**Returns** ElementTree – policy object

**Raises** IOError

## prisoner.workflow.SocialObjectGateway module

**exception** `prisoner.workflow.SocialObjectGateway.`**`InvalidPrivacyPolicy`**(*error*)

Bases: `exceptions.Exception`

**exception** `prisoner.workflow.SocialObjectGateway.`**`ServiceGatewayNotFound`**(*gateway*)

Bases: `exceptions.Exception`

**class** `prisoner.workflow.SocialObjectGateway.`**`SocialObjectsGateway`**(*server_url=None*)

Bases: `object`

This is a friendlier interface to PRISONER's internals, which participation clients should access. This coordinates access to other service gateways, and the management of experimental responses.

A single instance of this object should be maintained throughout the lifecycle of an experimental application.

**`GetObject`**(*provider*, *object_type*, *payload*, *allow_many=False*, *criteria=None*, *extra_args=None*)

Interface for retrieving an object from a service gateway. Requests are verified against the privacy policy, and returned objects are sanitised as appropriate. The payload and filter arguments are semantically distinct. See the documentation for each argument to understand how to use them.

**Parameters**

- **`provider`** (*`str`*) – name of provider to get object from
- **`object_type`** (*`str`*) – name of object to get
- **`payload`** (*`object`*) – This must contain a SocialObject or dictionary of arguments necessary for the ServiceGateway to make a meaningful request. For example, it may be a user ID to retrieve their photos, however it should not contain criteria for filtering the objects returned (see criteria). The expected payload depends on the ServiceGateway and the objects you are requesting. See the documentation for each object exposed by the ServiceGateway to see the payload it requests.
- **`criteria`** – Optional criteria for filtering the objects returned by the ServiceGateway. This expression is run on all objects returned by gateway, and only where it evaluates as True is the object returned. Uses syntax similar to lambda expressions, without prefix. x is used to refer to each instance of an object. eg. "'party" in x.tags'

**Returns** SocialObject – sanitised for consumption by participation client

**`GetObjectJSON`**(*provider*, *object_type*, *payload*, *criteria*, *extra_args=None*)
Interface for retrieving objects from a service gateway, for consumption by web services.

This differs from GetObject in some fundamental ways. GetObject is more pythonic - you request objects by supplying relevant SocialObjects, and you get SocialObject instances in return. This method however, receives plain-text responses, and returns JSON objects. Whereas GetObject expects a semantically-appropriate SocialObject as the payload (eg. supply an instance of Person to receive objects of a given type owned by that Person), this method expects a payload expressed as a query string, using the namespaced syntax found in the privacy policy spec. For example, a payload of "session:Lastfm.id" will be evaluated as "get objects authored by the user ID in the Last.fm session. "literal:lukeweb", similarly, returns objects owned by that literal user. JSON objects are returned, with the same fields as the Pythonic counterparts. A key difference is that the returned object has an additional attribute injected - prisoner_id. This is a unique identifier for the returned object *that is

valid for the duration of this session*. Rather than passing around full

instances of objects, subsequent queries, or publication of experimental responses, need only refer to this ID to ensure PRISONER is able to relate your requests back to the full representation of the data. Note that subsequent attempts to retrieve the cached object are subject to the privacy policy sanitisation process of the *original* request.

**`PostObject`**(*provider*, *object_type*, *payload*)
Request to write a Social Object to a given provider. Requests are verified against the privacy policy, and outgoing objects are sanitised as necessary.

**Parameters**

- **`provider`** (*`str`*) – Provider name
- **`object_type`** (*`str`*) – Type of object to write
- **`payload`** (*`Social Object`*) – Object to post to provider

**`PostObjectJSON`**(*provider*, *object_type*, *payload*)
Used by web services interface for pushing objects to a service gateway.

Expects a payload as a JSON dictionary, where the keys are the appropriate fields of <object_type> This method converts the dictionary to a native object and pushes it through the PRISONER pipe for sanitisation and publication

**`cache_object`**(*object_to_cache*)
Generates a unique identifier for this object, caches it, and returns the identifier.

**Parameters** **`object_to_cache`** (`SocialObject`) – SocialObject to cache

**Returns** str – object's identifier

**`complete_authentication`**(*provider*, *request=None*)
Completes the second stage of authentication with a provider.

**Parameters**

- **`provider`** (`str.`) – Name of provider to authenticate with.
- **`request`** – The request received from the provider when it

called the PRISONER callback. This should contain any parameters needed to complete authentication :type request: werkzeug Request

**`get_participant`**()

**`get_service_gateway`**(*provider*)
External wrapper to internal function

**`post_response`**(*schema*, *response*)
Passes the response to the PersistenceManager to write to the internal database. There must be an experimental design bound first.

**Parameters**

- **`schema`** (`str.`) – Name of the response table to write to
- **`response`** (`dict`) – The response dictionary to write to the specified schema

**`provide_experimental_design`**(*experimental_design*, *connection_string*)
Provide the experimental design for this experiment. Used to instantiate a PersistenceManager. Can only be done once per instance of SocialObjectGateway. This must be called before attemtping to persist any response data.

**Parameters**

- **`experimental_design`** (`str`) – path to an experimental design XML file
- **`connection_string`** (`str`) – database connection string for persisting data

**`provide_privacy_policy`**(*privacy_policy*)
Provide the privacy policy for this experiment. Used to instantiate an instance of PolicyProcessor. This can only be done once for an instance of SocialObjectGateway. This must be called before attempting to read or write Social Objects.

**Parameters** **`privacy_policy`** (`str`) – path to a privacy policy XML file

**`register_participant`**(*schema*, *participant*)

**`request_authentication`**(*provider*, *callback*)
Call this if it is necessary to perform authenticated API calls with a service gateway (usually required to write data as a person or to read sensitive data).

Each service gateway has its own res mechanism. Calling this will return a token needed to proceed with authentication. Authentication is completed by presenting a relevant interface to users, then calling complete_authentication() with its token.

**Parameters**

- **`provider`** (`str.`) – Name of provider to authenticate with.
- **`callback`** – URL to let PRISONER authentication server know

user has provided authentication :type callback: str. :returns: URL required to complete authentication

**restore_authentication** (*provider*, *access_token*)

Attempt to provide a service gateway with an existing access token (eg. stored in DB) to authenticate without going through clientside flow. Returns boolean value to indicate success. If False, a call should be made to requst_authentication() to begin clientside flow.

**Parameters**

- **provider** (*str*) – Name of provider to authenticate with
- **access_token** (*object*) – Object used to authenticate with this provider

:returns boolean - was authentication attempt successful?

### Module contents

## Submodules

## prisoner.SocialObjects module

**class** prisoner.SocialObjects.**Address**

Bases: *prisoner.SocialObjects.SocialObject*

Generally used as an attribute of Place, encodes a textual description of a physical address on Earth

**country**

The country name

**formatted**

A full textual representation of the address, formatted as for printing a mailing label

**locality**

The city, town, village, etc.

**postalCode**

The zip or postal code

**region**

The state or region

**streetAddress**

The street address including house number, street name, PO Box

**class** prisoner.SocialObjects.**Collection**

Bases: *prisoner.SocialObjects.SocialObject*

Represents a generic collection of SocialObjects. It may contain any number and any combination of SocialObjects.

**objects**

The collection of objects. Should be a list or SocialObjects instances.

**class** prisoner.SocialObjects.**Comment**

Bases: *prisoner.SocialObjects.SocialObject*

A textual response to another SocialObject. The base type should not be used for replying with rich content - video or images, etc.

**inReplyTo**

The SocialObject (or set of objects) this comment is in response to.

**class** prisoner.SocialObjects.**DateTimeJSONHandler** (*context*)

Bases: jsonpickle.handlers.BaseHandler

**`flatten`**(*obj*, *data*)

**`restore`**(*data*)

**class** `prisoner.SocialObjects.`**`Event`**

Bases: *`prisoner.SocialObjects.SocialObject`*

An event occuring in a place during a time interval.

**`attending`**
: A collection of People who have RSVP'd to an event

**`endTime`**
: A time object representing when the event ends

**`maybeAttending`**
: A collection of People who have responded to say they may attend the event

**`notAttending`**
: A collection of People who have responded to say they are not attending the event

**`startTime`**
: A time object representing when the event starts

**class** `prisoner.SocialObjects.`**`Image`**

Bases: *`prisoner.SocialObjects.SocialObject`*

A graphical image, such as a photo.

**`fullImage`**
: A URI for a full-size version of this image.

**exception** `prisoner.SocialObjects.`**`InvalidTransformationLevelError`**(*value*)

Bases: `exceptions.Exception`

**class** `prisoner.SocialObjects.`**`Note`**

Bases: *`prisoner.SocialObjects.SocialObject`*

A short text message, often used in a microblogging context, or to share short status updates. Shorter than blog posts, Notes are expected to have a shorter life and might not even expose a permalink

**class** `prisoner.SocialObjects.`**`Person`**

Bases: *`prisoner.SocialObjects.SocialObject`*

A human actor involved in the exchange of SocialObjects.

**`image`**
: An instance of Image used to visually represent this Person.

**class** `prisoner.SocialObjects.`**`Place`**

Bases: *`prisoner.SocialObjects.SocialObject`*

A location on Earth. For maximum flexibility, use geographic coordinates. Alternatively, a physical address or free-form location name may be provided, so long as the applications which consume Place objects can understand its semantics. A combination of location identifiers may be used.

**`address`**
: An instance of Address, for encoding a textual addresss

**`position`**
: Latitude, longitude and altitude of point on Earth. This must be an ISO 6709 string (eg. "+27.5916+086.5640+8850/")

**position_as_dict**()
Converts the internal ISO 6709 representation to a dictionary with 'lat' and 'lng' components, non-destructively

**class** prisoner.SocialObjects.**SocialObject**
Bases: object

SocialObjects are representations of social data, consumed and generated by a range of services and applications. Every SocialObject provides a small number of general attributes, with each implementation providing additional relevant attributes. SocialObjects must also provide transformation logic for each attribute, allowing each attribute to be sanitised to an appropriate level.

**author**
The person responsible for the creation of the object. For example, the person who wrote a post, uploaded a photo, etc. Should be an instance of Person.

**base_transform_name**(*string*, *transformation*, *level*)
The Base Social Objects package provides a number of standard transformations which are intended for use by any objects providing attributes of common types. This base transformation is designed to anonymise names of people, objects etc. but can be used for any string attribute

**Parameters**

- **string** (str) – the string to transform
- **transformation** (str) – "reduce" supported. Coarsens author object depending on value for level
- **level** (str) – first - reduce author's displayName to first name last - reduce author's displayName to last name initial - reduce author's displayName to initials of current names

**Raises** InvalidTransformationLevelError

**content**
The main content of this object. Where possible, this should be plain text, or a URI to an external resource. Avoid packing binary data into this property as it may be difficult to sanitise and serialize.

**displayName**
A natural language plain-text description of this object, without any additional markup. For example, the name of a location, or a person's full name.

**get_friendly_name**(*attribute*)
All Social Objects should include a dictionary of friendly names - mapping their attributes to human-readable terms. Friendly names may consist of several words, and must make sense in the following sentence construction:

"This experiment may retrieve this social object's <friendly name>"

Subclassed objects should provide their own self._friendly_names dictionary with mappings for each additional attribute it provides, or where it has semantically altered a base attribute. PRISONER will attempt to return a friendly name from the most specialised dictionary where possible

**Parameters** **attribute** (str) – Attribute to get friendly name of

**id**
A unique identifier for this object. Where possible, this should allow the service gateway to relate an instance of a SocialObject to its counterpart on the service

**location**
An instance of Place to indicate the location of an object, or the location in which it was used.

**`provider`**
The name of the ServiceGateway which generated this object, or where it is intended to be published to. This must map to an available ServiceGateway, or not be set.

**`published`**
A time object indicating when the object was created.

**`tags`**
A collection of SocialObjects associated with this object. This object must not be dependent on the tags to be semantically correct (eg. do not embed a collection of authors as tags)

**`transform_hash`**(*content*, *level='sha224'*)
Hashes content using given algorithm. Currently only supports SHA224

**Parameters**

- **`content`** – the content to be hashed
- **`level`** – the hashing algorithm to use (only supports sha224)

**Returns** hashed object

**`transform_reduce`**(*content*, *level*)
New-style transform for reduce. This is just a wrapper around the old base_transform_name

**Parameters**

- **`content`** – the content to be transformed, which will be cast to a string
- **`level`** – the level to reduce to.

**Returns** reduced content

**`updated`**
A time object indicating when the object was last updated.

**`url`**
A permament link to this object's online representation. This should be unique to this object and ideally permanent. It is acceptable for this link to be inaccessible without authentication.

## Module contents

# CHAPTER 4

# Indices and tables

- genindex
- modindex
- search

# Python Module Index

# Index