Dallinger Documentation

Release 5.0.0

Dallinger Development Team

Nov 01, 2018

Contents

Laboratory automation for the behavioral and social sciences.

CHAPTER 1

User Documentation

These documentation topics are intended to assist people who are attempting to launch experiments and analyse their data. They cover the basics of installing and setting up Dallinger, as well as use of the command line tools.

1.1 Installation

If you would like to contribute to Dallinger, please follow these *alternative install instructions*.

1.1.1 Installation Options

Dallinger is tested with Ubuntu 14.04 LTS, 16.04 LTS, 18.04 LTS and Mac OS X locally. We do not recommended running Dallinger with Microsoft Windows, however if you do, running Ubuntu in a virtual machine is the recommend method. You can also read about what is possible with using Dallinger with Docker *here*.

1.1.2 Using Dallinger with Docker

Docker is a containerization tool used for developing isolated software environments. Read more about using Dallinger with Docker *here*.

1.1.3 Install Python

Dallinger is written in the language Python. For it to work, you will need to have Python 2.7 installed, or alternatively Python 3.6 or higher. Python 3 is the preferred option. You can check what version of Python you have by running:

```
python --version
```

You will also need to have pip installed. It is included in some of the later versions of Python 3, but not all. (pip is a package manager for Python packages, or modules if you like.)

Follow the Python installation instructions.

1.1.4 Install Postgres

Follow the Postgresql installation instructions.

1.1.5 Create the Database

Follow the Create the databases instructions.

1.1.6 Install Heroku and Redis

Follow the Heroku and Redis installation instructions.

1.1.7 Install Git

Dallinger uses Git, a distributed version control system, for version control of its code. If you do not have it installed, you can install it as follows:

OSX

```
brew install git
```

Ubuntu

```
sudo apt install git
```

1.1.8 Set up a virtual environment

Follow the Virtual environment setup instructions.

Note: if you are using Anaconda, ignore this `virtualenv` section; use `conda` to create your virtual environment. Or, see the special *Anaconda installation instructions*.

1.1.9 Install Dallinger

Install Dallinger from the terminal by running

```
pip install dallinger[data]
```

Test that your installation works by running:

```
dallinger --version
```

If you use Anaconda, installing Dallinger probably failed. The problem is that you need to install bindings for the `psycopg2` package (it helps Python play nicely with Postgres) and you must use conda for conda to know where to look for the links. You do this with:

```
conda install psycopg2
```

Then, try the above installation commands. They should work now, meaning you can move on.

Next, you'll need *access keys for AWS, Heroku, etc.*.

1.2 Installing Dallinger with Anaconda

If you are interested in Dallinger and use Anaconda, you'll need to adapt the standard instructions slightly.

1.2.1 Install psycopg2

In order to get the correct bindings, you need to install `psycopg2` before installing Dallinger.

```
conda install psycopg2
conda install Dallinger[data]
```

The *[data]* optional extra makes sure that all the data analysis dependencies are installed.

1.2.2 Confirm Dallinger works

Now, we need to make sure that Dallinger and Anaconda play nice with one another. At this point, we'd check to make sure that Dallinger is properly installed by typing

```
dallinger --version
```

into the command line. You may get a long error message. Don't panic! Add the following to your `.bash_profile`:

```
export DYLD_FALLBACK_LIBRARY_PATH=$HOME/anaconda/lib/:$DYLD_FALLBACK_LIBRARY_PATH
```

After you `source` your `.bash_profile`, you can check your Dallinger version (using the same command that we used earlier), which should return the Dallinger version that you've installed.

1.2.3 Re-link Open SSL

Finally, you'll need to re-link `openssl`. Run the following:

```
brew install --upgrade openssl
brew unlink openssl && brew link openssl --force
```

1.3 Setting Up AWS, Mechanical Turk, and Heroku

Before you can use Dallinger, you will need accounts with Amazon Web Services, Amazon Mechanical Turk, and Heroku. You will then need to create a configuration file and set up your environment so that Dallinger can access your accounts.

1.3.1 Create the configuration file

The first step is to create the Dallinger configuration file in your home directory. You can do this using the Dallinger command-line utility through

```
dallinger setup
```

which will prepopulate a hidden file `.dallingerconfig` in your home directory. Alternatively, you can create this file yourself and fill it in like so:

```
[AWS Access]
aws_access_key_id = ???
aws_secret_access_key = ???
aws_region = us-east-1
```

In the next steps, we'll fill in your config file with keys.

Note: The `.dallingerconfig` can be configured with many different parameters, see *Configuration* for detailed explanation of each configuration option.

1.3.2 Amazon Web Services API Keys

There are two ways to get API keys for Amazon Web Services. If you are the only user in your AWS account, the simplest thing to do is generate root user access keys, by following these instructions. You might be presented a dialog box with options to continue to security credentials, or get started with IAM users. If you are the only user, or you are otherwise certain that this is what you want to do (see the following note), choose "Continue to Security Credentials".

N.B. One feature of AWS API keys is that they are only displayed once, and though they can be regenerated, doing so will render invalid previously generated keys. If you are running experiments using a laboratory account (or any other kind of group-owned account), regenerating keys will stop other users who have previously generated keys from being able to use the AWS account. Unless you are sure that you will not be interrupting others' workflows, it is advised that you do **not** generate new API keys. If you are not the primary user of the account, see if you can obtain these keys from others who have successfully used AWS.

If you are not the primary user of your AWS account, or are part of a working group that shares the account, the recommended way to create the access keys is by creating AIM users and generating keys for them. If someone else manages the AWS account, ask them to generate the user and keys for you. If you need to manage the users and keys by yourself, follow these instructions.

After you have generated and saved your AWS access keys, fill in the following lines of `.dallingerconfig`, replacing `???` with your keys:

```
[AWS Access]
aws_access_key_id = ???
aws_secret_access_key = ???
```

1.3.3 Amazon Mechanical Turk

It's worth signing up for Amazon Mechanical Turk (perhaps using your AWS account from above), both as a requester and as a worker. You'll use this to test and monitor experiments. You should also sign in to each sandbox, requester and worker using the same account. Store this account and password somewhere, but you don't need to tell it to Dallinger.

1.3.4 Heroku

Next, sign up for a Heroku account.

You should see an interface that looks something like the following:

Then, log in from the command line:

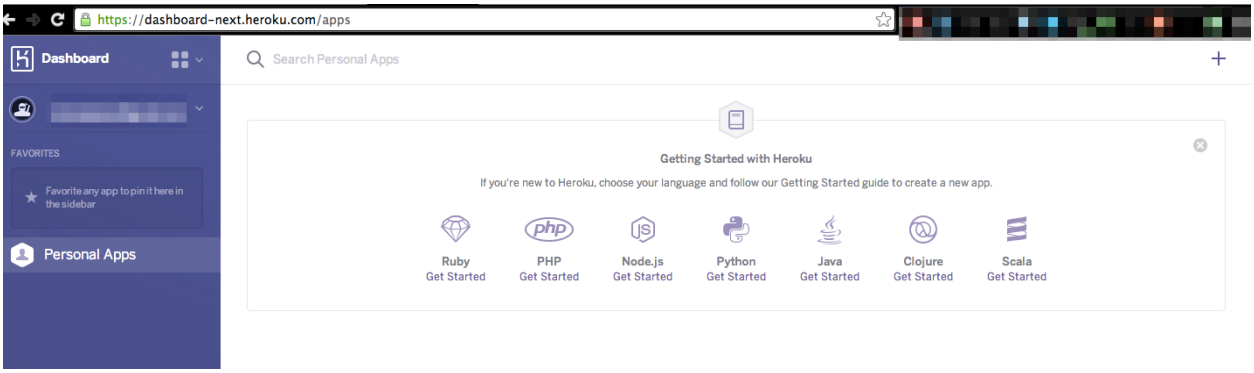


Fig. 1: This is the interface with the Heroku app

```
heroku login
```

1.3.5 Open Science Framework (optional)

There is an optional integration that uses the Open Science Framework (OSF) to register experiments. First, create an account on the OSF. Next create a new OSF personal access token on the OSF settings page. Since experiment registration requires writing to the OSF account, be sure to grant the full write scope when creating the token, by checking the osf.full_write box before creation.

Finally, fill in the appropriate section of `.dallingerconfig`:

```
[OSF]
osf_access_token = ???
```

1.3.6 Done?

Done. You're now all set up with the tools you need to work with Dallinger.

Next, we'll *test Dallinger to make sure it's working on your system*.

1.4 Command-Line Utility

Dallinger is executed from the command line within the experiment directory with the following commands:

1.4.1 verify

Verify that a directory is a Dallinger-compatible app.

1.4.2 bot

Spawn a bot and attach it to the specified application. The `--debug` flag connects the bot to the locally running instance of Dallinger. Alternatively, the `--app <app>` flag specifies a live experiment by its id.

1.4.3 debug

Run the experiment locally. An optional `--verbose` flag prints more detailed logs to the command line.

1.4.4 sandbox

Runs the experiment on MTurk's sandbox using Heroku as a server. An optional `--verbose` flag prints more detailed logs to the command line.

1.4.5 deploy

Runs the experiment live on MTurk using Heroku as a server. An optional `--verbose` flag prints more detailed logs to the command line. An optional `--bot` flag forces the bot recruiter to be used, rather than the configured recruiter.

1.4.6 logs

Open the app's logs in Papertrail. A required `--app <app>` flag specifies the experiment by its id.

1.4.7 summary

Return a summary of an experiment. A required `--app <app>` flag specifies the experiment by its id.

1.4.8 export

Download the database and partial server logs to a zipped folder within the data directory of the experimental folder. Databases are stored in CSV format. A required `--app <app>` flag specifies the experiment by its id.

1.4.9 qualify

Assign a Mechanical Turk qualification to one or more workers. Requires a `qualification`, which is a qualification ID, (or, if the `--by_name` is used, a qualification name), value `value`, and a list of one or more worker IDs, passed at the end of the command. This is useful when compensating workers if something goes wrong with the experiment.

1.4.10 revoke

Revoke a Mechanical Turk qualification for one or more workers. Requires a `qualification`, which is a qualification ID, (or, if the `--by_name` is used, a qualification name), an optional `reason` string, and a list of one or more MTurk worker IDs. This is useful when developing an experiment with "insider" participants, who would otherwise be prevented from accepting a HIT for an experiment they've already participated in.

1.4.11 hibernate

Temporarily scales down the specified app to save money. All dynos are removed and so are many of the add-ons. Hibernating apps are non-functional. It is likely that the app will not be entirely free while hibernating. To restore the app use `awaken`. A required `--app <app>` flag specifies the experiment by its id.

1.4.12 awaken

Restore a hibernating app. A required `--app <app>` flag specifies the experiment by its id.

1.4.13 destroy

Tear down an experiment server. A required `--app <app>` parameter specifies the experiment by its id. Optional `--expire-hit` flag can be provided to force expiration of MTurk HITs associated with the app (`--no-expire-hit` can be used to disable HIT expiration). If app is sandboxed, you will need to use the `--sandbox` flag to expire HITs from the MTurk sandbox.

1.4.14 hits

List all MTurk HITs for a dallinger app. A required `--app <app>` parameter specifies the experiment by its id. An optional `--sandbox` flag indicates to look for HITs in the MTurk sandbox.

1.4.15 expire

Expire all MTurk HITs for a dallinger app. A required `--app <app>` parameter specifies the experiment by its id. An optional `--sandbox` flag indicates to look for HITs in the MTurk sandbox.

1.4.16 apps

List all running heroku apps associated with the currently logged in heroku account. Returns the Dallinger app UID, app launch timestamp, and heroku app url for each running app.

1.5 Demoing Dallinger

First, make sure you have Dallinger installed:

- *Installation*
- *Developer Installation*

To test out Dallinger, we'll run a demo experiment in debug mode.

You can read more about this experiment here: Bartlett (1932) demo and download the experiment files.

Navigate to the bartlett1932 directory and run

```
dallinger debug
```

Make sure that the dallinger virtualenv is enabled so that the dallinger command is available to you.

You will see some output as Dallinger loads. When it is finished, you will see something that looks like:

```
12:00:00 PM web.1     |  2017-01-01 12:00:00,000 New participant requested: http://0.0.
↪0.0:5000/ad?assignmentId=debug9TXPFF&hitId=P8UTMZ&workerId=SP7HJ4&mode=debug
```

and your browser should automatically open to this URL. You can start interacting as the first participant in the experiment.

In the terminal, press Ctrl+C to exit the server.

Help, the experiment page is blank! This may happen if you are using an ad-blocker. Try disabling your ad-blocker and refresh the page.

It is worth noting here that occasionally if an experiment does not exit gracefully, one maybe required to manually cleanup some left over python processes, before running the same or another experiment with dallinger. See *Troubleshooting* for details.

1.6 Configuration

The Dallinger `configuration` module provides tools for reading and writing configuration parameters that control the behavior of an experiment. To use the configuration, first import the module and get the configuration object:

```
import dallinger

config = dallinger.config.get_config()
```

You can then get and set parameters:

```
config.get("duration")
config.set("duration", 0.50)
```

When retrieving a configuration parameter, Dallinger will look for the parameter first among environment variables, then in a `config.txt` in the experiment directory, and then in the `.dallingerconfig` file, using whichever value is found first. If the parameter is not found, Dallinger will use the default.

1.6.1 Built-in configuration

Built-in configuration parameters, grouped into categories:

General

`mode` ***unicode*** Run the experiment in this mode. Options include `debug` (local testing), `sandbox` (MTurk sandbox), and `live` (MTurk).

`logfile` ***unicode*** Where to write logs.

`loglevel` ***unicode*** A number between 0 and 4 that controls the verbosity of logs, from `debug` to `critical`.

`whimsical` ***boolean*** What's life without whimsy? Controls whether email notifications sent regarding various experiment errors are whimsical in tone, or more matter-of-fact.

Recruitment (General)

`auto_recruit` ***boolean*** A boolean on whether recruitment should be automatic.

`browser_exclude_rule` ***unicode - comma separated*** A set of rules you can apply to prevent participants with unsupported web browsers from participating in your experiment.

recruiter *unicode* The recruiter class to use during the experiment run. While this can be a full class name, it is more common to use the class's `nickname` property for this value; for example `mturk`, `cli`, `bots`, or `multi`. NOTE: when running in debug mode, the HotAir (`hotair`) recruiter will always be used. The exception is if the `--bots` option is passed to `dallinger debug`, in which case the BotRecruiter will be used instead.

recruiters *unicode - custom format* When using multiple recruiters in a single experiment run via the `multi` setting for the `recruiter` config key, `recruiters` allows you to specify which recruiters you'd like to use, and how many participants to recruit from each. The special syntax for this value is:

```
recruiters = [nickname 1]:  [recruits], [nickname 2]:  [recruits], etc.
```

For example, to recruit 5 human participants via MTurk, and 5 bot participants, the configuration would be:

```
recruiters = mturk:  5, bots:  5
```

Amazon Mechanical Turk Recruitment

aws_access_key_id *unicode* AWS access key ID.

aws_secret_access_key *unicode* AWS access key secret.

aws_region *unicode* AWS region to use. Defaults to `us-east-1`.

ad_group *unicode* Obsolete. See `group_name`.

assign_qualifications *boolean* A boolean which controls whether an experiment-specific qualification (based on the experiment ID), and a group qualification (based on the value of `group_name`) will be assigned to participants by the recruiter. This feature assumes a recruiter which supports qualifications, like the `MTurkRecruiter`.

group_name *unicode* Assign a named qualification to workers who complete a HIT.

qualification_blacklist *unicode - comma seperated* Comma-separated list of qualification names. Workers with qualifications in this list will be prevented from viewing and accepting the HIT.

title *unicode* The title of the HIT on Amazon Mechanical Turk.

description *unicode* The description of the HIT on Amazon Mechanical Turk.

keywords *unicode* A comma-separated list of keywords to use on Amazon Mechanical Turk.

lifetime *integer* How long in hours that your HIT remains visible to workers.

duration *float* How long in hours participants have until the HIT will time out.

us_only *boolean* Controls whether this HIT is available only to MTurk workers in the U.S.

base_payment *float* Base payment in U.S. dollars. All workers who accept the HIT are guaranteed this much compensation.

approve_requirement *integer* The percentage of past MTurk HITs that must have been approved for a worker to qualify to participate in your experiment. 1-100.

organization_name *unicode* Obsolete.

Email Notifications

See *Email Notification Setup* for a much more detailed explanation of these values and their use.

contact_email_on_error *unicode* The email address used as the recipient for error report emails, and the email displayed to workers when there is an error.

dallinger_email_address *unicode* An email address for use by Dallinger to send status emails.

smtp_host *unicode* Hostname and port of a mail server for outgoing mail. Defaults to `smtp.gmail.com:587`

smtp_username *unicode* Username for outgoing mail host.

smtp_password *unicode* Password for the outgoing mail host.

Deployment Configuration

database_url *unicode* URI of the Postgres database.

database_size *unicode* Size of the database on Heroku. See Heroku Postgres plans.

dyno_type *unicode* Heroku dyno type to use. See Heroku dynos types.

redis_size *unicode* Size of the redis server on Heroku. See Heroku Redis.

num_dynos_web *integer* Number of Heroku dynos to use for processing incoming HTTP requests. It is recommended that you use at least two.

num_dynos_worker *integer* Number of Heroku dynos to use for performing other computations.

host *unicode* IP address of the host.

port *unicode* Port of the host.

notification_url *unicode* URL where notifications are sent. This should not be set manually.

clock_on *boolean* If the clock process is on, it will perform a series of checks that ensure the integrity of the database.

heroku_team *unicode* The name of the Heroku team to which all applications will be assigned. This is useful for centralized billing. Note, however, that it will prevent you from using free-tier dynos.

worker_multiplier *float* Multiplier used to determine the number of gunicorn web worker processes started per Heroku CPU count. Reduce this if you see Heroku warnings about memory limits for your experiment. Default is *1.5*

1.6.2 Choosing configuration values

When running real experiments it is important to pick configuration variables that result in a deployment that performs appropriately.

The number of Heroku dynos that are required and their specifications can make a very large difference to how the application behaves.

num_dynos_web This configuration variable determines how many dynos are run to deal with web traffic. They will be transparently load-balanced, so the more web dynos are started the more simultaneous HTTP requests the stack can handle. If an experiment defines the `channel` variable to subscribe to websocket events then all of these callbacks happen on the dyno that handles the initial `/launch` POST, so experiments that use this functionality heavily receive significantly less benefit from increasing `num_dynos_web`. The optimum value differs between experiments, but a good rule of thumb is 1 web dyno for every 10-20 simultaneous human users.

num_dynos_worker Workers are dynos that pull tasks from a queue and execute them in the background. They are optimized for many short tasks, but they are also used to run bots which are very long-lived. Each worker can run up to 20 concurrent tasks, however they are co-operatively multitasked so a poorly behaving task can cause all others sharing its host to block. When running with bots, you should always pick a value of `num_dynos_worker` that is at least ``0.05*number_of_bots``, otherwise it is guaranteed to fail. In practice, there may well be experiment-specific tasks that also need to execute, and bots are more performant on underloaded dynos, so a better heuristic is `0.25*number_of_bots`.

dyno_type This determines how powerful the heroku dyno that's started is. It applies to both web and worker dyno types. The minimum recommended is `standard-1x`, which should be sufficient for experiments that do not rely on real-time coordination, such as demos/bartlett1932/index. Experiments that require significant power to process websocket events should consider the higher levels, `standard-2x`, `performance-m` and `performance-l`. In all but the most intensive experiments, either `dyno_type` or `num_dynos_web` should be increased, not both.

redis_size A larger value for this increases the number of connections available on the redis dyno. This should be increased for experiments that make substantial use of websockets. Values are `premium-0` to `premium-14`. It is very unlikely that values higher than `premium-5` are useful.

duration The duration parameter determines the number of hours that an MTurk worker has to complete the experiment. Choosing numbers that are too short can cause people to refuse to work on a HIT. A deadline that is too long may give people pause for thought as it may make the task seem underpaid. Set this to be significantly above the total time from start to finish that you'd expect a user to take in the worst case.

base_payment The amount of US dollars to pay for completion of the experiment. The higher this is, the easier it will be to attract workers.

1.7 Email Notification Setup

Dallinger can be configured to send email messages when errors occur during a running experiment. If this configuration is skipped, messages which would otherwise be emailed will be written to the experiment logs instead.

1.7.1 Instructions

Sending email from Dallinger requires 5 configuration settings, described in turn below. Like all configuration settings, they can be set up in either *.dallingerconfig* in your home directory, or in *config.txt* in the root directory of your experiment.

The Config Settings

smtp_host The hostname and port of the SMTP (outgoing email) server through which all email will be sent. This defaults to *smtp.gmail.com:587*, the Google SMTP server. If you want to send email *from* a Gmail address, or a custom domain set up to use Gmail for email, this default setting is what you want.

smtp_username The username with which to log into the SMTP server, which will very likely be an email address (if you are using a Gmail address to send email, you will use that address for this value).

smtp_password The password associated with the `smtp_username`.

NOTE If you are using two-factor authentication, see *Two-Factor Authentication*, below.

dallinger_email_address The email address to be used as the "from" address outgoing email notifications. For Gmail accounts, this address is likely to be overwritten by the Google SMTP server. See *Gmail "From" address rewriting* below.

contact_email_on_error Also an email address, and used in two ways:

1. It serves as the recipient address for outgoing notifications
2. It is displayed to experiment participants on the error page, so that they can make inquiries about compensation

Pitfalls and Solutions

A few other things which may get in the way of sending email successfully, or cause things to behave differently than expected:

Two-Factor Authentication

Having two-factor authentication enabled for the outgoing email account will prevent Dallinger from sending email without some additional steps. Detailed instructions are provided for Gmail, below. Other email services which support two-factor authentication may provide equivalent solutions.

Working with Google/Gmail Two-factor Authentication

If you are using Gmail with two-factor authentication, we recommend that you set up an application-specific password (what Google short-hands as "App password") specifically for Dallinger. You can set one up following these instructions (adapted from here):

1. Log into your Gmail web interface as usual, using two-factor authentication if necessary.
2. Click your name or photo near your Gmail inbox's top right corner.
3. Follow the *Google Account* link in the drop-down/overlay that appears.
4. Click *Signing in to Google* in the *Sign-in & security* section.
5. Under the *Password & sign-in method* section, click *App passwords*. (If prompted for your Gmail password, enter it and click *Next*.)
6. Select *Other (custom name)* in the *Select app* drop-down menu. Enter *Dallinger outgoing mail* or another descriptive name so you'll recognize what it's for when you view these settings in the future.
7. Click *Generate*.
8. Find and immediately copy the password under *Your app passwords*. Type or paste the password into the *.dallingerconfig* file in your home directory. You will not be able to view the password again, so if you miss it, you'll need to delete the one you just created and create a new one.
9. Click *Done*.

Firewall/antivirus

When developing locally, antivirus or firewall software may prevent outgoing email from being sent, and cause Dallinger to raise a *socket.timeout* error. Temporarily disabling these tools is the easiest workaround.

Google "Less secure apps"

If you do **not** have two-factor authentication enabled, Gmail may require that you enable "less secure apps" in order to send email from Dallinger. You will likely know you are encountering this problem because you will receive warning email messages from Google regarding "blocked sign-in attempts". To enable this, sign into Gmail, go to the *Less secure apps* section under *Google Account*, and turn on *Allow less secure apps*.

Gmail "From" address rewriting

Google automatically rewrites the *From* line of any email you send via its SMTP server to the default *Send mail as* address in your Gmail or Google Apps email account setting. This will result in the *dallinger_email_address* value being ignored, and the *smtp_username* appearing in the "From" header instead. A possible workaround: in your Google email under *Settings*, go to the *Accounts* tab/section and make "default" an account other than your Gmail/Google Apps account. This will cause Google's SMTP server to re-write the *From* field with this address instead.

Debug Mode

Email notifications are never sent when Dallinger is running in "debug" mode. The text of messages which would have been emailed will appear in the logging output instead.

1.8 Running Experiments Programmatically

Dallinger experiments can be run through a high-level Python API.

```
import dallinger

experiment = dallinger.experiments.Bartlett1932()
data = experiment.run({
    mode=live,
    base_payment=1.00,
})
```

All parameters in `config.txt` and `.dallingerconfig` can be specified in the configuration dictionary passed to the *`run()`* function. The return value is an object that allows you to access all the Dallinger data tables in a variety of useful formats. The following data tables are available:

```
data.infos
data.networks
data.nodes
data.notifications
data.participants
data.questions
data.transformations
data.transmissions
data.vectors
```

For each of these tables, e.g. `networks`, you can access the data in a variety of formats, including:

```
data.networks.csv    # Comma-separated value
data.networks.dict   # Python dictionary
data.networks.df     # pandas DataFrame
data.networks.html   # HTML table
data.networks.latex  # LaTeX table
data.networks.list   # Python list
data.networks.ods    # OpenDocument Spreadsheet
data.networks.tsv    # Tab-separated values
data.networks.xls    # Legacy Excel spreadsheet
data.networks.xlsx   # Modern Excel spreadsheet
data.networks.yaml   # YAML
```

See *Database API* for more details about these tables.

1.8.1 Parameterized Experiment Runs

This high-level API is particularly useful for running an experiment in a loop with modified configuration for each run. For example, an experimenter could run repeated ConcentrationGame experiments with varying numbers of participants:

```
import dallinger

collected = []
experiment = dallinger.experiments.ConcentrationGame()
for run_num in range(1, 10):
    data = experiment.run({
        mode=live,
        num_participants=run_num,
    })
    collected.append(data)
```

With this technique, an experimenter can use data from prior runs to modify the configuration for subsequent experiment runs.

1.8.2 Repeatability

It is often useful to share the code used to run an experiment in a way that ensures that re-running it will retrieve the same results. Dallinger provides a special method for that purpose: `collect()`. This method is similar to `run()` but it requires an *app_id* parameter. When that *app_id* corresponds to existing experiment data that can be retrieved (from either a local export or stored remotely), that data will be loaded. Otherwise, the experiment is run and the data is saved under the provided *app_id* so that subsequent calls to `collect()` with that *app_id* will retrieve the data instead of re-running the experiment.

For example, an experimenter could pre-generate a UUID using *dallinger uuid*, then collect data using that UUID:

```
import dallinger

my_app_id = "68f73876-48f3-d1e2-4df7-25e46c99ce28"
experiment = dallinger.experiments.Bartlett1932()
data = experiment.collect(my_app_id, {
    mode=live,
    base_payment=1.00,
})
```

The first run of the above code will run a live experiment and collect data. Subsequent runs will retrieve the data collected during the first run.

1.8.3 Importing Your Experiment

You can use this API directly on an imported experiment class if it is available in your python path:

```
from mypackage.experiment import MyFancyExperiment
data = MyFancyExperiment().run(...)
```

Alternatively, an experiment installed as a python package can register itself with Dallinger and appear in the experiments module. This is done by including a *dallinger.experiments* item in the *entry_points* argument in the call to *setup* in an experiment's *setup.py*. For example:

```
...
setup(
    ...,
    entry_points={'dallinger.experiments': ['mypackage.MyFancyExperiment']},
    ...
)
```

An experiment package registered in this manner can be imported from *dallinger.experiments*:

```
import dallinger

experiment = dallinger.experiments.MyFancyExperiment()
experiment.run(...)
```

See the *setup.py* from *dlgr.demos* for more examples.

1.9 Monitoring a Live Experiment

There are a number of ways that you can monitor a live experiment:

1.9.1 Command line tools

`dallinger summary --app {#id}`, where `{#id}` is the id (`w...`) of the application.

This will print a summary showing the number of participants with each status code, as well as the overall yield:

```
status  | count
----------------
1   | 26
101 | 80
103 | 43
104 | 2

Yield: 64.00%
```

1.9.2 Papertrail

You can use Papertrail to view and search the live logs of your experiment. You can access the logs either through the Heroku dashboard's Resources panel (https://dashboard.heroku.com/apps/{#id}/resources), where {#id} is the id of your experiment, or directly through Papertrail.com (https://papertrailapp.com/systems/{#id}/events).

Setting up alerts

You can set up Papertrail to send error notifications to Slack or another communications platform.

0. Take a deep breath.
1. Open the Papertrail logs.

2. Search for the term `error`.
3. To the right of the search bar, you will see a button titled "+ Save Search". Click it. Name the search "Errors". Then click "Save & Setup an Alert", which is to the right of "Save Search".
4. You will be directed to a page with a list of services that you can use to set up an alert.
5. Click, e.g., Slack.
6. Choose the desired frequency of alert. We recommend the minimum, 1 minute.
7. Under the heading "Slack details", open (*in a new tab or window*) the link new Papertrail integration.
8. This will bring you to a Slack page where you will choose a channel to post to. You may need to log in.
9. Select the desired channel.
10. Click "Add Papertrail Integration".
11. You will be brought to a page with more information about the integration.
12. Scroll down to Step 3 to get the Webhook URL. It should look something like `https://hooks.slack.com/services/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`.
13. Copy this link to your clipboard.
14. Change anything else you want and then scroll to the bottom and click "Save integration".
15. Go back to Papertrail page that you left in Step 7.
16. Paste the copied URL into the input text box labeled "Integration's Webhook URL" under the "Slack Details" heading.
17. Click "Create Alert" on the same page.
18. Victory.

1.10 Experiment Data

Dallinger keeps track of experiment data using the database. All generated data about Dallinger constructs, like networks, nodes, and participants, is tracked by the system. In addition, experiment specific data, such as questions and infos, can be stored.

The *info* table is perhaps the most useful for experiment creators. It is intended for saving data specific to an experiment. Whenever an important event needs to be recorded for an experiment, an Info can be created:

```
def record_event(self, node, contents, details):
    info = Info(origin=node, contents=contents, details=details)
    session.add(info)
    session.commit()
```

In the above example, we have a function to record an event that would be part of a long experiment code. Each time something important happens in the experiment, the function will be called. In this case, we take the related node as the first parameter, then a string representation of the event, and finally an optional details parameter, which can include a dictionary, or other data structure with details.

Dallinger allows users to export experiment data for performing analysis with the tools of their choice. Data from all experiment tables are exported in CSV format, which makes it easy to use in a variety of tools.

To export the data, the Dallinger *export* command is used. The command requires passing in the application id. Example:

```
$ dallinger export app=6ab5e918-44c0-f9bc-5d97-a5ddbbddb68a
```

This will connect to the database and export the data, which will be saved as a zip file inside the *data* directory:

```
$ ls data
6ab5e918-44c0-f9bc-5d97-a5ddbbddb68a.zip
```

To use the exported data, it is recommended that you unzip the file inside a working directory. This will create a new *data* directory, which will contain the experiment's exported tables as CSV files:

```
$ unzip 6ab5e918-44c0-f9bc-5d97-a5ddbbddb68a.zip
Archive:  6ab5e918-44c0-f9bc-5d97-a5ddbbddb68a-data.zip
  inflating: experiment_id.md
  inflating: data/network.csv
  inflating: data/info.csv
  inflating: data/notification.csv
  inflating: data/question.csv
  inflating: data/transformation.csv
  inflating: data/vector.csv
  inflating: data/transmission.csv
  inflating: data/participant.csv
  inflating: data/node.csv
```

Once the data is uncompressed, you can analyze it using many different applications. Excel, for example, will easily import the data, just by double clicking on one of the files.

In Python, pandas are a popular way of manipulating data. The library is required by Dallinger, so if you already have Dallinger running you can begin using it right away:

```
$ python
>>> import pandas
>>> df = pandas.read_csv('question.csv')
```

Pandas has a handy *read_csv* method, which will read a CSV file and convert it to a DataFrame, which is a sort of spreadsheet-like structure used by Pandas to work with data. Once the data is in a DataFrame, we can use all the DataFrame features to work with the data:

```
>>> df.info()
    <class 'pandas.core.frame.DataFrame'>
    RangeIndex: 6 entries, 0 to 5
    Data columns (total 14 columns):
    id                6 non-null int64
    creation_time     6 non-null datetime64[ns]
    property1         0 non-null object
    property2         0 non-null object
    property3         0 non-null object
    property4         0 non-null object
    property5         0 non-null object
    failed            6 non-null object
    time_of_death     0 non-null object
    type              6 non-null object
    participant_id    6 non-null int64
    number            6 non-null int64
    question          6 non-null object
    response          6 non-null object
    dtypes: datetime64[ns](1), int64(3), object(10)
    memory usage: 744.0+ bytes
```

(continues on next page)

(continued from previous page)

```
    None
    >>> df.response.describe()
    count                                          6
    unique                                         5
    top       {"engagement":"7","difficulty":"4"}
    freq                                           2
    Name: response, dtype: object
```

In this case, let's say we want to analyze questionnaire responses at the end of an experiment. We will only need the *response* column from the *question* table. Also, since this column is stored as a string, but holds a dictionary with the answers to the questions, we need to convert it into a suitable format for analysis:

```
>>> df = pandas.read_csv('question.csv', usecols=['response'],
            converters={'response': lambda x:eval(x).values()})
>>> df
      response
    0   [4, 7]
    1   [1, 6]
    2   [4, 7]
    3   [7, 7]
    4   [3, 6]
    5   [0, 3]
>>> responses=pandas.DataFrame(df['response'].values.tolist(),
            columns=['engagement', 'difficulty'], dtype='int64')
>>> responses
      engagement difficulty
    0          4          7
    1          1          6
    2          4          7
    3          7          7
    4          3          6
    5          0          3
```

First we create a DataFrame using *read_csv* as before, but this time, we specify which columns to use using the *usecols* parameter. To get the numeric values for the responses, we use a converter to convert the string back into a dictionary and extract the values.

At this point, we have both values in the response column. We really want to have one column for each value, so we create a new dataframe, converting the response values to a list and assigning each to a named column. We also make sure the values are integers, with the *dtype* parameter. This makes them plottable.

We can now make a simple bar chart of the responses using plot:

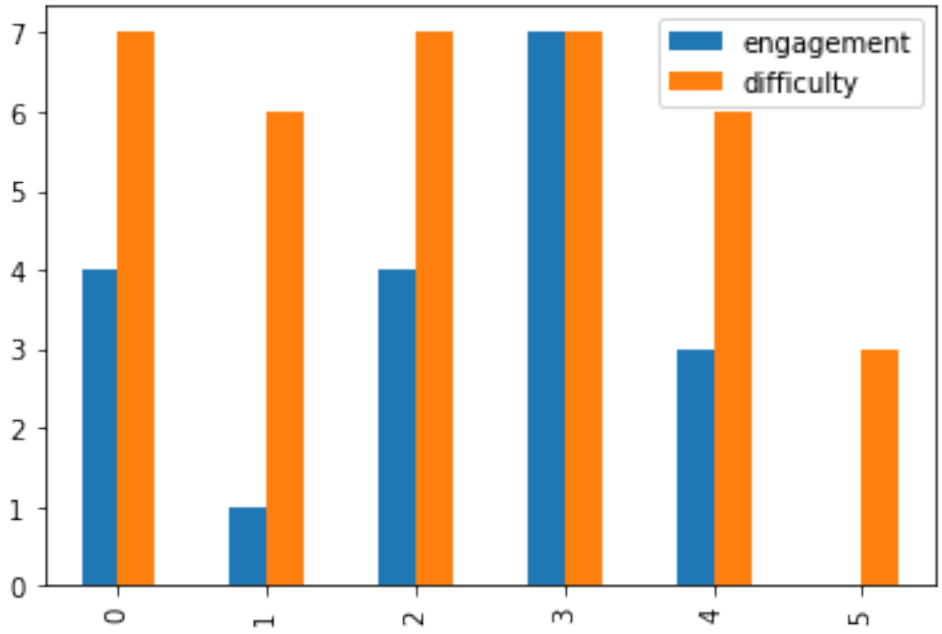
```
>>> responses.plot(kind='bar')
<matplotlib.axes._subplots.AxesSubplot at 0x7f7f0092dc90>
```

If you are running this in a Jupyter notebook, this would be the result:

Of course these are very simple examples. Pandas are a powerful library, and offer many analysis and visualization methods, but this should at least give an idea of what can be achieved.

Dallinger also has a helper class that allows us to handle experiment data in different formats. You can get the DataFrame using this, as well:

```
$ python
>>> from dallinger.data import Table
>>> data = Table('info.csv')
>>> df = data.df
```



It might seem like a roundabout way to get the DataFrame, but the table class has the advantage that the data can easily be converted to many other formats. All of these formats are accessed as properties of the Table instance, like *data.df* above. Supported formats are:

- csv. Comma-separated values.
- dict. A python dictionary.
- df. A pandas DataFrame.
- html. An html table.
- latex. A LaTex table.
- list. A python list.
- ods. An open document spreadsheet.
- tsv. Tab separated values.
- xls. Legacy Excel spreadsheet.
- xlsx. Excel spreadsheet.
- yaml. YAML format.

From the list above *dict*, *df*, and *list* can be used to handle the data inside a python interpreter or program, and the rest are better suited for display or analysis using other tools.

1.11 Viewing the PostgreSQL Database

1.11.1 OSX

Postico is a nice tool for examining Postgres databases on OS X. We use it to connect to live experiment databases. Here are the steps needed to do this:

1. Download Postico and place it in your Applications folder.
2. Open Postico.
3. Press the "New Favorite" button in the bottom left corner to access a new database.
4. Get the database credentials from the Heroku dashboard:
    - Go to https://dashboard.heroku.com/apps/{app_id}/resources
    - Under the **Add-ons** subheading, go to "Heroku Postgres :: Database"
    - Note the database credentials under the subheading "Connection Settings". You'll use these in step 5.
5. Fill in the database settings in Postico. You'll need to include the:
    - Host
    - Port
    - User
    - Password
    - Database
6. Connect to the database.
    - You may see a dialog box pop up saying that Postico cannot verify the identity of the server. Click "Connect" to proceed.

1.11.2 Ubuntu

pgAdmin4 can be used to inspect the contents of the database. Read more about it here.

1.12 Running bots as participants

Dallinger supports running simulated experiments using bots that participate in the experiment automatically.

Not all experiments will have bots available; the demos/bartlett1932/index and demos/chatroom/index demos are the only built-in experiments that do.

1.12.1 Running an experiment locally with bots

To run the experiment in debug mode using bots, use the *–bot* flag:

```
$ dallinger debug --bot
```

This overrides the *recruiter* configuration key to use the `BotRecruiter`. Instead of printing the URL for a participant or recruiting participants using Mechanical Turk, the bot recruiter will start running bots.

You may also set the configuration value `recruiter='bots'` in local or global configurations, as an environment variable or as a keyword argument to `run()`.

Note: Bots are run by worker processes. If the experiment recruits many bots at the same time, you may need to increase the `num_dynos_worker` config setting to run additional worker processes. Each worker process can run up to 20 bots (though if the bots are implemented using selenium to run a real browser, you'll probably hit resource limits before that).

1.12.2 Running an experiment with a mix of bots and real participants

It's also possible to run an experiment that mixes bot participants with real participants. To do this, edit the experiment's `config.txt` to specify recruiter configuration like this:

```
recruiter = multi
recruiters = bots: 2, cli: 1
```

The `recruiters` config setting is a specification of how many participants to recruit from which recruiters in what order. This example says to use the bot recruiter the first 2 times that the experiment requests a participant to be recruited, followed by the CLI recruiter the third time. (The CLI recruiter writes the participant's URL to the log, which triggers opening it in your browser if you are running in debug mode.)

To start the experiment with this configuration, run:

```
$ dallinger debug
```

Running a single bot

If you want to run a single bot as part of an ongoing experiment, you can use the *bot* command. This is useful for testing a single bot's behavior as part of a longer-running experiment, and allows easy access to the Python pdb debugger.

1.13 Registration on the OSF

Dallinger integrates with the Open Science Framework (OSF), creating a new OSF project and uploading your experiment code to the project on launch. To enable, specify a personal access token `osf_access_token` in your `.dallingerconfig` file. You can generate a new OSF personal access token on the OSF settings page.

1.14 Troubleshooting

A few common issues are reported when trying to run Dallinger. Always run with the *–verbose* flag for full logs

1.14.1 Python Processes Kept Alive

Sometimes when trying to run experiments consecutively in Debug mode, a straggling process creates Server 500 errors. These are caused by background python processes and/or gunicorn workers. Filter for them using:

```
ps -ef | grep -E "python|gunicorn"
```

This will display all running processes that have the name *python* or *gunicorn*. To kill all of them, run these commands:

```
pkill python
pkill gunicorn
```

1.14.2 Known Postgres issues

If you get an error like the following...

```
createuser: could not connect to database postgres: could not connect to server:
    Is the server running locally and accepting
    connections on Unix domain socket "/tmp/.s.PGSQL.5432"?
```

. . . then you probably did not start the app.

If you get a fatal error that your ROLE does not exist, run these commands:

```
createuser dallinger
dropdb dallinger
createdb -O dallinger dallinger
```

1.14.3 Common Sandbox Error

```
 Launching the experiment on MTurk...

 Error parsing response from /launch, check web dyno logs for details: <!DOCTYPE html>
    <html>
      <head>
        <meta name="viewport" content="width=device-width, initial-scale=1">
        <meta charset="utf-8">
        <title>Application Error</title>
        <style media="screen">
          html,body,iframe {
            margin: 0;
            padding: 0;
          }
          html,body {
            height: 100%;
            overflow: hidden;
          }
          iframe {
            width: 100%;
            height: 100%;
            border: 0;
          }
        </style>
      </head>
      <body>
        <iframe src="//www.herokucdn.com/error-pages/application-error.html"></iframe>
      </body>
    </html>
Traceback (most recent call last):
  File "/Users/user/.virtualenvs/dallinger/bin/dallinger", line 11, in <module>
    load_entry_point('dallinger', 'console_scripts', 'dallinger')()
  File "/Users/user/.virtualenvs/dallinger/lib/python3.6/site-packages/click/core.py",
↪ line 722, in __call__
    return self.main(*args, **kwargs)
  File "/Users/user/.virtualenvs/dallinger/lib/python3.6/site-packages/click/core.py",
↪ line 697, in main
    rv = self.invoke(ctx)
  File "/Users/user/.virtualenvs/dallinger/lib/python3.6/site-packages/click/core.py",
↪ line 1066, in invoke
    return _process_result(sub_ctx.command.invoke(sub_ctx))
  File "/Users/user/.virtualenvs/dallinger/lib/python3.6/site-packages/click/core.py",
↪ line 895, in invoke
```

(continues on next page)

(continued from previous page)

```
    return ctx.invoke(self.callback, **ctx.params)
  File "/Users/user/.virtualenvs/dallinger/lib/python3.6/site-packages/click/core.py",
↪ line 535, in invoke
    return callback(*args, **kwargs)
  File "/Users/user/Dallinger/dallinger/command_line.py", line 558, in sandbox
    _deploy_in_mode(u'sandbox', app, verbose)
  File "/Users/user/Dallinger/dallinger/command_line.py", line 550, in _deploy_in_mode
    deploy_sandbox_shared_setup(verbose=verbose, app=app)
  File "/Users/user/Dallinger/dallinger/command_line.py", line 518, in deploy_sandbox_
↪shared_setup
    launch_data = _handle_launch_data('{}/launch'.format(heroku_app.url))
  File "/Users/user/Dallinger/dallinger/command_line.py", line 386, in _handle_launch_
↪data
    launch_data = launch_request.json()
  File "/Users/user/.virtualenvs/dallinger/lib/python3.6/site-packages/requests/
↪models.py", line 892, in json
    return complexjson.loads(self.text, **kwargs)
  File "/Library/Frameworks/Python.framework/Versions/3.6/lib/python3.6/json/__init__.
↪py", line 339, in loads
    return _default_decoder.decode(s)
  File "/Library/Frameworks/Python.framework/Versions/3.6/lib/python3.6/json/decoder.
↪py", line 364, in decode
    obj, end = self.raw_decode(s, idx=_w(s, 0).end())
  File "/Library/Frameworks/Python.framework/Versions/3.6/lib/python3.6/json/decoder.
↪py", line 382, in raw_decode
    raise ValueError("No JSON object could be decoded")
```

If you get this from the sandbox, this usually means there's a deeper issue that requires *dallinger logs –app XXXXXX*. Usually this could be a requirements.txt file error (missing dependency or reference to an incorrect branch).

1.14.4 Combining Dallinger core development and running experiments

A common pitfall while doing development on the dallinger codebase while also working on external experiments which include dallinger as a dependency: you pip install a demo experiment in your active virtual environment, and it overwrites the dallinger.egg-link file in that environment's site-packages directory with an actual copy of the dallinger package.

When installing dallinger with the intent to work on dallinger, the recommended way to install dallinger itself is with pip's "editable mode", by passing the -e or –editable flag to pip install:

```
pip install -e .[data]
```

This creates a form of symbolic link in the active python's site-packages directory to the working copy of dallinger you're sitting in. This allows you to make changes to python files in the dallinger working copy and have them immediately active when using dallinger commands or any other actions that invoke the active python interpreter.

Running pip install without the -e flag, either while installing dallinger directly, or while installing a separate experiment which includes dallinger as a dependency, will instead place a copy of the dallinger package in the site-packages directory. These files will then be executed when the active python is running, and any changes to the files you're working on will be ignored.

You can check to see if you are working in "editable mode" by inspecting the contents of your active virtual environment's site-packages folder. In "editable mode", you will see a dallinger.egg-link file listed in the directory:

```
...
drwxr-xr-x    9 jesses  staff   306B May 29 12:30 coverage_pth-0.0.2.dist-info
-rw-r--r--    1 jesses  staff    44B May 29 12:30 coverage_pth.pth
-rw-r--r--    1 jesses  staff    33B Jun 14 16:08 dallinger.egg-link
drwxr-xr-x   21 jesses  staff   714B Mar 19 17:24 datashape
drwxr-xr-x   10 jesses  staff   340B Mar 19 17:24 datashape-0.5.2.dist-info
...
```

The contents of this file will include the path to the working copy that's active. If you instead see a directory tree with actual dallinger files, you can restore "editable mode" by re-running the installation steps for dallinger from the "Developer Installation" documentation.

CHAPTER 2

Dallinger Demos

Several demos that demonstrate Dallinger in action can be found here.

CHAPTER 3

Experiment Author Documentation

These documentation topics build on the previous set to include help with designing new experiments for others to use.

3.1 Developer Installation

We recommend installing Dallinger on Mac OS X. It's also possible to use Ubuntu, either directly or in a virtual machine. If you are attempting to use Dallinger on Microsoft Windows, running Ubuntu in a virtual machine is the recommend method.

If you are interested in using Dallinger with Docker, read more *here*.

3.1.1 Install Python

It recommended that you run Dallinger on Python 3. Dallinger has been tested to work on Python 3.6 and up. Dallinger also supports Python 2.7

You can check what version of Python you have by running:

```
python --version
```

Follow the Python installation instructions.

3.1.2 Install Postgres

Follow the Postgresql installation instructions.

3.1.3 Create the Databases

Follow the Create the databases instructions.

3.1.4 Install Heroku and Redis

Follow the Heroku and Redis installation instructions.

3.1.5 Set up a virtual environment

Follow the Virtual environment setup instructions.

Note: if you are using Anaconda, ignore this `virtualenv` section; use `conda` to create your virtual environment. Or, see the special *Anaconda installation instructions*.

3.1.6 Install prerequisites for building documentation

To be able to build the documentation, you will need yarn.

Please follow the instructions here to install it.

3.1.7 Install Dallinger

Next, navigate to the directory where you want to house your development work on Dallinger. Once there, clone the Git repository using:

```
git clone https://github.com/Dallinger/Dallinger
```

This will create a directory called `Dallinger` in your current directory.

Change into your the new directory and make sure you are still in your virtual environment before installing the dependencies. If you want to be extra careful, run the command `workon dallinger`, which will ensure that you are in the right virtual environment.

Note: if you are using Anaconda – as of August 10, 2016 – you will need to follow special *Anaconda installation instructions*. This should be fixed in future versions.

```
cd Dallinger
```

Now we need to install the dependencies using pip:

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

Next run `setup.py` with the argument `develop`:

```
pip install -e .[data]
```

Test that your installation works by running:

```
dallinger --version
```

Note: if you are using Anaconda and get a long traceback here, please see the special *Installing Dallinger with Anaconda*.

3.1.8 Install the dlgr.demos sub-package

Both the test suite and the included demo experiments require installing the `dlgr.demos` sub-package in order to run. Install this in "develop mode" with the `-e` option, so that any changes you make to a demo will be immediately reflected on your next test or debug session.

From the root `Dallinger` directory you created in the previous step, run the installation command:

```
pip install -e demos
```

Next, you'll need *access keys for AWS, Heroku, etc.*.

3.2 Creating an Experiment

The easiest way to create an experiment is to use the Dallinger Cookiecutter template. Cookiecutter is a tool that creates projects from project templates. There is a Dallinger template available for this tool.

The first step is to get Cookiecutter itself installed. Like Dallinger, Cookiecutter uses Python, so it can be installed in the same way that Dallinger was installed. If you haven't installed Dallinger yet, please consult the *installation instructions* first.

In most cases, you can install Cookiecutter using Python's *pip* installer:

```
pip install cookiecutter
```

After that, you can use the *cookiecutter* command to create a new experiment in your current directory:

```
cookiecutter https://github.com/Dallinger/cookiecutter-dallinger.git
```

Cookiecutter works by asking some questions about the project you are going to create, and uses that information to set up a directory structure that contains your project. A Dallinger experiment is a Python package, so you'll need to answer a few questions about this before Cookiecutter creates your experiment's directory.

The questions are below. Be sure to follow indications about allowed characters, or your experiment may not run:

- *namespace*: This can be used as a general "container" or "brand" name for your experiments. It should be all lower case and not contain any spaces or special characters other than _.
- *experiment_name*: The experiment will be stored in this sub-directory. This should be all lower case and not contain any spaces or special characters other than _.
- *repo_name*: The GitHub repository name where experiment package will eventually live. This should not contain any spaces or special characters other than - and _.
- *package_name*: The python package name for your experiment. This is usually the name of your namespace and your experiment name separated by a dot. This should be all lower case and not contain any spaces or special characters other than _.
- *experiment_class*: The python class name for your custom experiment class. This should not contain any spaces or special characters. This is where the main code of your experiment will live.
- *experiment_description*: A short description of your experiment
- *author*: The package author's full name
- *author_email*: The contact email for the experiment author.
- *author_github*: The GitHub account name where the package will eventually live.

If you do not intend to publish your experiment and do not plan to store it in a github repository, you can just hit <enter> when you get to those questions. The defaults should be fine. Just make sure to have an original answer for the *experiment_name* question, and you should be good to go.

A sample Cookiecutter session is shown below. Note that the questions begin right after Cookiecutter downloads the project repository:

```
$ cookiecutter https://github.com/Dallinger/cookiecutter-dallinger.git
Cloning into 'cookiecutter-dallinger'...
remote: Counting objects: 150, done.
remote: Compressing objects: 100% (17/17), done.
remote: Total 150 (delta 8), reused 17 (delta 6), pack-reused 126
Receiving objects: 100% (150/150), 133.18 KiB | 297.00 KiB/s, done.
Resolving deltas: 100% (54/54), done.
namespace [dlgr_contrib]: myexperiments
experiment_name [testexperiment]: pushbutton
repo_name [myexperiments.pushbutton]:
package_name [myexperiments.pushbutton]:
experiment_class [TestExperiment]: PushButton
experiment_description [A simple Dallinger experiment.]: An experiment where the user␣
↪has to press a button
author [Jordan Suchow]: John Smith
author_github [suchow]: jsmith
author_email [suchow@berkeley.edu]: jsmith@smith.net
```

Once you are finished with those questions, Cookiecutter will create a directory structure containing a basic experiment which you can then modify to create your own. In the case of the example above, that directory will be named *myexperiments.pushbutton*.

When you clone the cookiecutter template from a GitHub repository, as we did here, cookiecutter saves the downloaded template inside your home directory, in the *.cookiecutter* sub-directory. The next time you run it, cookiecutter can use the stored template, or you can update it to the latest version. The default behavior is to ask you what you want to do. If you see a question like the following, just press <enter> to get the latest version:

```
You've downloaded /home/jsmith/.cookiecutters/cookiecutter-dallinger
before. Is it okay to delete and re-download it? [yes]:
```

If you answer *no*, cookiecutter will use the saved version. This can be useful if you are working off-line and need to start a project.

The template creates a runnable experiment, so you could change into the newly created directory right away and install your package:

```
$ cd myexperiments.pushbutton
$ pip install -e .
```

This command will allow you to run the experiment using Dallinger. You just need to change to the directory named for your experiment:

```
$ cd myexperiments/pushbutton
$ dallinger debug
```

This is enough to run the experiment, but to actually begin developing your experiment, you'll need to install the development requirements, like this:

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

Make sure you run this command from the initial directory created by Cookiecutter. In this case the directory is *myexperiments.pushbutton*.

3.2.1 The Experiment Package

There are several files and directories that are created with the *cookiecutter* command. Let's start with a general overview before going into each file in detail.

The directory structure of the package is the following:

```
- myexperiments.pushbutton
  - myexperiments
    - pushbutton
      - static
        - css
        - images
        - scripts
      - templates
  - tests
  - docs
    - source
      - _static
      - _templates
  - licenses
```

myexperiments.pushbutton

The main package directory contains files required to define the experiment as a Python package. Other than adding requirements and keeping the README up to date, you probably won't need to touch these files a lot after initial setup.

myexperiments.pushbutton/myexperiments

This is what is know in Python as a *namespace* directory. Its only purpose is marking itself as a container of several packages under a common name. The idea is that using a namespace, you can have many related but independent packages under one name, but you don't need to have all of them inside a single project.

myexperiments.pushbutton/myexperiments/pushbutton

Contains the code and resources (images, styles, scripts) for your experiment. This is where your main work will be performed.

myexperiments.pushbutton/tests

This is where the automated tests for your experiment go.

myexperiments.pushbutton/docs

The files stored here are the source files for your experiment's documentation. Dallinger uses Sphinx for documenting the project, and it's recommended that you use the same system for documenting your experiment.

myexperiments.pushbutton/licenses

This directory contains the experiment's license for distribution. Dallinger uses the MIT license, and it's encouraged, but not required, that you use the same.

3.2.2 Detailed Description for Support Files

Now that you are familiar with the main project structure, let's go over the details for the most important files in the package. Once you know what each file is for, you will be ready to begin developing your experiment. In this section we'll deal with the support files, which include tests, documentation and Python packaging files.

myexperiments.pushbutton/setup.py

This is a Python file that contains the package information, which is used by Python to setup the package, but also to publish it to the Python Package Repository (PYPI). Most of the questions you answered when creating the package with Cookiecutter are used here. As you develop your experiment, you might need to update the *version* variable defined here, which starts as "0.1.0". You may also wish to edit the *keywords* and *classifiers*, to help with your package's classification. Other than that, the file can be left untouched.

myexperiments.pushbutton/constraints.txt

This text file contains the minimal version requirements for some of the Python dependencies used by the experiment. Out of the box, this includes Dallinger and development support packages. If you add any dependencies to your experiment, it would be a good idea to enter the package version here, to avoid any surprises down the line.

myexperiments.pushbutton/requirements.txt

The Python packages required by your experiment should be listed here. Do not include versions, just the package name. Versions are handled in *constraints.txt*, discussed above. The file looks like this:

```
-c constraints.txt
dallinger
requests
```

The first line is what tells the installer which versions to use, and then the dependencies go below, one on each line by itself. The experiment template includes just two dependencies, *dallinger* and *requests*.

myexperiments.pushbutton/dev-requirements.txt

Similar to *requirements.txt* above, but contains the development dependencies. You should only change this if you add a development specific tool to your package. The format is the same as for the other requirements.

myexperiments.pushbutton/README.md

This is where the name and purpose of your experiment are explained, along with minimal installation instructions. More detailed documentation should go in the *docs* directory.

Other files in myexperiments.pushbutton

There are a few more files in the *myexperiments.pushbutton* directory. Here is a quick description of each:

- *.gitignore*. Used by *git* to keep track of which files to ignore when looking for changes in your project.
- *.travis.yml*. Travis is a continuous integration service, which can run your experiment's tests each time you push some changes. This is the configuration file where this is set up.
- *CHANGELOG.md*. This is where you should keep track of changes to your experiment. It is appended to *README.md* to form your experiment's basic description.
- *CONTRIBUTING.md*. Guidelines for collaborating with your project.
- *MANIFEST.in*. Used by the installer to determine which files and directories to include in uploads of your package.
- *setup.cfg*. Used by the installer to define metadata and settings for some development extensions.
- *tox.ini*. Sets up the testing environment.

myexperiments.pushbutton/test/test_pushbutton.py

This is a sample test suite for your experiment. It's intended only as a placeholder, and does not actually test anything as it is. See the documentation for pytest for information about setting up tests.

To run the tests as they are, and once you start adding your own, use the *pytest* command. Make sure you install dev-requirements.txt before running the tests, then enter this command from the directory that was created when you initially ran the cookiecutter command.

```
$ pytest
===================== test session starts ==============================
platform linux2 -- Python 2.7.15rc1, pytest-3.7.1, py-1.5.4, pluggy-0.7.1
rootdir: /home/jsmith/myexperiments.pushbutton, inifile:
collected 1 item

test/test_pushbutton.py .                                          [100%]

====================== 1 passed in 0.08 seconds ========================
```

myexperiments.pushbutton/docs/Makefile

The Sphinx documentation system uses this file to execute documentation building commands. Most of the time you will be building HTML documentation, for which you would use the following command:

```
$ make html
```

Make sure that you are in the *docs* directory and that the development requirements have been installed before running this.

The development requirements include an Sphinx plugin for checking the spelling of your documentation. This can be very useful:

```
$ make spelling
```

The *docs* directory also includes *makefile.bat*, which does the same tasks on Microsoft Windows systems.

myexperiments.pushbutton/docs/source/index.rst

This is where your main documentation will be written. Be sure to read the Sphinx documentation first, in particular the reStructuredText Primer.

myexperiments.pushbutton/docs/source/spelling_wordlist.txt

This file contains a list of words that you want the spell checker to recognize as valid. There might be some terms related to your experiment which are not common words but should not trigger a spelling error. Add them here.

Other files and directories in myexperiments.pushbutton/docs/source

There are a few more files in the documentation directory. Here's a brief explanation of each:

- *acknowledgments.rst*. A place for thanking any institutions or individuals that may have helped with the experiment. Can be used as an example of how to add new pages to your docs and link them to the table of contents (see the link in *index.rst*).
- *conf.py*. Python configuration for Sphinx. You don't need to touch this unless you start experimenting with plugins and documentation themes.
- *_static*. Static resources for the theme.
- *_templates*. Layout templates for the theme.

3.2.3 Experiment Code in Detail

As we reviewed in the previous section, there are lots of files which make your experiment distributable as a Python package. Of course, the most important part of the experiment template is the actual experiment code, which is where most of your work will take place. In this section, we describe each and every file in the experiment directory.

myexperiments.pushbutton/myexperiments/pushbutton/__init__.py

This is an empty file that marks your experiment's directory as a Python module. Though some developers add module initialization code here, it's OK if you keep it empty.

myexperiments.pushbutton/myexperiments/pushbutton/config.txt

The configuration file is used to pass parameters to the experiment to control its behavior. It's divided into four sections, which we'll briefly discuss next.

```
[Experiment]
mode = sandbox
auto_recruit = true
custom_variable = true
num_participants = 2
```

The first is the *Experiment* section. Here we define the experiment specific parameters. Most of these parameters are described in the *configuration section*.

The parameter *mode* sets the experiment mode, which can be one of debug (local testing), sandbox (MTurk sandbox), and live (MTurk). *auto_recruit* turns automatic participant recruitment on or off. *num_participants* sets the number of participants that will be recruited.

Of particular interest in this section is the *custom_variable* parameter. This is part of an example of how to add custom variables to an experiment. Here we set the value to *True*. See the experiment code below to understand how to define the variable.

```
[MTurk]
title = pushbutton
description = An experiment where the user has to press a button
keywords = Psychology
base_payment = 1.00
lifetime = 24
duration = 0.1
contact_email_on_error = jsmith@smith.net
browser_exclude_rule = MSIE, mobile, tablet
```

The next section is for the *MTurk* configuration parameters. Again, those are all discussed in the configuration section. Note that many of the parameter values above came directly from the Cookiecutter template questions.

```
[Database]
database_url = postgresql://postgres@localhost/dallinger
database_size = standard-0
```

The *Database* section contains just the database URL and size parameters. These should only be changed if you have your database in a non standard location.

```
[Server]
dyno_type = free
num_dynos_web = 1
num_dynos_worker = 1
host = 0.0.0.0
notification_url = None
clock_on = false
logfile = -
```

Finally, the *Server* section contains Heroku related parameters. Depending on the number of participants and size of the experiment, you might need to set the *dyno_type* and *num_dynos_web* parameters to something else, but be aware that most dyno types require a paid account. For more information about dyno types, please take a look at the heroku guide.

myexperiments.pushbutton/myexperiments/pushbutton/experiment.py

At last, we get to the experiment code. This is where most of your effort will take place. The *pushbutton* experiment is simple and the code is short, but it's important that you understand everything that happens here.

```
from dallinger.config import get_config
from dallinger.experiments import Experiment
from dallinger.networks import Empty
try:
        from bots import Bot
        Bot = Bot
except ImportError:
        pass
```

The first section of the code consists of some import statements to get the Dallinger framework parts ready.

After the Dallinger imports we try to import a bot from within the experiment directory. If none are defined, we simply skip this step. See the next section for more about bots.

```
config = get_config()


def extra_parameters():

        types = {
                'custom_variable': bool,
                'num_participants': int,
        }

        for key in types:
                config.register(key, types[key])
```

Next, we get the experiment configuration, which includes parsing the *config.txt* file shown above. The *get_config()* call also looks for an *extra_parameters* function, which is used to register the *custom_variable* and *num_participants* parameters discussed in the configuration section above.

```
class PushButton(Experiment):
        """Define the structure of the experiment."""
        num_participants = 1

        def __init__(self, session=None):
                """Call the same parent constructor, then call setup() if we have a␣
↪session.
                """
                super(PushButton, self).__init__(session)
                if session:
                        self.setup()

        def configure(self):
                super(PushButton, self).configure()
                self.experiment_repeats = 1
                self.custom_variable = config.get('custom_variable')
                self.num_participants = config.get('num_participants', 1)

        def create_network(self):
                """Return a new network."""
                return Empty(max_size=self.num_participants)
```

Finally, we have the *PushButton* class, which contains the main experiment code. It inherits its behavior from Dallinger's *Experiment* class, which we imported before. Since this is a very simple experiment, we don't have a lot of custom code here, other than setting up initial values for our custom parameters in the *configure* method.

If you had a class defined somewhere else representing some objects in your experiment, the place to initialize an instance would be the *__init__* method, which is called by Python on experiment initialization. The best place to do that would be the line after the *self.setup()* call, right after we are sure that we have a session.

Your experiment can do whatever you want, and use any dependencies that you need. The Python code is used mainly for backend tasks, while most interactivity depends on Javascript and HTML pages, which are discussed below.

myexperiments.pushbutton/myexperiments/pushbutton/bots.py

One of Dallinger's features is the ability to have automated experiment participants, or *bots*. These allow the experimenter to perform simulated runs of an experiment using hundreds or even thousands of participants easily. To support bots, an experiment needs to have a *bots.py* file that defines at least one bot. Our sample experiment has one, which if you recall was imported at the top of the experiment code.

There are two kinds of bots. The first, or regular bot, uses a webdriver to simulate all the browser interactions that a real human would have with the experiment. The other bot type is the high performance bot, which skips the browser simulation and interacts directly with the server.

```
import logging
import requests

from selenium.webdriver.common.by import By
from selenium.common.exceptions import TimeoutException
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC

from dallinger.bots import BotBase, HighPerformanceBotBase

logger = logging.getLogger(__file__)
```

The bot code first imports the bot base classes, along with some webdriver code for the regular bot, and the *requests* library, for the high performance bot.

```
class Bot(BotBase):
        """Bot tasks for experiment participation"""

        def participate(self):
                """Click the button."""
                try:
                        logger.info("Entering participate method")
                        submit = WebDriverWait(self.driver, 10).until(
                                EC.element_to_be_clickable((By.ID, 'submit-response
↪')))
                        submit.click()
                        return True
                except TimeoutException:
                        return False
```

The *Bot* class inherits from *BotBase*. A bot needs to have a *participate* method, which simulates a subject's participation. For this experiment, we simply wait until a clickable button with the id *submit-response* is loaded, and then we click it. That's it. Other experiments will of course require more complex interactions, but this is the gist of it.

To write a bot you need to know fairly well what your experiment does, plus a good command of the Selenium webdriver API, which thankfully has extensive documentation.

```
class HighPerformanceBot(HighPerformanceBotBase):
        """Bot for experiment participation with direct server interaction"""

        def participate(self):
                """Click the button."""
                self.log('Bot player participating.')
                node_id = None
                while True:
                        # create node
                        url = "{host}/node/{self.participant_id}".format(
                                host=self.host,
                                self=self
                        )
                        result = requests.post(url)
                        if result.status_code == 500 or result.json()['status'] ==
↪'error':
                                self.stochastic_sleep()
```

(continues on next page)

(continued from previous page)

```
                                continue

                        node_id = result.json.get('node', {}).get('id')

                while node_id:
                        # add info
                        url = "{host}/info/{node_id}".format(
                                host=self.host,
                                node_id=node_id
                        )
                        result = requests.post(url, data={"contents": "Submitted",
↪    "info_type": "Info"})
                        if result.status_code == 500 or result.json()['status'] ==
↪'error':
                                self.stochastic_sleep()
                                continue

                        return
```

The high performance bot works very differently. It uses the *requests* library to directly post URLs to the server, passing expected values as request parameters. This works much faster than simulating a browser, thus allowing for more bots to participate in an experiment using fewer resources.

myexperiments.pushbutton/myexperiments/pushbutton/templates/layout.html

This template defines the layout to be used by the all the experiment pages.

```
{% extends "base/layout.html" %}

{% block title -%}
        Psychology Experiment
{%- endblock %}

{% block libs %}
        <script src="/static/scripts/store+json2.min.js" type="text/javascript"> </
↪script>
        {{ super() }}
        <script src="/static/scripts/experiment.js" type="text/javascript"> </script>
{% endblock %}
```

As far as layout goes, this template doesn't do much else than setting the title, but the important part to notice here is that we include the experiment's Javascript files. Here is where you can add any Javascript libraries that you need to use for your experiment.

myexperiments.pushbutton/myexperiments/pushbutton/templates/ad.html

The ad template is where the experiment is presented to a potential user. In this experiment, we simply use the default ad template.

myexperiments.pushbutton/myexperiments/pushbutton/templates/consent.html

The consent template is where the user accepts (or not) to participate in the experiment.

```
{% extends "base/consent.html" %}


{% block consent_button %}
    <!-- custom consent button/action -->
    <button type="button" id="consent" class="btn btn-primary btn-lg">I agree</button>
{% endblock %}
```

In our experiment, we extend the original consent template, and use the *consent_button* block to add a custom button for expressing consent.

myexperiments.pushbutton/myexperiments/pushbutton/templates/instructions.html

Next come the instructions for the experiment. For our instructions template, notice how we don't extend an "instructions" template, but rather the more generic *layout* template, because instructions are much more particular to the experiment objectives and interaction mechanisms.

```
{% extends "layout.html" %}

{% block body %}
    <div class="main_div">
        <hr>

        <p>In this experiment, you will click a button.</p>

        <hr>

        <div>
            <div class="row">
                <div class="col-xs-10"></div>
                <div class="col-xs-2">
                    <button type="button" class="btn btn-success btn-lg"
                        onClick="dallinger.allowExit(); dallinger.goToPage('exp');">
                    Begin</button>
                </div>
            </div>
        </div>
</div>
{% endblock %}
```

The instructions are the last stop before beginning the actual experiment, so we have to direct the user to the experiment page. This is done by using the *dallinger.goToPage* method in the button's *onClick* handler. Notice the call to *dallinger.allowExit* right before the page change. This is needed because Dallinger is designed to prevent users from accidentally leaving the experiment by closing the browser window before it's finished. The *allowExit* call means that in this case it's fine to leave the page, since we are going to the experiment page.

```
{% block scripts %}
        <script>
                dallinger.createParticipant();
        </script>
{% endblock %}
```

A Dallinger experiment requires a participant to be created before beginning. Sometimes this is done conditionally or at a specific event in the experiment flow. Since this experiment just requires pushing the button, we create the participant on page load by calling the *dallinger.createParticipant* method.

myexperiments.pushbutton/myexperiments/pushbutton/templates/exp.html

The *exp* template is where the main experiment action happens. In this case, there's not a lot of action, though.

```
{% extends "layout.html" %}

{% block body %}
        <div class="main_div">
                <div id="stimulus">
                        <h1>Click the button</h1>
                        <button id="submit-response" type="button" class="btn btn-
↪primary">Submit</button>
                </div>
        </div>
{% endblock %}

{% block scripts %}
        <script>
                create_agent();
        </script>
{% endblock %}
```

We fill the *body* block with a simple * * that includes a heading and the button to press. Notice how the *submit-response* id corresponds to the one that the bot code, discussed above, uses to find the button in the page.

The template doesn't include any mechanism for sending the form to the experiment server. This is done separately by the experiment's Javascript code, described below.

myexperiments.pushbutton/myexperiments/pushbutton/templates/questionnaire.html

Dallinger experiments conclude with the user filling in a questionnaire about the completed experiment. It's possible to add custom questions to this questionnaire, which our *questionnaire* template does:

```
{% extends "base/questionnaire.html" %}

{% block questions %}
<!-- additional custom questions -->
<div class="row question">
    <div class="col-md-8">
        On a scale of 1-10 (where 10 is the most engaged), please rate the button:
    </div>
    <div class="col-md-4">
        <select id="button-quality" name="button-quality">
            <option value="10">10 - Very good button</option>
            <option value="9">9</option>
            <option value="8">8</option>
            <option value="7">7</option>
            <option value="6">6</option>
            <option value="5" SELECTED>5 - Moderately good button</option>
            <option value="4">4</option>
            <option value="3">3</option>
            <option value="2">2</option>
            <option value="1">1 - Terrible button</option>
        </select>
    </div>
</div>
{% endblock %}
```

In this case we add a simple select question, but you can use any Javascript form tools to add more complex question UI elements.

myexperiments.pushbutton/myexperiments/pushbutton/static/scripts/experiment.js

The final piece in the puzzle is the *experiment.js* file, which contains the Javascript code for the experiment. Like the Python code, this is a simple example, but it can be as complex as you need, and use any Javascript libraries that you wish to include in your experiment.

```
var my_node_id;

$(document).ready(function() {

      // do not allow user to close or reload
      dallinger.preventExit = true;

      // Print the consent form.
      $("#print-consent").click(function() {
            window.print();
      });

      // Consent to the experiment.
      $("#consent").click(function() {
            store.set("recruiter", dallinger.getUrlParameter("recruiter"));
            store.set("hit_id", dallinger.getUrlParameter("hit_id"));
            store.set("worker_id", dallinger.getUrlParameter("worker_id"));
            store.set("assignment_id", dallinger.getUrlParameter("assignment_id"));
            store.set("mode", dallinger.getUrlParameter("mode"));

            dallinger.allowExit();
            window.location.href = '/instructions';
      });

      // Consent to the experiment.
      $("#no-consent").click(function() {
            dallinger.allowExit();
            window.close();
      });
```

The first few methods deal with the consent form. Basically, if the user consents, we go to the instructions page, and if not, the window is closed and the experiment ends. As you can see, there's also a button to print the consent page.

```
  $("#submit-response").click(function() {
        $("#submit-response").addClass('disabled');
        $("#submit-response").html('Sending...');
        dallinger.createInfo(my_node_id, {contents: "Submitted", info_type: "Info"})
        .done(function (resp) {
          dallinger.allowExit();
          dallinger.goToPage('questionnaire');
        })
        .fail(function (rejection) {
          dallinger.error(rejection);
        });
  });
});

// Create the agent.
```

(continues on next page)

(continued from previous page)

```
var create_agent = function() {
  // Setup participant and get node id
  $("#submit-response").addClass('disabled');
  dallinger.createAgent()
  .done(function (resp) {
        my_node_id = resp.node.id;
        $("#submit-response").removeClass('disabled');
  })
  .fail(function (rejection) {
        dallinger.error(rejection);
  });
};
```

For the experiment page, when the *submit-response* button is clicked, we create an *Info* to record the submission and send the user to the questionnaire page, which completes the experiment. If there was some sort of error, we display an error page.

The *create_agent* function is called when the experiment page loads, to make sure the button is not enabled until Dallinger is fully setup for the experiment.

3.2.4 Extending the Template

Understanding the experiment files is one thing, but how do we go from template to new experiment? In this section, we'll extend the cookiecutter template to create a full experiment. This way, the most common points of extension and user requirements will be discussed, thus making it easier to think about creating original experiments.

The Bartlett 1932 Experiment

Sir Frederic Charles Bartlett was a British psychologist and the first professor of experimental psychology at the University of Cambridge. His most important work was *Remembering* (1932) which consisted of experimental studies on remembering, imaging, and perceiving.

For our work in this section, we will take one of Bartlett's experiments and turn it into a Dallinger experiment. Our experiment will be simple: participants will be given a text, and then they will have to recreate that text word for word as best as they can.

Starting the Cookiecutter template

First, we need to create our experiment template, using cookiecutter. If you recall, the initial section of this tutorial showed how to do this:

```
cookiecutter https://github.com/Dallinger/cookiecutter-dallinger.git
```

Make sure to answer "bartlett1932" to the *experiment_name* question. You can use the default values for the rest.

Setting Up the Network

The first thing to decide is how participants will interact with the experiment and with each other. Some experiments might just need participants to individually interact with the experiment, while others may require groups of people communicating with each other as well.

Dallinger organizes all experiment participants in *networks*. A network can include various kinds of nodes. Most nodes are associated with participants, but there are other kinds of nodes, like sources, which are used to transmit

information. Nodes are connected to other nodes in different ways, depending on the type of network that is defined for the experiment.

Sources are an important kind of node, because many times the information (stimulus) required for conducting the experiment will come from one. A source can only transmit information, never receive it. For this experiment, we will use a source to send the text that the user must read and recreate.

Dallinger supports various kinds of networks out of the box, and you can create your own too. The most common networks are:

- *Chain*. A network where each new node is connected to the most recently added node. The top node of the chain can be a source.
- *FullyConnected*. A network in which each node is connected to every other node. This includes sources.
- *Empty*. A network where every node is isolated from the rest. It can include a source, in which case it will be connected to the nodes.

For this experiment, we will use a chain network. The top node will be a source, so that we can use different texts on each run, and send them to each newly connected participant. In fact, most of the Python code for the experiment will deal with network management. Let's get started. All the code in this section goes into the *experiment.py* file generated by the cookiecutter:

```
from dallinger.experiment import Experiment
from dallinger.networks import Chain

from . import models


class Bartlett1932(Experiment):
        """An experiment from Bartlett's Remembering."""

        def __init__(self, session=None):
                super(Bartlett1932, self).__init__(session)
                self.models = models
                self.experiment_repeats = 1
                self.initial_recruitment_size = 1
                if session:
                        self.setup()
```

First, we import the *Experiment* class, which we will extend for our Bartlett experiment. Next, we import *Chain*, which is the class for our chosen network. After that, we import our models, which will be discussed in the next section.

Following this, we define the experiment class *Bartlett1932*, subclassing Dallinger's Experiment class. The *__init__* method calls the Experiment initialization first, then does common setup work. For other experiments, you might need to change the number of *experiment_repeats* (how many times the experiment is run) and the *initial_recruitment_size* (how many participants are going to be recruited initially). In this case, we set both to 1.

Note that as part of the initialization, we take the models we imported above and assign them to the created instance.

The last line calls *self.setup*, which is defined as follows:

```
def setup(self):
        if not self.networks():
                super(Bartlett1932, self).setup()
                for net in self.networks():
                        self.models.WarOfTheGhostsSource(network=net)
```

The *self.networks()* call at the top, will get all the networks defined for this experiment. When it is first run, this will return an empty list, in which case we will call the Experiment setup. After this call, the network will be defined.

Once we have a network, we add our source to it as the first node. This will be discussed in more detail in the next section. Just take note that the source constructor takes the current network as a parameter.

The network setup code will call the *create_network* method in our experiment:

```
def create_network(self):
        return Chain(max_size=5)
```

The only thing this method does is create a chain network, with a maximum size of 5.

Our experiment will also need to transmit the source information when a new participant joins. That is achieved using the *add_node_to_network* method. You can add this method to any experiment where you need to do something to newly added nodes:

```
def add_node_to_network(self, node, network):
        network.add_node(node)
        parents = node.neighbors(direction="from")
        if len(parents):
                parent = parents[0]
                parent.transmit()
        node.receive()
```

The method will get as parameters the new node and the network to which it is being added. The first thing to do is not forgetting to add the node to the network. Once that is safely behind, we get the node's parents using the *neighbors* method. The parents are any nodes that the current node is connecting from, so we use the *direction="from"* parameter in the call.

If there are any parents (and in this case, there will be). We get the first one, and call its *transmit* method. Finally, the node's *receive* method is called, to receive the transmission.

Recruitment

Closely connected to the experiment network structure, recruitment is the method by which we get experiment participants. For this, Dallinger uses a *Recruiter* subclass. Among other things, a recruiter is responsible for opening recruitment, closing recruitment, and recruiting new participants for the experiment.

As you might already know, Dallinger works closely with Amazon's Mechanical Turk, which for the purposes of our experiments, you can think of as a crowdsourcing marketplace for experiment participants. The default Dallinger recruiter knows how to make experiments available for MTurk users, and how to recruit those users into an experiment.

An experiment's *recruit* method communicates with the recruiter to get the participants into its network:

```
def recruit(self):
        if self.networks(full=False):
                self.recruiter.recruit(n=1)
        else:
                self.recruiter.close_recruitment()
```

In our case, we only need to get participants one by one. We first check if the experiment networks are already full, in which case we skip the recruitment call (*full=False* will only return non-full networks). If there is space, we call the *recruit* method of the recruiter. Otherwise, we call *close_recruiment*, to end recruitment for this run.

It is important to note that recruitment will only start automatically if the experiment is configured to do so, bu setting *auto_recruit* to true in the *config.txt* file. The template that we created already has this variable set up like this.

Sources and Models

Earlier, we mentioned that we needed a source of information that could send new participants the text to be read and recalled for our experiment. In fact, we assumed that this already existed, and proceeded to add the *from . import models* line in our code in the previous section.

To make this work, we need to create a *models.py* file inside our experiment, and add this code:

```
from dallinger.nodes import Source
import random


class WarOfTheGhostsSource(Source):

        __mapper_args__ = {
                "polymorphic_identity": "war_of_the_ghosts_source"
        }

        def _contents(self):
                stories = [
                        "ghosts.md",
                        "cricket.md",
                        "moochi.md",
                        "outwit.md",
                        "raid.md",
                        "species.md",
                        "tennis.md",
                        "vagabond.md"
                ]
                story = random.choice(stories)
                with open("static/stimuli/{}".format(story), "r") as f:
                        return f.read()
```

Recall that Dallinger uses a database to store experiment data. Most of Dallinger's main objects, including *Source*, are defined as SQLAlchemy models. To define a source, the only requirement is that it provide a *_contents* method, which should return the source information.

For our experiment, we will add a static/stimuli directory where we'll store our story text files. In the code above, you can see that we explicitly name eight stories. If you are following along and typing the code as we go, you can get those files from the dallinger repository. You can also add any text files that you have, and simply change the *stories* list above to use their names.

Our *_contents* method just selects one of these files randomly and returns its full content (*f.read()* does that).

When a node's *transmit* method is called, dallinger looks for its *_what* method and calls it to get the information to be transmitted. In the case of a source, this in turn calls the source's *create_information* method, which finally calls the *_contents* method and returns the result. The chain of calls is like this:

```
transmit() -> _what() -> create_information() -> _contents().
```

This might seem like a roundabout way to get the information, but it allows us to override any of the steps and return different information types or other modifications. Much of Dallinger is designed in this way, making it easy to create compatible, but perhaps completely different versions of its main constructs.

The Experiment Code

Now that we are done setting up the experiment's infrastructure, we can write the code that will drive the actual experiment. Dallinger is very flexible, and you can design really complicated experiments for it. Some will require

pretty heavy backend code, and probably a handful of dependencies. For this kind of advanced experiments, a lot of the code could be in Python.

Dallinger also includes a Redis-based chat backend, which can be used to relay messages from experiment participants to the application and each other. All you have to do to enable this is to define a *channel* class variable with a string prefix for your experiment, and then you can use the experiment's *send* method to handle messages. Using this backend, you can easily create chat-enabled experiments, and even graphical UIs that can communicate user actions using channel messages.

For this tutorial, however, we are keeping it simple, and thus will not require any other Python code for it. We already have a source for the texts defined, the network is set up, and recruitment is enabled, so all we need to get the Bartlett experiment going is a simple Javascript UI.

The code that we will walk through will be saved in our *experiment.js* file:

```
var my_node_id

// Consent to the experiment.
$(document).ready(function() {

  dallinger.preventExit = true;
```

The *experiment.js* file will be executed on page load (see below for the template walk through), so we use the JQuery *$(document).ready* hook to run our code.

The very first thing we do is setting *dallinger.preventExit* to True, which will prevent experiment participants from closing the window or reloading the page. This is to avoid the experiment being interrupted and the leaving the participant in an inconsistent state.

Next, we define a few functions that will be called from the various experiment templates. This are functions that are more or less required for all experiments:

```
$("#print-consent").click(function() {
  window.print();
});

$("#consent").click(function() {
  store.set("recruiter", dallinger.getUrlParameter("recruiter"));
  store.set("hit_id", dallinger.getUrlParameter("hit_id"));
  store.set("worker_id", dallinger.getUrlParameter("worker_id"));
  store.set("assignment_id", dallinger.getUrlParameter("assignment_id"));
  store.set("mode", dallinger.getUrlParameter("mode"));

  dallinger.allowExit();
  window.location.href = '/instructions';
});

$("#no-consent").click(function() {
  dallinger.allowExit();
  window.close();
});

$("#go-to-experiment").click(function() {
  dallinger.allowExit();
  window.location.href = '/exp';
});
```

Mostly, these functions are related to the user expressing consent to participate in the experiment, and getting to the real experiment page.

The consent page will have a *print-consent* button, which will simply call the browser's print function for printing the page.

Next, if the user clicks *consent*, and thus agrees to participate in the experiment, we store the experiment and participant information from the URL, so that we can retrieve it later. The *store.set* calls use a local storage library to keep the values handy.

Once we have saved the data, we enable exiting the window, and direct the user to the instructions page.

If the user clicked on the *no-consent* button instead, it means that they did not consent to participate in the experiment. In that case, we enable exiting, and simply close the window. We are done.

If the user got as far as the instructions page. They will see a button that will sent them to the experiment when clicked. This is the *go-to-experiment* button, which again enables page exiting and sets the location to the experiment page.

We now come to our experiment specific code. The plan for our UI is like this: we will have a page displaying the text, and a text area widget to write the text that the user can recall after reading it. We will have both in a single page, but only show one at a time. When the page loads, the user will see the text, followed by a *finish-reading* button:

```
$("#finish-reading").click(function() {
  $("#stimulus").hide();
  $("#response-form").show();
  $("#submit-response").removeClass('disabled');
  $("#submit-response").html('Submit');
});
```

When the user finishes reading, and clicks on the button, we hide the text and show the response form. This form will have a *submit-response* button, which we enable. Finally, the text of the button is changed to read "Submit".

This, and all the Javascript code in this section, uses the JQuery Javascript library, so check the JQuery documentation if you need more information.

Now for the *submit-response* button code:

```
  $("#submit-response").click(function() {
    $("#submit-response").addClass('disabled');
    $("#submit-response").html('Sending...');

    var response = $("#reproduction").val();

    $("#reproduction").val("");

    dallinger.createInfo(my_node_id, {
      contents: response,
      info_type: 'Info'
    }).done(function (resp) {
      create_agent();
    });
  });

});
```

When the user is done typing the text and clicks on the *submit-response* button, we disable the button and set the text to "Sending...". Next, we get the typed text from the *reproduction* text area, and wipe out the text.

The *dallinger.createInfo* function calls the Dallinger Python backend, which creates a Dallinger Info object associated with the current participant. This info will store the recalled text. If the info creation succeeds, the *create_agent* function will be called:

```
var create_agent = function() {
  $('#finish-reading').prop('disabled', true);
  dallinger.createAgent()
  .done(function (resp) {
    $('#finish-reading').prop('disabled', false);
    my_node_id = resp.node.id;
    get_info();
  })
  .fail(function (rejection) {
    if (rejection.status === 403) {
      dallinger.allowExit();
      dallinger.goToPage('questionnaire');
    } else {
      dallinger.error(rejection);
    }
  });
};
```

The *create_agent* function is called twice in this experiment. The first time when the experiment page loads, and the second time when the *submit-response* button is clicked.

Both times, it first disables the *finish-reading* button before calling the *dallinger.createAgent* function. This function calls the Python backend, to create an experiment node for the current participant.

The first time, this call will succeed, since there is no node defined for this participant. In that case, we enable the *finish-reading* button and save the returned node's id in the *my_node_id* global variable defined at the start of our Javascript code. Finally, we call the *get_info* function defined below.

The second time that *create_agent* is called, is when the text is submitted by the user. When that happens, the underlying *createAgent* call will fail, and return a rejection status of "403". The code above checks for that status, and if it finds it, that's the signal for us to finish the experiment and send the user to the Dallinger questionnaire page. If the rejection status is not "403", that means something unexpected happened, and we need to raise a Dallinger error, effectively ending the experiment.

Now let's discuss the *get_info* function mentioned above, which is called when the experiment first calls the *create_agent* function:

```
var get_info = function() {
  dallinger.getReceivedInfos(my_node_id)
  .done(function (resp) {
    var story = resp.infos[0].contents;
    $("#story").html(story);
    $("#stimulus").show();
    $("#response-form").hide();
    $("#finish-reading").show();
  })
  .fail(function (rejection) {
    console.log(rejection);
    $('body').html(rejection.html);
  });
};
```

Remember that in the Python code above, in the *add_node_to_network* method, we looked for the participant's parent, and then called its *transmit* method, followed by the node's own *receive* method. This transmits the parent node's info to the new node. The Javascript *get_info* function tries to get that info by calling *dallinger.getReceivedInfos* with the node id that we saved after successfully calling *dallinger.createAgent*.

For the first participant, this info will contain the text generated by the source we defined above. That is, the full text of one of the stimulus stories, chosen at random. The second participant will get the text as recalled by the first

participant, and so on. The last participant will likely have a much different text to work with than the first.

Once *get_info* gets the text, it puts it in the *story* textarea, and shows it to the user, by displaying the *stimulus* div. Then it makes sure the *response-form* is not visible, and shows the *finish-reading* button.

If anything fails, we log the rejection message to the console, and show the error to the user.

The experiment templates

The experiment uses regular dallinger templates for the ad page and consent form. It does define its own layout, as an example of how to include dependencies. Here's the full *layout.html* template:

```
{% extends "base/layout.html" %}

{% block title -%}
        Bartlett 1932 Experiment
{%- endblock %}

{% block libs %}
        <script src="/static/scripts/store+json2.min.js" type="text/javascript"> </
↪script>
        {{ super() }}
        <script src="/static/scripts/experiment.js" type="text/javascript"> </script>
{% endblock %}
```

The only important part if the layout template is the *libs* block. Here you can add any Javascript dependencies that your experiment needs. Just place them in the experiment's static directory, and they will be available for linking from this page.

Note how we load everything else before the *experiment.js* file that contains our experiment code (The *super* call brings up any dependencies defined in the base layout).

Next comes the *instructions.html* template:

```
{% extends "layout.html" %}

{% block body %}
        <div class="main_div">
                <h1>Instructions</h1>

                <hr>

                <p>In this experiment, you will read a passage of text. Your job is␣
↪to remember the passage as well as you can, because you will be asked some␣
↪questions about it afterwards.</p>

                <hr>

                <div>
                        <div class="row">
                                <div class="col-xs-10"></div>
                                <div class="col-xs-2">
                                        <button id="go-to-experiment" type="button"␣
↪class="btn btn-success btn-lg">
                                        Begin</button>
                                </div>
                        </div>
                </div>
```

(continues on next page)

(continued from previous page)

```
        </div>
{% endblock %}

{% block scripts %}
        <script>
                dallinger.createParticipant();
        </script>
{% endblock %}
```

Here is where you will put specific instructions for your experiment. Since we get here right after consenting to participate in the experiment, it's also a good place to create the experiment participant node. This is done by calling the *dallinger.createParticipant* function upon page load.

Notice also that after the instructions we add the *go-to-experiment* button that will send the user to the experiment page, where the main UI for our experiment is defined:

```
{% extends "layout.html" %}

{% block body %}
        <div class="main_div">
                <div id="stimulus">
                        <h1>Read the following text:</h1>
                        <div><blockquote id="story"><p>&lt;&lt; loading &gt;&gt;</p></
↪blockquote></div>
                        <button id="finish-reading" type="button" class="btn btn-
↪primary">I'm done reading.</button>
                </div>

                <div id="response-form" style="display:none;">
                        <h1>Now reproduce the passage, verbatim:</h1>
                        <p><b>Note:</b> Your task is to recreate the text, word for␣
↪word, to the best of your ability.<p>
                        <textarea id="reproduction" class="form-control" rows="10"></
↪textarea>
                        <p></p>
                        <button id="submit-response" type="button" class="btn btn-
↪primary">Submit response.</button>
                </div>
        </div>
{% endblock %}

{% block scripts %}
        <script>
                create_agent();
        </script>
{% endblock %}
```

The *exp.html* template is the one that connects with the experiment code we described above. There is *stimulus* div where the story text will be displayed, inside the *story* blockquote tag. There is also the *finish-reading* button. which will be disabled until we get the story text from the source.

After that, we have the *response-form* div, which contains the *reproduction* textarea where the user will type the text. Note that the div's *display* attribute is set to *none*, so the form will not be visible at page load time. Finally, the *submit-response* button will take care of initiating the submission process.

At the bottom of the template, inside a script tag, is the *create_agent* call that will get the source info and enable the stimulus area.

Dallinger's experiment server uses *Flask*, which in turn uses the *Jinja2* templating engine. Consult the Flask documentation for more information about how the templates work.

Creating a Participant Bot

We now have a complete experiment, but there's one more interesting thing that we will cover in this tutorial. Dallinger allows the possibility of using *bot* participants. That is, automated participants that know how to do an experiment's tasks. It is even possible to mix human and bot participants.

For this experiment, we will add a bot that can navigate through the experiment and submit the response at the end. Bots have perfect memories, but we could spend a lot of effort trying to make them act as forgetful humans. We will not do so, since it is out of the scope of this tutorial.

A basic bot gets the same exact pages that a human would, and needs to use a *webdriver* to go from page to page. Dallinger bots use the *selenium* webdrivers, which need a few imports to begin (add this to *experiment.py*):

```
from selenium.webdriver.common.by import By
from selenium.common.exceptions import TimeoutException
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC

from dallinger.bots import BotBase
```

After the selenium imports, we import *BotBase* from dallinger, which our bot will subclass. The only required method for a bot is the *participate* method, which is called by the bot framework when the bot is recruited.

Here is the bot code:

```
class Bot(BotBase):

        def participate(self):
                try:
                        ready = WebDriverWait(self.driver, 10).until(
                                EC.element_to_be_clickable((By.ID, 'finish-reading')))
                        stimulus = self.driver.find_element_by_id('stimulus')
                        story = stimulus.find_element_by_id('story')
                        story_text = story.text
                        ready.click()
                        submit = WebDriverWait(self.driver, 10).until(
                                EC.element_to_be_clickable((By.ID, 'submit-response
↪')))
                        textarea = WebDriverWait(self.driver, 10).until(
                                EC.element_to_be_clickable((By.ID, 'reproduction')))
                        textarea.clear()
                        text = self.transform_text(story_text)
                        textarea.send_keys(text)
                        submit.click()
                        return True
                except TimeoutException:
                        return False

        def transform_text(self, text):
                return "Some transformation...and %s" % text
```

The participate method needs to return *True* if the participation was successful, and *False* otherwise. Since the webdriver could fail at getting the correct page in time, we wrap the whole participation sequence in a *try* clause. Combined with the *WebDiverWait* method of the webdriver, this will raise a *TimeoutException* if anything fails and the bot can't proceed after the specified timeout. In this example, we use 10 seconds for the timeout.

The rest is simple: the bot waits until it can see the *finish-reading* button and assigns it to the *ready* variable. It then finds the *stimulus* div and the *story* inside of that, and extracts the story text. Once it gets the text, the bot "clicks" the ready button.

The bot next waits for the *submit-response* div to be active, and the *reproduction* textarea activated. Just to do something with it for this example, the bot calls the *transform_text* method, which just adds a few words to the story text. It then sends the text to the textarea element, using its *send_keys* method. After that, the task is complete, and the form is submitted (*submit.click*). Finally, the bot returns *True* to signal success.

3.2.5 Developing Your Own Experiment

Now that you are more familiar with the full experiment contents, and have seen how to go from template to finished experiment, you are in position to begin extending the code to create your first experiment. Dallinger has an extensive API, so you will probably need to refer to the documentation constantly as you go along. Here are some resources within the documentation that should prove to be very useful while you develop your experiment further:

- *The Web API*.
- *The Javascript API*.
- *The Database API*.
- *The Experiment Class*.
- *Writing Bots*.

3.3 Dallinger with Docker

With the release of Dallinger version 5.0.0, we have created a Python script that uses docker-compose to provide an automated installation and configuration of Dallinger to run experiments.

The code and detailed instructions can be found in this github repository.

Please note that we consider this to be a working yet experimental method of running Dallinger. It adds an extra level of complexity which can potentially get in the way when trying to create and debug a new experiment as debugging is more diffcult than when using Dallinger natively or in a virtual machine. Having said that, there are can be certain advantages to this method, since Docker can install everything required to run Dallinger quickly in comparison to installing all the requirements yourself, and on platforms such as Microsoft Windows where a native installation is not possible.

3.4 Running the tests

If you push a commit to a branch in the Dallinger organization on GitHub, or open a pull request from your own fork, Dallinger's automated code tests will be run on Travis.

Current build status:

The tests include:

- Making sure that a source distribution of the Python package can be created.
- Running flake8 to make sure Python code conforms to the PEP 8 style guide.
- Running the tests for the Python code using pytest and making sure they pass in Python 2.7 and 3.6.
- Making sure that code coverage for the Python code is above the desired threshold.

- Making sure the docs build without error.

If you see ImportErrors related to demo packages, this most likely means you have not installed the `dlgr.demos` sub-package. See the *Dallinger development installation instructions* for details.

3.4.1 Amazon Mechanical Turk Integration Tests

You can also run all these tests locally, with some additional requirements:

- The Amazon Web Services credentials set in .dallingerconfig must correspond to a valid MTurk Sandbox Requester account.
- Some tests require access to an MTurk Sandbox Worker account, so you should create this account (probably using the same AWS account as above).
- The Worker ID from the Worker account (visible on the dashboard) needs to be set in `tests/config.py`, which should be created by making a copy of `tests/config.py.in` before setting the value. `tests/config.py` is excluded from version control, so your Id will not be pushed to a remote repository.

3.4.2 Commands

You can run all tests locally, simply by running:

```
tox
```

To run just the Python tests:

```
pytest
```

To run the Python tests excluding those that interact with Amazon Mechanical Turk, run:

```
pytest -m "not mturk"
```

To run all tests except those that require a MTurk Worker ID, run:

```
pytest -m "not mturkworker"
```

To run the complete, comprehensive suite of tests which interact Mechanical Turk, add the `mturkfull` option when running the tests:

```
pytest --mturkfull
```

To build documentation:

```
tox -e docs
```

To run flake8:

```
flake8
```

3.5 The Experiment Class

Experiments are designed in Dallinger by creating a custom subclass of the base Experiment class. The code for the Experiment class is in experiments.py. Unlike the *other classes*, each experiment involves only a single Experiment

object and it is not stored as an entry in a corresponding table, rather each Experiment is a set of instructions that tell the server what to do with the database when the server receives requests from outside.

class dallinger.experiment.**Experiment**(*session=None*)

Define the structure of an experiment.

verbose

Boolean, determines whether the experiment logs output when running. Default is True.

task

String, the name of the experiment. Default is "Experiment title".

session

session, the experiment's connection to the database.

practice_repeats

int, the number of practice networks (see *role*). Default is 0.

experiment_repeats

int, the number of non practice networks (see *role*). Default is 0.

recruiter

initial_recruitment_size

int, the number of participants requested when the experiment first starts. Default is 1.

known_classes

dictionary, the classes Dallinger can make in response to front-end requests. Experiments can add new classes to this dictionary.

public_properties

__init__(*session=None*)

Create the experiment class. Sets the default value of attributes.

add_node_to_network(*node*, *network*)

Add a node to a network.

This passes *node* to `add_node()`.

assignment_abandoned(*participant*)

What to do if a participant abandons the hit.

This runs when a notification from AWS is received indicating that *participant* has run out of time. Calls `fail_participant()`.

assignment_reassigned(*participant*)

What to do if the assignment assigned to a participant is reassigned to another participant while the first participant is still working.

This runs when a participant is created with the same assignment_id as another participant if the earlier participant still has the status "working". Calls `fail_participant()`.

assignment_returned(*participant*)

What to do if a participant returns the hit.

This runs when a notification from AWS is received indicating that *participant* has returned the experiment assignment. Calls `fail_participant()`.

attention_check(*participant*)

Check if participant performed adequately.

Return a boolean value indicating whether the *participant*'s data is acceptable. This is mean to check the participant's data to determine that they paid attention. This check will run once the *participant* completes the experiment. By default performs no checks and returns True. See also `data_check()`.

attention_check_failed(*participant*)
What to do if a participant fails the attention check.

Runs when *participant* has failed the `attention_check()`. By default calls `fail_participant()`.

bonus(*participant*)
The bonus to be awarded to the given participant.

Return the value of the bonus to be paid to *participant*. By default returns 0.

bonus_reason()
The reason offered to the participant for giving the bonus.

Return a string that will be included in an email sent to the *participant* receiving a bonus. By default it is "Thank you for participating! Here is your bonus."

collect(*app_id*, *exp_config=None*, *bot=False*, ***kwargs*)
Collect data for the provided experiment id.

The `app_id` parameter must be a valid UUID. If an existing data file is found for the UUID it will be returned, otherwise - if the UUID is not already registered - the experiment will be run and data collected.

See *run()* method for other parameters.

create_network()
Return a new network.

create_node(*participant*, *network*)
Create a node for a participant.

data_check(*participant*)
Check that the data are acceptable.

Return a boolean value indicating whether the *participant*'s data is acceptable. This is meant to check for missing or invalid data. This check will be run once the *participant* completes the experiment. By default performs no checks and returns True. See also, `attention_check()`.

data_check_failed(*participant*)
What to do if a participant fails the data check.

Runs when *participant* has failed `data_check()`. By default calls `fail_participant()`.

events_for_replay(*session=None*, *target=None*)
Returns an ordered list of "events" for replaying. Experiments may override this method to provide custom replay logic. The "events" returned by this method will be passed to *replay_event()*. The default implementation simply returns all *Info* objects in the order they were created.

fail_participant(*participant*)
Fail all the nodes of a participant.

get_network_for_participant(*participant*)
Find a network for a participant.

If no networks are available, None will be returned. By default participants can participate only once in each network and participants first complete networks with *role="practice"* before doing all other networks in a random order.

info_get_request(*node*, *infos*)
Run when a request to get infos is complete.

info_post_request(*node*, *info*)
Run when a request to create an info is complete.

is_complete()
Method for custom determination of experiment completion. Experiments should override this to provide custom experiment completion logic. Returns *None* to use the experiment server default logic, otherwise should return *True* or *False*.

is_overrecruited(*waiting_count*)
Returns True if the number of people waiting is in excess of the total number expected, indicating that this and subsequent users should skip the experiment. A quorum value of 0 means we don't limit recruitment, and always return False.

log(*text*, *key='?????'*, *force=False*)
Print a string to the logs.

log_summary()
Log a summary of all the participants' status codes.

classmethod make_uuid(*app_id=None*)
Generates a new UUID. This is a class method and can be called as *Experiment.make_uuid()*. Takes an optional *app_id* which is converted to a string and, if it is a valid UUID, returned.

networks(*role='all'*, *full='all'*)
All the networks in the experiment.

node_get_request(*node=None*, *nodes=None*)
Run when a request to get nodes is complete.

node_post_request(*participant*, *node*)
Run when a request to make a node is complete.

recruit()
Recruit participants to the experiment as needed.

This method runs whenever a participant successfully completes the experiment (participants who fail to finish successfully are automatically replaced). By default it recruits 1 participant at a time until all networks are full.

replay_event(*event*)
Stub method to replay an event returned by *events_for_replay()*. Experiments must override this method to provide replay support.

replay_start()
Stub method for starting an experiment replay. Experiments must override this method to provide replay support.

replay_finish()
Stub method for ending an experiment replay. Experiments must override this method to provide replay support.

replay_started()
Returns *True* if an experiment replay has started.

run(*exp_config=None*, *app_id=None*, *bot=False*, ***kwargs*)
Deploy and run an experiment.

The exp_config object is either a dictionary or a `localconfig.LocalConfig` object with parameters specific to the experiment run grouped by section.

save(**objects*)
Add all the objects to the session and commit them.

This only needs to be done for networks and participants.

setup()
Create the networks if they don't already exist.

submission_successful(*participant*)
Run when a participant submits successfully.

transformation_get_request(*node*, *transformations*)
Run when a request to get transformations is complete.

transformation_post_request(*node*, *transformation*)
Run when a request to transform an info is complete.

transmission_get_request(*node*, *transmissions*)
Run when a request to get transmissions is complete.

transmission_post_request(*node*, *transmissions*)
Run when a request to transmit is complete.

vector_get_request(*node*, *vectors*)
Run when a request to get vectors is complete.

vector_post_request(*node*, *vectors*)
Run when a request to connect is complete.

3.6 Database API

The classes involved in a Dallinger experiment are: *Network*, *Node*, *Vector*, *Info*, *Transmission*, *Transformation*, *Participant*, and *Question*. The code for all these classes can be seen in `models.py`. Each class has a corresponding table in the database, with each instance stored as a row in the table. Accordingly, each class is defined, in part, by the columns that constitute the table it is stored in. In addition, the classes have relationships to other objects and a number of functions.

The classes have relationships to each other as shown in the diagram below. Be careful to note which way the arrows point. A *Node* is a point in a *Network* that might be associated with a *Participant*. A *Vector* is a directional connection between a *Node* and another *Node*. An *Info* is information created by a *Node*. A *Transmission* is an instance of an *Info* being sent along a *Vector*. A *Transformation* is a relationship between an *Info* and another *Info*. A *Question* is a survey response created by a *Participant*.

3.6.1 SharedMixin

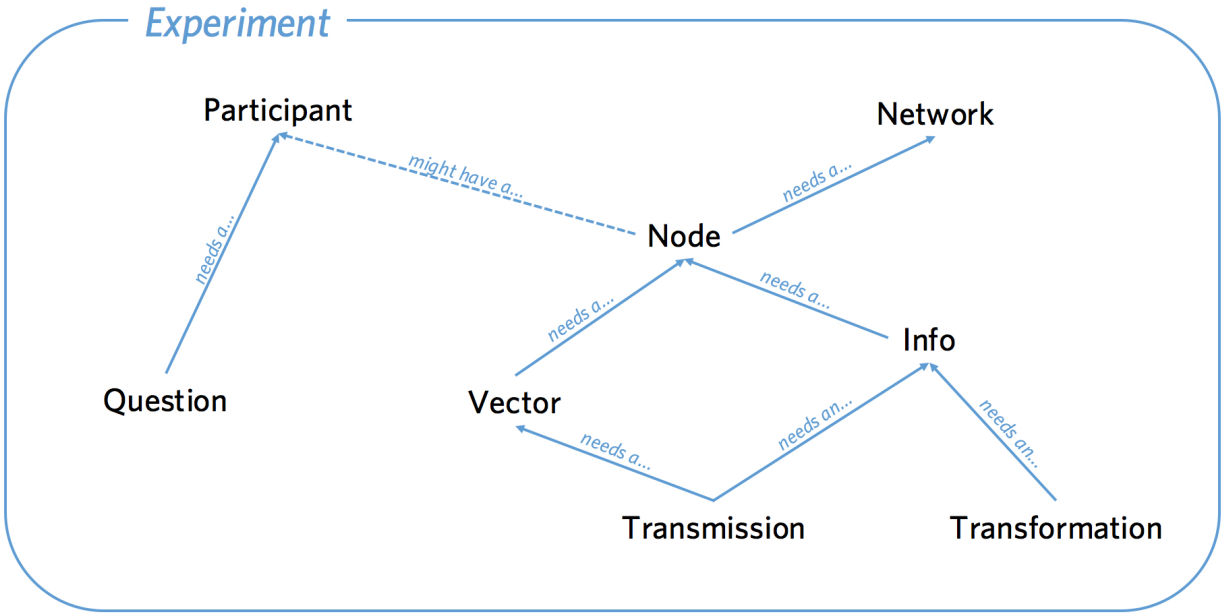
All Dallinger classes inherit from a `SharedMixin` which provides multiple columns that are common across tables:

`SharedMixin`.**id**
a unique number for every entry. 1, 2, 3 and so on...

`SharedMixin`.**creation_time**
the time at which the Network was created.

`SharedMixin`.**property1**
a generic column that can be used to store experiment-specific details in String form.

`SharedMixin`.**property2**
a generic column that can be used to store experiment-specific details in String form.



`SharedMixin.`**`property3`**
a generic column that can be used to store experiment-specific details in String form.

`SharedMixin.`**`property4`**
a generic column that can be used to store experiment-specific details in String form.

`SharedMixin.`**`property5`**
a generic column that can be used to store experiment-specific details in String form.

`SharedMixin.`**`failed`**
boolean indicating whether the Network has failed which prompts Dallinger to ignore it unless specified otherwise. Objects are usually failed to indicate something has gone wrong.

`SharedMixin.`**`time_of_death`**
the time at which failing occurred

3.6.2 Network

The *`Network`* object can be imagined as a set of other objects with some functions that perform operations over those objects. The objects that *`Network`*'s have direct access to are all the *`Node`*'s in the network, the *`Vector`*'s between those Nodes, Infos created by those Nodes, Transmissions sent along the Vectors by those Nodes and Transformations of those Infos. Participants and Questions do not exist within Networks. An experiment may involve multiple Networks, Transmissions can only occur within networks, not between them.

`class` `dallinger.models.`**`Network`**(***kwargs*)
Contains and manages a set of Nodes and Vectors etc.

Columns

`Network.`**`type`**
A String giving the name of the class. Defaults to "network". This allows subclassing.

`Network.`**`max_size`**
How big the network can get, this number is used by the full() method to decide whether the network is full

Network.**full**
Whether the network is currently full

Network.**role**
The role of the network. By default dallinger initializes all networks as either "practice" or "experiment"

Relationships

dallinger.models.Network.**all_nodes**
All the Nodes in the network.

dallinger.models.Network.**all_vectors**
All the vectors in the network.

dallinger.models.Network.**all_infos**
All the infos in the network.

dallinger.models.Network.**networks_transmissions**
All the transmissions int he network.

dallinger.models.Network.**networks_transformations**
All the transformations in the network.

Methods

Network.**__repr__**()
The string representation of a network.

Network.**__json__**()
Return json description of a participant.

Network.**calculate_full**()
Set whether the network is full.

Network.**fail**()
Fail an entire network.

Network.**infos**(*type=None*, *failed=False*)
Get infos in the network.

type specifies the type of info (defaults to Info). failed { False, True, "all" } specifies the failed state of the infos. To get infos from a specific node, see the infos() method in class *Node*.

Network.**latest_transmission_recipient**()
Get the node that most recently received a transmission.

Network.**nodes**(*type=None*, *failed=False*, *participant_id=None*)
Get nodes in the network.

type specifies the type of Node. Failed can be "all", False (default) or True. If a participant_id is passed only nodes with that participant_id will be returned.

Network.**print_verbose**()
Print a verbose representation of a network.

Network.**size**(*type=None*, *failed=False*)
How many nodes in a network.

type specifies the class of node, failed can be True/False/all.

Network.**transformations**(*type=None*, *failed=False*)
Get transformations in the network.

type specifies the type of transformation (default = Transformation). failed = { False, True, "all" }

To get transformations from a specific node, see Node.transformations().

Network.**transmissions**(*status='all'*, *failed=False*)
Get transmissions in the network.

status { "all", "received", "pending" } failed { False, True, "all" } To get transmissions from a specific vector, see the transmissions() method in class Vector.

Network.**vectors**(*failed=False*)
Get vectors in the network.

failed = { False, True, "all" } To get the vectors to/from to a specific node, see Node.vectors().

3.6.3 Node

Each Node represents a single point in a single network. A Node must be within a Network and may also be associated with a Participant.

class dallinger.models.**Node**(*network*, *participant=None*)
A point in a network.

Columns

Node.**type**
A String giving the name of the class. Defaults to `node`. This allows subclassing.

Node.**network_id**
the id of the network that this node is a part of

Node.**participant_id**
the id of the participant whose node this is

Relationships

Node.**network**
the network the node is in

Node.**participant**
the participant the node is associated with

dallinger.models.Node.**all_outgoing_vectors**
All the vectors going out from this Node.

dallinger.models.Node.**all_incoming_vectors**
All the vectors coming in to this Node.

dallinger.models.Node.**all_infos**
All Infos created by this Node.

dallinger.models.Node.**all_outgoing_transmissions**
All Transmissions sent from this Node.

dallinger.models.Node.**all_incoming_transmissions**
All Transmissions sent to this Node.

dallinger.models.Node.**transformations_here**
All transformations that took place at this Node.

Methods

Node.**__repr__**()
The string representation of a node.

Node.**__json__**()
The json of a node.

Node.**_to_whom**()
To whom to transmit if to_whom is not specified.

Return the default value of `to_whom` for *transmit()*. Should not return None or a list containing None.

Node.**_what**()
What to transmit if what is not specified.

Return the default value of `what` for *transmit()*. Should not return None or a list containing None.

Node.**connect**(*whom*, *direction='to'*)
Create a vector from self to/from whom.

Return a list of newly created vector between the node and whom. `whom` can be a specific node or a (nested) list of nodes. Nodes can only connect with nodes in the same network. In addition nodes cannot connect with themselves or with Sources. `direction` specifies the direction of the connection it can be "to" (node -> whom), "from" (whom -> node) or both (node <-> whom). The default is "to".

Whom may be a (nested) list of nodes.

Will raise an error if:

1. whom is not a node or list of nodes
2. whom is/contains a source if direction is to or both
3. whom is/contains self
4. whom is/contains a node in a different network

If self is already connected to/from whom a Warning is raised and nothing happens.

This method returns a list of the vectors created (even if there is only one).

Node.**fail**()
Fail a node, setting its status to "failed".

Also fails all vectors that connect to or from the node. You cannot fail a node that has already failed, but you can fail a dead node.

Set node.failed to True and `time_of_death` to now. Instruct all not-failed vectors connected to this node, infos made by this node, transmissions to or from this node and transformations made by this node to fail.

Node.**is_connected**(*whom*, *direction='to'*, *failed=None*)
Check whether this node is connected [to/from] whom.

whom can be a list of nodes or a single node. direction can be "to" (default), "from", "both" or "either".

If whom is a single node this method returns a boolean, otherwise it returns a list of booleans

Node.**infos**(*type=None*, *failed=False*)
Get infos that originate from this node.

Type must be a subclass of *Info*, the default is `Info`. Failed can be True, False or "all".

Node.**mutate**(*info_in*)
Replicate an info + mutation.

To mutate an info, that info must have a method called `_mutated_contents`.

Node.**neighbors**(*type=None*, *direction='to'*, *failed=None*)
Get a node's neighbors - nodes that are directly connected to it.

Type specifies the class of neighbour and must be a subclass of Node (default is Node). Connection is the direction of the connections and can be "to" (default), "from", "either", or "both".

Node.**receive**(*what=None*)
Receive some transmissions.

Received transmissions are marked as received, then their infos are passed to update().

"what" can be:

1. None (the default) in which case all pending transmissions are received.
2. a specific transmission.

Will raise an error if the node is told to receive a transmission it has not been sent.

Node.**received_infos**(*type=None*, *failed=None*)
Get infos that have been sent to this node.

Type must be a subclass of info, the default is Info.

Node.**replicate**(*info_in*)
Replicate an info.

Node.**transformations**(*type=None*, *failed=False*)
Get Transformations done by this Node.

type must be a type of Transformation (defaults to Transformation) Failed can be True, False or "all"

Node.**transmissions**(*direction='outgoing'*, *status='all'*, *failed=False*)
Get transmissions sent to or from this node.

Direction can be "all", "incoming" or "outgoing" (default). Status can be "all" (default), "pending", or "received". failed can be True, False or "all"

Node.**transmit**(*what=None*, *to_whom=None*)
Transmit one or more infos from one node to another.

"what" dictates which infos are sent, it can be:

1. None (in which case the node's _what method is called).
2. an Info (in which case the node transmits the info)
3. a subclass of Info (in which case the node transmits all its infos of that type)
4. a list of any combination of the above

"to_whom" dictates which node(s) the infos are sent to, it can be:

1. None (in which case the node's _to_whom method is called)
2. a Node (in which case the node transmits to that node)
3. a subclass of Node (in which case the node transmits to all nodes of that type it is connected to)
4. a list of any combination of the above

Will additionally raise an error if:

1. _what() or _to_whom() returns None or a list containing None.
2. what is/contains an info that does not originate from the transmitting node
3. to_whom is/contains a node that the transmitting node does not have a not-failed connection with.

Node.**update**(*infos*)
Process received infos.

Update controls the default behavior of a node when it receives infos. By default it does nothing.

Node.**vectors**(*direction='all'*, *failed=False*)
Get vectors that connect at this node.

Direction can be "incoming", "outgoing" or "all" (default). Failed can be True, False or all

3.6.4 Vector

A vector is a directional link between two nodes. Nodes connected by a vector can send Transmissions to each other, but because Vectors have a direction, two Vectors are needed for bi-directional Transmissions.

class dallinger.models.**Vector**(*origin*, *destination*)
A directed path that links two Nodes.

Nodes can only send each other information if they are linked by a Vector.

Columns

Vector.**origin_id**
the id of the Node at which the vector originates

Vector.**destination_id**
the id of the Node at which the vector terminates.

Vector.**network_id**
the id of the network the vector is in.

Relationships

Vector.**origin**
the Node at which the vector originates.

Vector.**destination**
the Node at which the vector terminates.

Vector.**network**
the network the vector is in.

dallinger.models.Vector.**all_transmissions**
All Transmissions sent along the Vector.

Methods

Vector.**__repr__**()
The string representation of a vector.

Vector.**__json__**()
The json representation of a vector.

Vector.**fail**()
Fail a vector.

Vector.**transmissions**(*status='all'*)
Get transmissions sent along this Vector.

Status can be "all" (the default), "pending", or "received".

3.6.5 Info

An Info is a piece of information created by a Node. It can be sent along Vectors as part of a Transmission.

class dallinger.models.**Info**(*origin*, *contents=None*, *details=None*)
A unit of information.

Columns

Info.**id**

Info.**creation_time**

Info.**property1**

Info.**property2**

Info.**property3**

Info.**property4**

Info.**property5**

Info.**failed**

Info.**time_of_death**

Info.**type**
a String giving the name of the class. Defaults to "info". This allows subclassing.

Info.**origin_id**
the id of the Node that created the info

Info.**network_id**
the id of the network the info is in

Info.**contents**
the contents of the info. Must be stored as a String.

Relationships

Info.**origin**
the Node that created the info.

Info.**network**
the network the info is in

dallinger.models.Info.**all_transmissions**
All Transmissions of this Info.

dallinger.models.Info.**transformation_applied_to**
All Transformations of which this info is the info_in

dallinger.models.Info.**transformation_whence**
All Transformations of which this info is the info_out

Methods

Info.**__repr__**()
The string representation of an info.

Info.**__json__**()
The json representation of an info.

Info.**_mutated_contents**()
The mutated contents of an info.

When an info is asked to mutate, this method will be executed in order to determine the contents of the new info created.

The base class function raises an error and so must be overwritten to be used.

Info.**fail**()
Fail an info.

Set info.failed to True and *time_of_death* to now. Instruct all transmissions and transformations involving this info to fail.

Info.**transformations**(*relationship='all'*)
Get all the transformations of this info.

Return a list of transformations involving this info. relationship can be "parent" (in which case only transformations where the info is the info_in are returned), "child" (in which case only transformations where the info is the info_out are returned) or all (in which case any transformations where the info is the info_out or the info_in are returned). The default is all

Info.**transmissions**(*status='all'*)
Get all the transmissions of this info.

status can be all/pending/received.

3.6.6 Transmission

A transmission represents an instance of an Info being sent along a Vector. Transmissions are not necessarily received when they are sent (like an email) and must also be received by the Node they are sent to.

class dallinger.models.**Transmission**(*vector*, *info*)
An instance of an Info being sent along a Vector.

Columns

Transmission.**origin_id**
the id of the Node that sent the transmission

Transmission.**destination_id**
the id of the Node that the transmission was sent to

Transmission.**vector_id**
the id of the vector the info was sent along

Transmission.**network_id**
the id of the network the transmission is in

Transmission.**info_id**
the id of the info that was transmitted

Transmission.**receive_time**
the time at which the transmission was received

Transmission.**status**
the status of the transmission, can be "pending", which means the transmission has been sent, but not received; or "received", which means the transmission has been sent and received

Relationships

Transmission.**origin**
the Node that sent the transmission.

Transmission.**destination**
the Node that the transmission was sent to.

Transmission.**vector**
the vector the info was sent along.

Transmission.**network**
the network the transmission is in.

Transmission.**info**
the info that was transmitted.

Methods

Transmission.**__repr__**()
The string representation of a transmission.

Transmission.**__json__**()
The json representation of a transmissions.

Transmission.**fail**()
Fail a transmission.

Transmission.**mark_received**()
Mark a transmission as having been received.

3.6.7 Transformation

A Transformation is a relationship between two Infos. It is similar to how a Vector indicates a relationship between two Nodes, but whereas a Vector allows Nodes to Transmit to each other, Transformations don't allow Infos to do anything new. Instead they are a form of book-keeping allowing you to keep track of relationships between various Infos.

class dallinger.models.**Transformation**(*info_in*, *info_out*)
An instance of one info being transformed into another.

Columns

Transformation.**type**
a String giving the name of the class. Defaults to "transformation". This allows subclassing.

Transformation.**node_id**
the id of the Node that did the transformation.

Transformation.**network_id**
the id of the network the transformation is in.

Transformation.**info_in_id**
the id of the info that was transformed.

Transformation.**info_out_id**
the id of the info produced by the transformation.

Relationships

Transformation.**node**
the Node that did the transformation.

Transformation.**network**
the network the transmission is in.

Transformation.**info_in**
the info that was transformed.

Transformation.**info_out**
the info produced by the transformation.

Methods

Transformation.**__repr__**()
The string representation of a transformation.

Transformation.**__json__**()
The json representation of a transformation.

Transformation.**fail**()
Fail a transformation.

3.6.8 Participant

The Participant object corresponds to a real world participant. Each person who takes part will have a corresponding entry in the Participant table. Participants can be associated with Nodes and Questions.

class dallinger.models.**Participant**(*recruiter_id*, *worker_id*, *assignment_id*, *hit_id*, *mode*, *fingerprint_hash=None*)
An ex silico participant.

Columns

Participant.**type**
a String giving the name of the class. Defaults to "participant". This allows subclassing.

Participant.**worker_id**
A String, the worker id of the participant.

Participant.**assignment_id**
A String, the assignment id of the participant.

Participant.**unique_id**
A String, a concatenation of *worker_id* and *assignment_id*

Participant.**hit_id**
A String, the id of the hit the participant is working on

Participant.**mode**
A String, the mode in which Dallinger is running – live, sandbox or debug.

Participant.**end_time**
The time at which the participant finished.

Participant.**base_pay**
The amount the participant was paid for finishing the experiment.

Participant.**bonus**
the amount the participant was paid as a bonus.

Participant.**status**
String representing the current status of the participant, can be –

- working - participant is working
- submitted - participant has submitted their work
- approved - their work has been approved and they have been paid
- rejected - their work has been rejected
- returned - they returned the hit before finishing
- abandoned - they ran out of time
- did_not_attend - the participant finished, but failed the attention check
- bad_data - the participant finished, but their data was malformed
- missing_notification - this indicates that Dallinger has inferred that a Mechanical Turk notification corresponding to this participant failed to arrive. This is an uncommon, but potentially serious issue.

Relationships

dallinger.models.Participant.**all_questions**
All the questions associated with this participant.

dallinger.models.Participant.**all_nodes**
All the Nodes associated with this participant.

Methods

Participant.**__json__**()
Return json description of a participant.

Participant.**fail**()
Fail a participant.

Set *failed* to True and *time_of_death* to now. Instruct all not-failed nodes associated with the participant to fail.

Participant.**infos**(*type=None*, *failed=False*)
Get all infos created by the participants nodes.

Return a list of infos produced by nodes associated with the participant. If specified, `type` filters by class. By default, failed infos are excluded, to include only failed nodes use `failed=True`, for all nodes use `failed=all`. Note that failed filters the infos, not the nodes - infos from all nodes (whether failed or not) can be returned.

`Participant.`**`nodes`**(*type=None*, *failed=False*)
Get nodes associated with this participant.

Return a list of nodes associated with the participant. If specified, `type` filters by class. By default failed nodes are excluded, to include only failed nodes use `failed=True`, for all nodes use `failed=all`.

`Participant.`**`questions`**(*type=None*)
Get questions associated with this participant.

Return a list of questions associated with the participant. If specified, `type` filters by class.

3.6.9 Question

A Question is a way to store information associated with a Participant as opposed to a Node (Infos are made by Nodes, not Participants). Questions are generally useful for storing responses debriefing questions etc.

`class` `dallinger.models.`**`Question`**(*participant*, *question*, *response*, *number*)
Responses of a participant to debriefing questions.

Columns

`Question.`**`type`**
a String giving the name of the class. Defaults to "question". This allows subclassing.

`Question.`**`participant_id`**
the participant who made the response

`Question.`**`number`**
A number identifying the question. e.g., each participant might complete three questions numbered 1, 2, and 3.

`Question.`**`question`**
the text of the question

`Question.`**`response`**
the participant's response. Stored as a string.

Relationships

`Question.`**`participant`**
the participant who answered the question

Methods

`Question.`**`__json__`**`()`
Return json description of a question.

`Question.`**`fail`**`()`
Fail a question.

Set *`failed`* to True and *`time_of_death`* to now.

3.7 Web API

The Dallinger API allows the experiment frontend to communicate with the backend. Many of these routes correspond to specific functions of Dallinger's *classes*, particularly `dallinger.models.Node`. For example, nodes have a `connect` method that creates new vectors between nodes and there is a corresponding `connect/` route that allows the frontend to call this method.

3.7.1 Miscellaneous routes

```
GET /ad_address/<mode>/<hit_id>
```

Used to get the address of the experiment on the gunicorn server and to return participants to Mechanical Turk upon completion of the experiment. This route is pinged automatically by the function `submitAssignment` in dallinger2.js.

```
GET /<directory>/<page>
```

Returns the html page with the name `<page>` from the directory called `<directory>`.

```
GET /summary
```

Returns a summary of the statuses of Participants.

```
GET /<page>
```

Returns the html page with the name `<page>`.

3.7.2 Experiment routes

```
GET /experiment/<property>
```

Returns the value of the requested property as a JSON `<property>`. The property must be a key in the `experiment.public_properties` mapping and be JSON serializable. Experiments have no public properties by default.

```
GET /info/<node_id>/<info_id>
```

Returns a JSON description of the requested info as `info`. `node_id` must be specified to ensure the requesting node has access to the requested info. Calls experiment method 'info_get_request(node, info).

```
POST /info/<node_id>
```

Create an info with its origin set to the specified node. *contents* must be passed as data. `info_type` can be passed as data and will cause the info to be of the specified type. Also calls experiment method `info_post_request(node, info)`.

```
POST /launch
```

Initializes the experiment and opens recruitment. This route is automatically pinged by Dallinger.

```
GET /network/<network_id>
```

Returns a JSON description of the requested network as `network`.

```
POST /node/<node_id>/connect/<other_node_id>
```

Create vector(s) between the `node` and `other_node` by calling `node.connect(whom=other_node)`. Direction can be passed as data and will be forwarded as an argument. Calls experiment method `vector_post_request(node, vectors)`. Returns a list of JSON descriptions of the created vectors as `vectors`.

```
GET /node/<node_id>/infos
```

Returns a list of JSON descriptions of the infos created by the node as `infos`. Infos are identified by calling `node.infos()`. `info_type` can be passed as data and will be forwarded as an argument. Requesting node and the list of infos are also passed to experiment method `info_get_request(node, infos)`.

```
GET /node/<node_id>/neighbors
```

Returns a list of JSON descriptions of the node's neighbors as `nodes`. Neighbors are identified by calling `node.neighbors()`. `node_type` and `connection` can be passed as data and will be forwarded as arguments. Requesting node and list of neighbors are also passed to experiment method `node_get_request(node, nodes)`.

```
GET /node/<node_id>/received_infos
```

Returns a list of JSON descriptions of the infos sent to the node as `infos`. Infos are identified by calling `node.received_infos()`. `info_type` can be passed as data and will be forwarded as an argument. Requesting node and the list of infos are also passed to experiment method `info_get_request(node, infos)`.

```
GET /node/<int:node_id>/transformations
```

Returns a list of JSON descriptions of all the transformations of a node identified using `node.transformations()`. The node id must be specified in the url. You can also pass `transformation_type` as data and it will be forwarded to `node.transformations()` as the argument `type`.

```
GET /node/<node_id>/transmissions
```

Returns a list of JSON descriptions of the transmissions sent to/from the node as `transmissions`. Transmissions are identified by calling `node.transmissions()`. `direction` and `status` can be passed as data and will be forwarded as arguments. Requesting node and the list of transmissions are also passed to experiment method `transmission_get_request(node, transmissions)`.

```
POST /node/<node_id>/transmit
```

Transmit to another node by calling `node.transmit()`. The sender's node id must be specified in the url. As with `node.transmit()` the key parameters are `what` and `to_whom` and they should be passed as data. However, the values these accept are more limited than for the backend due to the necessity of serialization.

If `what` and `to_whom` are not specified they will default to `None`. Alternatively you can pass an int (e.g. '5') or a class name (e.g. `Info` or `Agent`). Passing an int will get that info/node, passing a class name will pass the class. Note that if the class you are specifying is a custom class it will need to be added to the dictionary of known_classes in your experiment code.

You may also pass the values property1, property2, property3, property4 and property5. If passed this will fill in the relevant values of the transmissions created with the values you specified.

The transmitting node and a list of created transmissions are sent to experiment method `transmission_post_request(node, transmissions)`. This route returns a list of JSON descriptions of the created transmissions as `transmissions`. For example, to transmit all infos of type Meme to the node with id 10:

```
reqwest({
    url: "/node/" + my_node_id + "/transmit",
    method: 'post',
    type: 'json',
    data: {
        what: "Meme",
        to_whom: 10,
    },
});
```

```
GET /node/<node_id>/vectors
```

Returns a list of JSON descriptions of vectors connected to the node as `vectors`. Vectors are identified by calling `node.vectors()`. `direction` and `failed` can be passed as data and will be forwarded as arguments. Requesting node and list of vectors are also passed to experiment method `vector_get_request(node, vectors)`.

```
POST /node/<participant_id>
```

Create a node for the specified participant. The route calls the following experiment methods: `get_network_for_participant(participant)`, `create_node(network, participant)`, `add_node_to_network(node, network)`, and `node_post_request(participant, node)`. Returns a JSON description of the created node as `node`.

```
POST /notifications
GET /notifications
```

This is the route to which notifications from AWS are sent. It is also possible to send your own notifications to this route, thereby simulating notifications from AWS. Necessary arguments are `Event.1.EventType`, which can be `AssignmentAccepted`, `AssignmentAbandoned`, `AssignmentReturned` or `AssignmentSubmitted`, and `Event.1.AssignmentId`, which is the id of the relevant assignment. In addition, Dallinger uses a custom event type of `NotificationMissing`.

```
GET /participant/<participant_id>
```

Returns a JSON description of the requested participant as `participant`.

```
POST /participant/<worker_id>/<hit_id>/<assignment_id>/<mode>
```

Create a participant. Returns a JSON description of the participant as `participant`.

```
POST /question/<participant_id>
```

Create a question. `question`, `response` and `question_id` should be passed as data. Does not return anything.

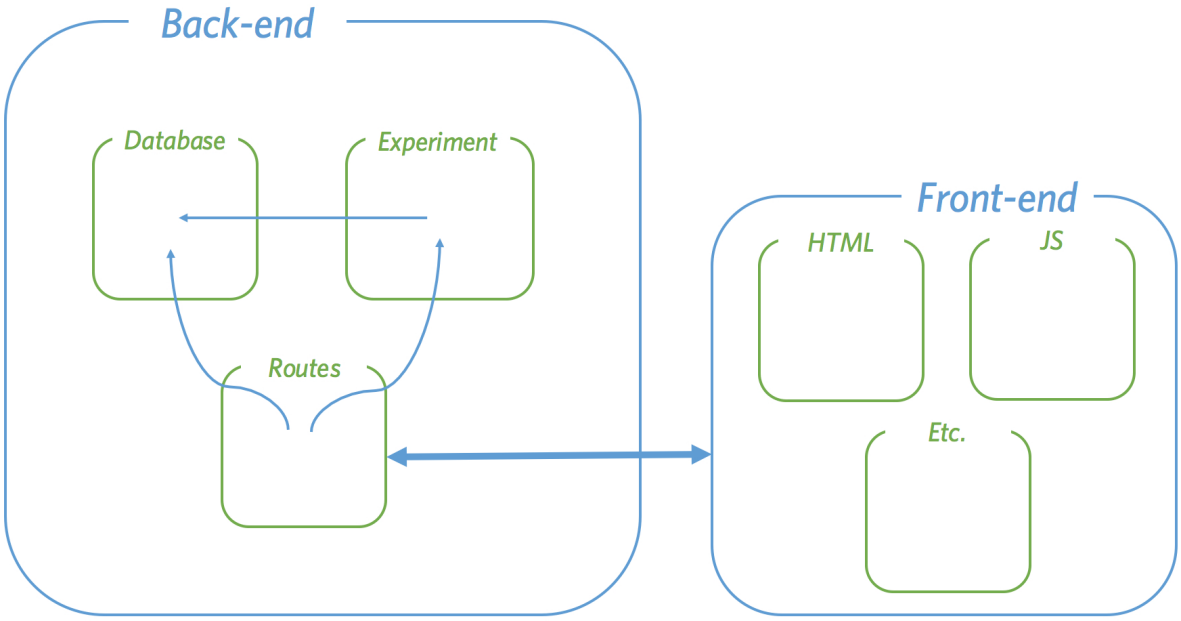
```
POST /transformation/<int:node_id>/<int:info_in_id>/<int:info_out_id>
```

Create a transformation from `info_in` to `info_out` at the specified node. `transformation_type` can be passed as data and the transformation will be of that class if it is a known class. Returns a JSON description of the created transformation.

3.8 Communicating With the Server

When an experiment is running, the database and the experiment class (i.e. the instructions for what to do with the database) will be hosted on a server, the server is also known as the "back-end". However, participants will take part

in experiments via an interactive web-site (the "front-end"). Accordingly for an experiment to proceed there must be a means of communication between the front and back ends. This is achieved with routes:



Routes are specific web addresses on the server that respond to requests from the front-end. Routes have direct access to the database, though most of the time they will pass requests to the experiment which will in turn access the database. As such, changing the behavior of the experiment class is the easiest way to create a new experiment. However it is also possible to change the behavior of the routes or add new routes entirely.

Requests generally come in two types: "get" requests, which ask for information from the database, and "post" requests which send new information to be added to the database. Once a request is complete the back-end sends a response back to the front-end. Minimally, this will include a notification that the request was successfully processed, but often it will also include additional information.

As long as requests are properly formatted and correctly addressed to routes, the back-end will send the appropriate response. This means that the front-end could take any form. For instance requests could come from a standard HTML/CSS/JS webpage, a more sophisticated web-app, or even from the experiment itself.

3.9 Javascript API

Dallinger provides a javascript API to facilitate creating web-based experiments. All of the dallinger demos use this API to communicate with the experiment server. The API is defined in the *dallinger2.js* script, which is included in the default experiment templates.

3.9.1 The *dallinger* object

Any page that includes *dallinger2.js* script will have a *dallinger* object added to the *window* global namespace. This object defines a number of functions for interacting with Dallinger experiments.

Making requests to experiment routes

dallinger provides functions which can be used to asynchronously interact with any of the experiment routes described in *Web API*:

`dallinger.get`(*route*, *data*)

Convenience method for making an AJAX `GET` request to a specified route. Any callbacks provided to the *done()* method of the returned *Deferred* object will be passed the JSON object returned by the the API route (referred to as *data* below). Any callbacks provided to the *fail()* method of the returned *Deferred* object will be passed an instance of *AjaxRejection*, see *Deferred objects*.

Arguments

- **route** (*string*) – Experiment route, e.g. `/info/$nodeId`
- **data** (*object*) – Optional data to include in request

Returns **jQuery.Deferred** – See *Deferred objects*

Examples:

```
var response = dallinger.get('/participant/1');
// Wait for response and handle data
response.done(function (data) {...});
```

`dallinger.post`(*route*, *data*)

Convenience method for making an AJAX `POST` request to a specified route. Any callbacks provided to the *done()* method of the returned *Deferred* object will be passed the JSON object returned by the the API route (referred to as *data* below). Any callbacks provided to the *fail()* method of the returned *Deferred* object will be passed an instance of *AjaxRejection*, see *Deferred objects*.

Arguments

- **route** (*string*) – Experiment route, e.g. `/info/$nodeId`
- **data** (*object*) – Optional data to include in request

Returns **jQuery.Deferred** – See *Deferred objects*

Examples:

```
var response = dallinger.post('/info/1', {details: {a: 1}});
// Wait for response and handle data or failure
response.done(function (data) {...}).fail(function (rejection) {...});
```

The *dallinger* object also provides functions that make requests to specific experiment routes:

`dallinger.createAgent`()

Creates a new experiment *Node* for the current partcipant.

Returns **jQuery.Deferred** – See *Deferred objects*

Examples:

```
var response = dallinger.createAgent();
// Wait for response
response.done(function (data) {... handle data.node ...});
```

`dallinger.createInfo`(*nodeId*, *data*)

Creates a new *Info* object in the experiment database.

Arguments

- **nodeId** (*number*) – The id of the participant's experiment node
- **data** (*Object*) – Experimental data (see *Info*)

Returns **jQuery.Deferred** – See *Deferred objects*

Examples:

```
var response = dallinger.createInfo(1, {details: {a: 1}});
// Wait for response
response.done(function (data) {... handle data.info ...});
```

dallinger.**getInfo**(*nodeId*, *infoId*)

Get a specific *Info* object from the experiment database.

Arguments

- **nodeId** (*number*) – The id of an experiment node
- **infoId** (*number*) – The id of the Info object to be retrieved

Returns **jQuery.Deferred** – See *Deferred objects*

Examples:

```
var response = dallinger.getInfo(1, 1);
// Wait for response
response.done(function (data) {... handle data.info ...});
```

dallinger.**getInfos**(*nodeId*)

Get all *Info* objects for the specified node.

Arguments

- **nodeId** (*number*) – The id of an experiment node.

Returns **jQuery.Deferred** – See *Deferred objects*

Examples:

```
var response = dallinger.getInfos(1, 1);
// Wait for response
response.done(function (data) {... handle data.infos ...});
```

dallinger.**getReceivedInfos**(*nodeId*)

Get all the *Info* objects a node has been sent and has received.

Arguments

- **nodeId** (*number*) – The id of an experiment node.

Returns **jQuery.Deferred** – See *Deferred objects*

Examples:

```
var response = dallinger.getReceivedInfostInfos(1);
// Wait for response
response.done(function (data) {... handle data.infos ...});
```

dallinger.**getTransmissions**(*nodeId*, *data*)

Get all *Transmission* objects connected to a node.

Arguments

- **nodeId** (*number*) – The id of an experiment node.

- **data** (*Object*) – Additional parameters, specifically `direction` (to/from/all) and `status` (all/pending/received).

Returns **jQuery.Deferred** – See *Deferred objects*

Examples:

```
var response = dallinger.getTransmissions(1, {direction: "to", status: "all"});
// Wait for response
response.done(function (data) {... handle data.transmissions ...});
```

Additionally, there is a helper method to handle error responses from experiment API calls (see *Deferred objects* below):

`dallinger.`**`error`**`(`*rejection*`)`

Handles experiment errors by requesting feedback from the participant and attempts to complete the experiment (and compensate participants).

Arguments

- **rejection** (*dallinger.AjaxRejection*) – information about the AJAX error.

Examples:

```
// Let dallinger handle the error
dallinger.createAgent().fail(dallinger.error);

// Custom handling, then request feedback and complete if possible
dallinger.getInfo(info).fail(function (rejection) {
 ... handle rejection data ...
 dallinger.error(rejection);
});
```

Deferred objects

All of the above functions make use of jQuery.Deferred, and return *Deferred* objects. These *Deferred* objects provide the following methods to facilitate handling asynchronous responses once they've completed:

- `.done(callback)`: Provide a callback to handle data from a successful response
- `.fail(fail_callback)`: Provide a callback to handle error responses
- `.then(callback[, fail_callback, progress_callback])`: Provide callbacks to handle successes, failures, and progress updates.

The fail_callback function will be passed a *dallinger.AjaxRejection* object which includes detailed information about the error. Unexpected errors should be handled by calling the `dallinger.error()` method with the *AjaxRejection* object.

Experiment Initialization and Completion

In addition to the request functions above, there are a few functions that are used by the default experiment templates to setup and complete an experiment. If you are writing a highly customized experiment, you may need to use these explicitly:

`dallinger.`**`createParticipant`**`()`

Create a new experiment *Participant* by making a `POST` request to the experiment */participant/* route. If the experiment requires a quorum, the response will not resolve until the quorum is met. If the participant is

requested after the quorum has already been reached, the `dallinger.skip_experiment` flag will be set and the experiment will be skipped.

This method is called automatically by the default waiting room page.

Returns **jQuery.Deferred** – See *Deferred objects*

`dallinger.`**`hasAdBlocker`**(*callback*)
Determine if the user has an ad blocker installed. If an ad blocker is detected the callback will be executed asynchronously after a small delay.

This method is called automatically from the experiment default template.

Arguments

- **`callback`** (*`function`*) – a function, with no arguments, to call if an ad blocker is running.

`dallinger.`**`submitAssignment`**()
Notify the experiment that the participant's assignment is complete. Performs a `GET` request to the experiment's `/worker_complete` route.

Returns **jQuery.Deferred** – See *Deferred objects*

Examples:

```
// Mark the assignment complete and perform a custom function when successful
result = dallinger.submitAssignment();
result.done(function (data) {... handle ``data.status`` ...}).fail(
    dallinger.error
);
```

`dallinger.`**`submitQuestionnaire`**(*name="questionnaire"*)
Submits a *Question* object to the experiment server. This method is called automatically from the default questionnaire page.

Arguments

- **`name`** (*`string`*) – optional questionnaire name

`dallinger.`**`waitForQuorum`**()
Waits for a WebSocket message indicating that quorum has been reached.

This method is called automatically within *createParticipant()* and the default waiting room page.

Returns **jQuery.Deferred** – See *Deferred objects*

Helper functions and properties

Finally, there are a few miscellaneous utility functions and properties which are useful when writing a custom experiment:

`dallinger.`**`getUrlParameter`**(*sParam*)
Returns a url query string value given the parameter name.

Arguments

- **`sParam`** (*`string`*) – name of url parameter

Returns **string|boolean** – the parameter value if available; `true` if parameter is in the url but has no value;

Examples:

```
// Given a url with ``?param1=aaa&param2``, the following returns "aaa"
dallinger.getUrlParameter("param1");
// this returns true
dallinger.getUrlParameter("param2");
// and this returns null
dallinger.getUrlParameter("param3");
```

dallinger.**goToPage**(*page*)

Advance the participant to a given html page; the participant_id will be included in the url query string.

Arguments

- **page** (*string*) – Name of page to load, the .html extension should not be included.

dallinger.**identity**

dallinger.identity provides information about the participant. It has the following string properties:

recruiter - Type of recruiter

hitId - MTurk HIT Id

workerId - MTurk Worker Id

assignmentId - MTurk Assignment Id

mode - Dallinger experiment mode

participantId - Dallinger participant Id

3.10 Rewarding participants

It is common for experiments to remunerate participants in two ways, a base payment for participation and a bonus for their particular performance. Payments are managed through the recruiter being used, so it is important to consider any differences if changing the recruiter to ensure that there isn't an inadvertent change to the mechanics of the experiment.

3.10.1 Base payment

The base payment is controlled by the base_payment configuration variable, which is a number of US dollars. This can be set as any configuration value and is accessed directly by the recruiter rather than being mediated through the experiment.

For example, to deploy an experiment using a specific payout of 4.99 USD the following command line invocation can be used:

base_payment=4.99 dallinger deploy

3.10.2 Bonus payment

The bonus payment is more complex, as it is set by the experiment class in response to an individual participant completing the experiment. In order to keep the overall payment amounts flexible it is strongly recommended to parameterize this calculation.

There are many strategies for awarding bonuses, some examples of which are documented below. In each case, bonus(self, participant) is a reference to *bonus()* in your experiment class.

Time based bonuses

This pays the user a bonus based on the amount of time they spent on the experiment. While this helps to pay users fairly for their time it also incentivises slow performance of the task. Without a maximum being set or adequate attention checks it may be possible for participants to receive a large bonus by ignoring the experiment for some time.

This method is a good fit if there is a lot of variation between how long it takes people to complete a task while putting in the same effort, for example if there is a reliance on waiting rooms.

```
def bonus(self, participant):
    """Give the participant a bonus for waiting."""
    elapsed_time = participant.end_time - participant.creation_time
    # keep to two decimal points to represent cents
    payout = round(
        (elapsed_time.total_seconds() / 3600.0) * config.get('payment_per_hour', 5.
↪00),
        2
    )
    return min(payout, config.get('max_bonus_amount', 10000.00))
```

This expects two configuration parameters, `payment_per_hour` and `max_bonus_amount` in addition to the `base_payment` value.

The bonus is then calculated as the number of hours between the participant being created and them finishing the experiment, at `payment_per_hour` dollars per hour, with a maximum of `max_bonus_amount`.

Performance based bonuses

This pays the user based on how well they perform in the experiment. It is very important that this calculation be performed by the Experiment class rather than the front-end Javascript, as otherwise unscrupulous users could specify arbitrary rewards.

The `bonus` function should be kept as simple as possible, delegating to other functions for readability.

For example, the demos/bartlett1932/index demo involves showing participants a piece of text and asking them to reproduce it from memory. A simple reward function could be as follows:

```
def get_submitted_text(self, participant):
    """The text a given participant submitted"""
    node = participant.nodes()[0]
    return node.infos()[0].contents

def get_read_text(self, participant):
    """The text that a given participant was shown to memorize"""
    node = participant.nodes()[0]
    incoming = node.all_incoming_vectors[0]
    parent_node = incoming.origin
    return parent_node.infos()[0].contents

def text_similarity(self, one, two):
    """Return a measure of the similarity between two texts"""
    try:
        from Levenshtein import ratio
    except ImportError:
        from difflib import SequenceMatcher
        ratio = lambda x, y: SequenceMatcher(None, x, y).ratio()
    return ratio(one, two)
```

(continues on next page)

(continued from previous page)

```
def bonus(self, participant):
    performance = self.text_similarity(
        self.get_submitted_text(participant),
        self.get_read_text(participant)
    )
    payout = round(config.get('bonus_amount', 0.00) * performance, 2)
    return min(payout, config.get('max_bonus_amount', 10000.00))
```

The majority of the work in determining how a user has performed is handled by helper functions, to avoid confusing the logic of the bonus function, which is kept easy to read.

There is a secondary advantage, in that the performance helper functions can be used by other parts of the code. The main place these can be useful is the `attention_check` function, which is used to determine if a user was actively participating in the experiment or not.

In this example, it is possible that users will 'cheat' by copy/pasting the text they were supposed to remember, and therefore get the full reward. Alternatively, they may simply submit without trying, making the rest of the run useless. Although we wouldn't want to award the user a bonus for either of these, it's more appropriate for this to fail the `attention_check`, as the participant will be automatically replaced.

That may look like this:

```
def attention_check(self, participant):
    performance = self.text_similarity(
        self.get_submitted_text(participant),
        self.get_read_text(participant)
    )
    return (
        config.get('min_expected_performance', 0.1)
        <= performance <=
        config.get('max_expected_performance', 0.8)
    )
```

Javascript-only experiments

Sometimes experimenters may wish to convert an existing Javascript and HTML experiment to run within the Dallinger framework. Such games rely on logic entirely running in the user's browser, rather than instructions from the Dallinger Experiment class. However, code running in the user's browser cannot be trusted to determine how much the user should be paid, as it is open to manipulation through debugging tools.

Note: It might seem unlikely that users would bother to cheat, but it is quite easy for technically proficient users to do so if they choose, and the temptation of changing their payout may be too much to resist.

In order to integrate with Dallinger, the experiment must use the dallinger2.js function `createInfo` function to send its current state to the server. This is what allows analysis of the user's performance later, so it's important to send as much information as possible.

The included demos/twentyfortyeight/index demo is an example of this type of experiment. It shows a popular javascript game with no interaction with the server or other players. Tiles in the grid have numbers associated with them, which can be combined to gain higher numbered tiles. If the experimenter wanted to give a bonus based on the highest tile the user reached there is a strong incentive for the player to try and cheat and therefore receive a much larger payout than expected.

In this case, the data is sent to the server as:

```
if (moved) {
    this.addRandomTile();

    dallinger.createInfo(my_node_id, {
        contents: JSON.stringify(game.serialize()),
        info_type: "State"
    });
};
```

The experiment can then look at the latest state that was sent in order to find the highest card a user found.

```
def performance(self, participant):
    latest_info = participant.infos()[0]
    grid_state = json.loads(latest_info.contents)
    values = [
        cell['value']
        for row in grid_state['grid']['cells']
        for cell in row
    ]
    return min(2048.0 / max(values), 1.0)

def bonus(self, participant):
    performance = self.performance(participant)
    payout = round(config.get('bonus_amount', 0.00) * performance, 2)
    return min(payout, config.get('max_bonus_amount', 10000.00))
```

However, the states the experiment is looking at are still supplied by the user's browser, so although cheating would be more complex than simply changing a score it is still possible for them to cause a fraudulent state to be sent.

For this reason, we need to implement the game's logic in Python so that the `attention_check` can check that the user's play history is consistent. Again, this has the advantage that a user who cheats is removed from the experiment rather than simply receiving a diminished reward.

This may look something like:

```
def is_possible_transition(self, old, new):
    """Check if it is possible to get from the old state to the new state in one step"
↪""
    ...
    return True

def attention_check(self, participant):
    """Find all pairs of grid states and check they are all legitimate successors"""
    states = []
    for info in reversed(participant.infos()):
        states.append(json.loads(info.contents))
    pairs = zip(states, states[1:])
    return all(self.is_possible_transition(old, new) for (old, new) in pairs)
```

where `is_possible_transition` would be a rather complex function implementing the game's rules.

Note: In all these cases, it is strongly recommended to set a maximum bonus and return the minimum value between the bonus calculated and the maximum bonus, ensuring that no bugs or unexpected cheating cause a larger bonus to be awarded than expected.

3.11 Waiting rooms

By default, Dallinger begins an experiment as soon as a user agrees to the informed consent form and has read the instructions. However, some experiment designs require multiple users to be synchronized.

For this reason, Dallinger includes a waiting room implementation, which will hold users between instructions and the experiment until a certain number are ready.

3.11.1 Using the waiting room

To use the waiting room, users must first be directed into it rather than the experiment. The demos/chatroom/index demo shows an example of this.

Your `instructions.html` should call `dallinger.goToPage('waiting')` and should not call `dallinger.createParticipant`.

You will also need to define how many users should be held together before progressing. This is done through the `quorum` global variable. The waiting room will call a javascript function called `getQuorum` which should set quorum to be the appropriate value for your experiment.

3.12 Writing bots

When you run an experiment using the bot recruiter, it will look for a class named `Bot` in your `experiment.py` module.

The Bot class should typically be a subclass of either *`BotBase`* (for bots that interact with the experiment by controlling a real browser using `selenium`) or *`HighPerformanceBotBase`* (for bots that interact with the experiment server directly via HTTP or websockets).

The interaction of the base bots with the experiment takes place in several phases:

1. Signup (including creating a Participant)
2. Participation in the experiment
3. Signoff (including completing the questionnaire)
4. Recording completion (complete or failed)

To build a bot, you will definitely need to implement the `participate` method which will be called once the bot has navigated to the main experiment page. If the structure of your ad, consent, instructions or questionnaire pages differs significantly from the demo experiments, you may need to override other methods too.

3.12.1 High-performance bots

The *`HighPerformanceBotBase`* can be used as a basis for a bot that interacts with the experiment server directly over HTTP rather than using a real browser. This scales better than using Selenium bots, but requires expressing the bot's behavior in terms of HTTP requests rather than in terms of DOM interactions.

For a guide to Dallinger's web API, see *Web API*.

For an example of a high-performance bot implementation, see the Griduniverse bots. These bots interact primarily via websockets rather than HTTP.

API documentation

class dallinger.bots.**HighPerformanceBotBase**(*URL*, *assignment_id=''*, *worker_id=''*, *participant_id=''*, *hit_id=''*)

A base class for bots that do not interact using a real browser.

Instead, this kind of bot makes requests directly to the experiment server.

complete_experiment(*status*)

Record worker completion status to the experiment server.

This is done using a GET request to the /worker_complete or /worker_failed endpoints.

complete_questionnaire()

Complete the standard debriefing form.

Answers the questions in the base questionnaire.

driver

Returns a Selenium WebDriver instance of the type requested in the configuration.

on_signup(*data*)

Take any needed action on response from /participant call.

run_experiment()

Runs the phases of interacting with the experiment including signup, participation, signoff, and recording completion.

sign_off()

Submit questionnaire and finish.

This is done using a POST request to the /question/ endpoint.

sign_up()

Signs up a participant for the experiment.

This is done using a POST request to the /participant/ endpoint.

subscribe_to_quorum_channel()

In case the experiment enforces a quorum, listen for notifications before creating Partipant objects.

3.12.2 Selenium bots

The *BotBase* provides a basis for a bot that interacts with an experiment using Selenium, which means that a separate, real browser session is controlled by each bot. This approach does not scale very well because there is a lot of overhead to running a browser, but it does allow for interacting with the experiment in a way similar to real participants.

By default, Selenium will try to run PhantomJS, a headless browser meant for scripting. However, it also supports using Firefox and Chrome through configuration variables.

```
webdriver_type = firefox
```

We recommend using Firefox when writing bots, as it allows you to visually see its output and allows you to attach the development console directly to the bot's browser session.

For an example of a selenium bot implementation, see the Bartlett1932 bots.

For documentation of the Python Selenium WebDriver API, see Selenium with Python.

API documentation

class dallinger.bots.**BotBase**(*URL*, *assignment_id=''*, *worker_id=''*, *participant_id=''*, *hit_id=''*)

A base class for bots that works with the built-in demos.

This kind of bot uses Selenium to interact with the experiment using a real browser.

complete_experiment(*status*)
Sends worker status ('worker_complete' or 'worker_failed') to the experiment server.

complete_questionnaire()
Complete the standard debriefing form.

Answers the questions in the base questionnaire.

driver
Returns a Selenium WebDriver instance of the type requested in the configuration.

participate()
Participate in the experiment.

This method must be implemented by subclasses of `BotBase`.

run_experiment()
Sign up, run the `participate` method, then sign off and close the driver.

sign_off()
Submit questionnaire and finish.

This uses Selenium to click the submit button on the questionnaire and return to the original window.

sign_up()
Accept HIT, give consent and start experiment.

This uses Selenium to click through buttons on the ad, consent, and instruction pages.

Scaling Selenium bots

For example you may want to run a dedicated computer on your lab network to host bots, without slowing down experimenter computers. It is recommended that you run Selenium in a hub configuration, as a single Selenium instance will limit the number of concurrent sessions.

You can also provide a URL to a Selenium WebDriver instance using the `webdriver_url` configuration setting. This is required if you're running Selenium in a hub configuration. The hub does not need to be on the same computer as Dallinger, but it does need to be able to access the computer running Dallinger directly by its IP address.

On Apple macOS, we recommend using Homebrew to install and run selenium, using:

```
brew install selenium-server-standalone
selenium-server -port 4444
```

On other platforms, download the latest `selenium-server-standalone.jar` file from SeleniumHQ and run a hub using:

```
java -jar selenium-server-standalone-3.3.1.jar -role hub
```

and attach multiple nodes by running:

```
java -jar selenium-server-standalone-3.3.1.jar -role node -hub http://hubcomputer.
↪example.com:4444/grid/register
```

These nodes may be on other computers on the local network or on the same host machine. If they are on the same host you will need to add `-port 4446` (for some port number) such that each Selenium node on the same server is listening on a different port.

You will also need to set up the browser interfaces on each computer that's running a node. This requires being able to run the browser and having the correct driver available in the system path, so the Selenium server can run it.

We recommend using Chrome when running large numbers of bots, as it is more feature-complete than PhantomJS but with better performance at scale than Firefox. It is best to run at most three Firefox sessions on commodity hardware, so for best results 16 bots should be run over 6 Selenium servers. This will depend on how processor intensive your experiment is. It may be possible to run more sessions without performance degradation.

3.13 Extra Configuration

To create a new experiment-specific configuration variable, define `extra_parameters` in your `experiment.py` file:

```
def extra_parameters():
    config.register('n', int, [], False)
```

Here, `'n'` is a string with the name of the parameter, `int` is its type, `[]` is a list of synonyms that be used to access the same parameter, and `False` is a boolean signifying that this configuration parameter is not sensitive and can be saved in plain text. Once defined in this way, a parameter can be used anywhere that built-in parameters are used.

CHAPTER 4

Core Contribution Documentation

These documentation topics cover setting up a development version of Dallinger, in order to contribute to the development of Dallinger itself. This is not needed in order to develop new experiments.

4.1 Releasing a new version of Dallinger.

The Dallinger branch *master* features the latest official release for 3.X.X, and *2.x-maintenance* features the latest official 2.X.X release.

1. After you've merged the changes you want into both *master* and *2.x-maintenance*, the branches are ready for the version upgrade. We're using semantic versioning, so there are three parts to the version number. when making a release you need to decide which parts should get bumped, which determines what command you give to *bumpversion*. *major* is for breaking changes, *minor* for features, *patch* for bug fixes. Example: Running *bumpversion patch*, which will change every mention of the current version in the codebase and increase it by *0.0.1*.

2. Log your updates by editing the CHANGELOG.md, where you'll link to your version's tree using: *https://github.com/dallinger/dallinger/tree/vX.X.X*. Mark the PR with the *release* label.

3. Merge this release with the commit "Release version X.X.X."

4. After that's merged, you'll want to tag the merge commit with *git tag vX.X.X* and do *git push origin –tags*. PyPI releases versions based on the tags via *.travis.yml*.

5. If you are releasing an upgrade to an old version, revert the PyPI change and make it show the highest version number. We do this because PyPI shows the last updated version to be the latest version which may be incorrect.

CHAPTER 5

General Information

5.1 Acknowledgments

Dallinger is sponsored by the Defense Advanced Research Projects Agency through the NGS2 program. The contents of this documentation does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

Dallinger's predecessor, Wallace, was supported in part by the National Science Foundation through grants 1456709 and 1408652.

5.2 Dallinger's incubator

Dallinger was one of the first scientists to perform experimental evolution. See his Wikipedia article for the specifics of his incubation experiments.

Python Module Index

d

Index

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W