EXPERt User Guide

Josh Fennell*

Contents

*josh@jfnl.org

1 Introduction

EXPERt is an application designed to support running web browser-based scientific experiments. It provides a built-in web server, within which experiments run as loadable bundles. These bundles specify the structure of an experiment using Python code, as well as providing view templates and JavaScript code to implement the browser-side experience for participants, along with any relevant media or data files. The code, templates, and scripts you provide can inherit from and build upon components built into EXPERt, making it possible to set up simple experiments with minimal effort.

2 Features

The following are some of the notable features provided by EXPERt:

2.1 Resumption of experiments

In-progress experiments can be stopped and resumed at a later time.

2.2 Replication of experiments

Participants are assigned pre-generated profiles that determine their experimental condition and any other parameters of their experience. This allows a new "run" of an experiment to be conducted using the exact same profiles as a previous run.

2.3 Real-time monitoring

EXPERt includes a dashboard for real-time monitoring of participant progress, including elapsed time, number of tasks completed, and completion status.

2.4 Flexible timeouts

EXPERt supports both a configurable global inactivity timeout and timeouts that can be applied to specific sections of an experiment.

2.5 Questionnaires

Support is included for questionnaires including multiple choice, short-answer, and long-answer questions, which can be optional or mandatory.

2.6 Adaptive experiment paths

Experiments don't have to be linear; experiments can consist of branching pathways, and a participant's path can change according to their responses.

3 Initial setup

EXPERt has been tested with Python version 3.9, but may work with some older versions.

Initial setup steps:

1. Install the PDM package manager, which is used to manage EXPERt's dependencies.
2. From within the `EXPERt` directory, execute the `pdm install` command, which will download and install all required build- and run-time dependencies.

EXPERt should now be ready to run the built-in example experiment.

4 Usage

This section provides an overview of how experiments in EXPERt are structured, as well as how to get an experiment running.

4.1 Experiment bundle

All files related to a given experiment are stored together in the **experiment bundle**, a folder which may be placed anywhere on the file system that EXPERt has read and write access to. The name of this folder will be used as the name of the experiment. The experiment bundle contains the essential code and templates required for every experiment, as well as any static resources to be served (images, audio files, etc.), auto-generated participant profiles, and the results of each experiment run. Any other data files required to run an experiment can also be placed in this folder. The path to this folder is accessible within your main experiment Python class (described in the next section) as `self.dir_path`.

4.2 Code

The Python code to configure an experiment is stored in the `src` folder within the experiment bundle. This folder must contain a file called `__init__.py`. This file contains the main class for your experiment; this class can be named as desired, but must be a subclass of `expert.experiment.Experiment`. An instance of this class will be created for every participant in the experiment. The `__init__.py` for the included example experiment contains comments explaining the structure of this file.

4.2.1 Tasks

EXPERt experiments are built from units called **tasks**, represented in the code by instances of the `expert.tasks.Task` class (or subclasses thereof). A task corresponds to a single screen where some action is performed and input (a **response**) is optionally collected. In the most basic type of experiment, tasks are simply presented in sequence. However, the response collected in a given task can be used to determine what the following task will be,

allowing for experiments that adapt as they progress.

The `Task` constructor has the following signature:

```
Task(inst, template=None, variables=None, timeout_secs=None)
```

where `inst` is the current participant's experiment class instance, `template` is the name of the template to associate with the task (corresponding to a file in the `templates` folder named `task_[template].html.jinja`; see below), and `variables` is a dictionary of task-specific data items (described below). `timeout_secs` specifies an optional global timeout clock that will start counting down once the participant reaches this task. If the timeout reaches 0 before the participant completes the experiment (or the timeout is disabled by a later task by setting its `timeout_secs` parameter to a negative value), the participant will time out.

Every `Task` instance has a property called `next_tasks` containing a list of tasks, one of which will be chosen to immediately follow the task in question. In a simple experiment, this list will contain at most one item. However, if the list contains multiple items, the default behavior is to treat the participant's response to the current task as an index into the list, selecting the next task from the available options.

To specifying the sequence of tasks in your experiment, you first create the initial task by instantiating `Task` (or a custom subclass) and assigning it to `self.task`. Subsequent tasks may then be added by chaining calls to `Task.then()` in the following manner:

```
self.task.then('welcome').then('instructions') ...
```

For convenience, the arguments to `.then()` are simply the arguments that would be passed to the `Task` (or subclass) constructor, excluding the initial `inst` parameter (the first argument may optionally be a subclass of `Task`). `.then()` can also be used to create dynamic,

forking paths through an experiment by NOT chaining calls. E.g.,

```
task_a.then('branch1')
task_a.then('branch2')
```

creates two possible tasks to follow `task_a`, using templates `branch1` and `branch2`, respectively.

`Task.then_all()` is a shortcut method for creating a linear sequence of tasks without having to chain multiple calls to `.then()`. `.then_all()` accepts a **sequence** of **task descriptors**. A task descriptor consists of either an instance of `expert.tasks.TaskDesc`, or a single argument (i.e., a template name) that will be passed on to `.then()`. An instance of `TaskDesc` simply encapsulates any positional and keyword arguments to pass to `.then()`, and is constructed as follows:

```
TaskDesc(posargs, kwargs)
```

All tasks have an associated set of variables that are exported to their template and script. Variables are set on a `Task` instance by passing a dictionary of items as the `variables` argument to the `Task` constructor or `.then()`. Task variables can be accessed within templates using "mustache" syntax: `{{variable_name}}`. In scripts, they are available from the `task.vars` object. All tasks are provided with the following variables:

- `exp_exper`: name of the experiment
- `exp_sid`: participant session ID
- `exp_prolific_pid`: Prolific participant ID (if using Prolific)
- `exp_prolific_completion_url`
- `exp_num_tasks`

- `exp_task_cursor`
- `exp_state`
- `task_type`: task template name
- `exp_tool_mode`
- `exp_url_prefix`
- `exp_static`: URL for built-in static file folder
- `exp_js`: URL for built-in script folder
- `exp_expercss`: URL for experiment bundle main.css file
- `exp_experimg`: URL for experiment bundle images folder
- `exp_window_title`: experiment window title
- `exp_favicon`: favicon filename
- `exp_progbar_enabled`: whether the progress bar is enabled

4.3 Templates

The displayed content for a task is specified using Jinja templates, placed in the `templates` subfolder of the experiment bundle. (The Template Designer Documentation describes the template syntax.) EXPERt comes with a number of built-in templates that can be extended to create tasks of various types.

4.4 Scripts

Each task type can have its own script (JavaScript file) to control how participants interact with the task. Scripts are stored in the `static/js` subfolder of the experiment folder. The

script file for a task named, e.g., 'foo', must be named `task_foo.js`. To load the script, the task template must contain the line:

```
{% set task_script = 'foo' %}
```

Every script must contain an exported class called `taskClass`, which must be a subclass of the `Task` JavaScript class, found in `/$(expert_url_prefix)/js/task.js`.

This class is instantiated when a task page is loaded in the browser. It should contain code to initialize the tasks's view. Importantly, if multiple tasks of the same type appear in sequence, the class is only instantiated for the very first such task. `Task.reset()` should be overridden to ensure that each such task in the sequence is properly initialized.

The `Task` instance receives the variables for a given task in its `vars` property. This is where you would find values such as the name of the current stimulus, as set on the `Task` objects in your Python code.

4.5 Other resources

Scripts, media files, stylesheets, and other resources are stored in the `static` subfolder of the experiment bundle. The stylesheets included with EXPERt were generated using Sass, the original source code for which is included in the top-level `sass` folder. However, the use of Sass is entirely optional in creating stylesheets for your own project.

4.6 Profiles

The `profiles` subfolder of the experiment bundle contains automatically generated participant profiles for each condition in the experiment. These profiles may specify, e.g., which stimuli participants in a given condition are exposed to, their order, and any other relevant

data. As participants arrive, they are assigned a random unused profile from the condition with the least amount of participants thus far.

It's important to note that profiles are only generated if none are already present in the experiment bundle. Thus, changing parameters within your experiment code or starting a new experiment run will not automatically result in new profiles being generated.

Further, only one set of profiles is ever stored in the experiment bundle. Before removing them to allow the generation of new profiles, it is wise to be certain that the existing set of profiles were not used for an important prior run of the experiment for which results exist.

4.7 Runs and responses

A **run** of an experiment consists of a set of response data files collected from all participants who took part in the experiment. Each response file is named with the profile assigned to a given participant. All runs are stored in the `runs` subfolder of the experiment bundle. Each **run folder** is named with a timestamp for when the run began. Within the run folder are the **condition folders**, which contain the results produced by the participants in each condition.

Responses may be stored in either CSV or JSON format. The fields in each file are as follows:

1. `tstamp`: timestamp when response was submitted (i.e., when the task was completed)
2. `task`: name of task
3. `resp`: actual task response; complex responses (e.g., questionnaire responses) are displayed as formatted Python data structures

Extra fields may also be present, if defined in the `resp_extra` field of the task instance; these may be useful for task-specific data such as the stimulus ID. Every response file also

includes the following pseudo-responses (with the given task names):

- `SID`: unique participant session ID
- `IPHASH`: hash of participant's IP address
- `USER_AGENT`: contents of participant's browser `User-Agent` header
- `PROLIFIC_PID`

Other pseudo-response data can also be included if desired.

4.8 Invocation

Before running EXPERt for the first time, it is necessary to execute the following shell command:

```
eval "$(pdm --pep582)"
```

This command correctly sets your shell's `PYTHONPATH` variable to allow python to access the dependencies that pdm has installed locally. Note, however, that these changes to `PYTHONPATH` do not persist beyond your current session, and you will need to run the command again if you log out or close your terminal.

Once `PYTHONPATH` has been set, EXPERt can be launched with the following command:

```
python src/runexp.py [experiment_path] (-l [listen_on]) (-u [url_prefix])
(-t) (-D) (-r [target] | -p [target]) (-c [conditions])
```

Arguments in parentheses are optional. `experiment_path` specifies the path to the experiment bundle. `-l` (or `--listen`) specifies the hostname or IP address and port number for the server to listen on, defaulting to `127.0.0.1:5000`. `-u` specifies the web server URL prefix (default: `expert`). `-t` runs EXPERt in Tool Mode (described below). `-D` causes

EXPERt to perform a Dummy Run (see below), rather than launching the web server. `-r` (`--resume`) and `-p` (`--replicate`) are mutually exclusive. They are provided along with a `target` (timestamp of experiment run to resume or replicate, which must match the name of a run folder). `-c` (`--conditions`) specifies a list of conditions (comma-separated, without spaces) from which participant profiles will be chosen.

When launched with default settings, the experiment will be available at the following URL:

`http://127.0.0.1:5000/expert`

The monitoring dashboard is accessible from:

`http://127.0.0.1:5000/expert/dashboard/[dashboard_code]`

where, `dashboard_code` is the code printed to the log when EXPERt was launched. The URL prefix, `expert` by default, can be customized (see Configuration, below).

4.9 Dashboard

The monitoring and control dashboard is accessible from:

`http://127.0.0.1:5000/expert/dashboard/[dashboard_code]`

where `dashboard_code` is a secret code specified via the `dashboard_code` configuration parameter (see section Configuration).

The primary feature of the dashboard is a list displaying the real-time status of participants in an experiment. However, several other features are available:

- Download Results: Download a zip file collecting the results of a particular run of an experiment.
- Download ID Mapping: Download a zip file containing a file for each participant con-

taining their individual anonymous session ID along with any experiment responses (or pseudo-responses) specified in the `pii` configuration parameter (see section Configuration) as containing personally-identifying information. Such responses are kept separate from the main response data, which only identifies participants by their session ID.

- Delete ID Mapping: Delete all ID mapping files for a run from the server.
- Start New Run: Immediately terminate the current run, and begin a new one.

4.10 Deployment

EXPERt provides its own web server, which is suitable for production use. However, it can also be run in reverse-proxy fashion in tandem with a more full-featured web server. An example configuration file is provided for the `nginx` server.

4.11 Configuration

Various global experiment parameters can be set by providing a file called `cfg.json` in the top level of the experiment bundle. Default settings for all configuration parameters may be found in `EXPERt/src/cfg.json`. The configurable parameters are as follows:

- `host`: host to listen on
- `port`: port to listen on
- `url_prefix`: initial component of URL
- `dashboard_code`: private code used to access the dashboard
- `favicon`: URL of bookmarks icon
- `output_format`: either `csv` or `json`

- `static_dir`: name of static files folder in experiment bundle
- `templates_dir`: name of templates folder in experiment bundle
- `profiles_dir`: name of profiles folder in experiment bundle
- `runs_dir`: name of run results folder in experiment bundle
- `dls_dir`: name of folder for downloadable files in experiment bundle
- `subjid_length`: length of profile names in characters
- `subjid_symbols`: string of characters to use when generating profile names
- `inact_timeout_secs`: duration of global inactivity timeout (separate from task sequence timeout described above); the inactivity timeout clock is reset every time the participant completes a task (i.e., presses the 'Next' button)
- `monitor_check_interval`: how long the background task that monitors participant progress sleeps between checks (specified in seconds)
- `save_ip_hash`: whether to store a pseudo-response with a hash of the participant's IP address
- `save_user_agent`: whether to store a pseudo-response with the participant's user-agent string
- `prolific_pid_param`: name of the Prolific participant ID URL parameter
- `prolific_completion_url`: return-to-Prolific URL displayed at the end of an experiment
- `pii`: list of task names considered to contain personally-identifying information
- `progbar_enabled`: whether the experiment progress bar should be displayed
- `tool_display_total_tasks`: whether the total number of tasks should be displayed in Tool Mode