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 plug-in modules. These modules 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 folder

All files related to a given experiment are stored together in the **experiment 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 folder 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 folder. 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(sid, template=None, variables=None, timeout_secs=None)
```

where `sid` is the session ID of the current participant (available in your experiment class as `self.sid`), `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 `sid` 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:

- `exper`: name of the experiment
- `sid`: participant session ID
- `task_type`: task template name

4.3 Templates

The displayed content for a task is specified using Jinja templates, placed in the `templates` subfolder of the experiment folder. (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 function called `initTask`, declared as follows:

```
export function initTask(task) { ... }
```

This function is called whenever a new task begins. It should contain code to initialize the tasks's view. Importantly, if multiple tasks of the same type appear in sequence, the task template is only fully loaded into the browser for the very first such task. However, `initTask` is still called at the start of each subsequent task of the same type, and must take care to properly reset any onscreen items that should not have their state persist between tasks.

`initTask` receives the variables for a given task in the `vars` attribute of the `task` object. 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 folder. 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 folder 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.

4.7 Runs and responses

All runs of an experiment are stored in the `runs` subfolder of the experiment folder. 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. The individual response files within each condition folder are named with the name of the profile assigned to the corresponding participant.

Responses are stored in CSV format. The columns in each file are as follows:

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

displayed as formatted Python data structures

4. `extra`: content of `resp_extra` field of task instance; any other task-specific data, e.g., stimulus ID, stimulus group

Every response file 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

Other pseudo-response data can also be included if desired (e.g., Mechanical Turk completion code).

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]) (-r [target]
| -p [target]) (-c [conditions])
```

Arguments in parentheses are optional. `experiment_path` specifies the path to the experiment folder. `-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`. `-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.

4.9 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.

4.10 Configuration

Various global experiment parameters can be set by modifying `expert/globalparams.py`. Notable parameters include the following:

- `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)