Bride of Frankensystem

Release 2.0

Colby Johanson

Jul 02, 2026

CONTENTS

Bride of Frankensystem (BOFS) is an open-source framework for online behavioral experiments and surveys. It sits between survey-only platforms (Qualtrics, SurveyMonkey) and building from scratch: editing TOML, JSON, and HTML gets you a running study, and Python is available when you need it.

CHAPTER

ONE

WHAT IS BOFS?

Bride of Frankensystem (BOFS) is an open-source framework for building online behavioral experiments and surveys. Instead of a drag-and-drop editor, you describe your study in plain-text files — a configuration file for settings and page flow, JSON files (a structured plain-text format) for questionnaires, and HTML files for custom pages. BOFS handles participant routing, condition assignment, consent forms, data storage, and provides an admin panel for monitoring and export.

1.1 Where BOFS fits

If you have used other tools for online studies, BOFS sits at a specific spot in the landscape.

- **Survey platforms (Qualtrics, SurveyMonkey, Google Forms)** handle questionnaires with point-and-click editing on hosted infrastructure. They cannot embed a JavaScript task or open the data layer for custom logic. If your study is questionnaires only and your institution already provides Qualtrics, that may be a simpler choice.
- **JavaScript experiment libraries (jsPsych, lab.js, PsychoJS)** handle in-browser trial logic — precise timing, key capture, randomization. They do not host the surrounding study (consent, condition assignment, sessions, admin panel). A jsPsych, lab.js, or PsychoJS task can run inside a BOFS custom page; the two are complementary.
- **Custom Flask, Django, or Express applications** (general-purpose web frameworks) give you full control of every detail, and full responsibility for everything else. When BOFS's built-in patterns are not enough, it exposes the same Flask underneath — custom routes drop into the same project.

BOFS sits in the middle: research-specific scaffolding (consent, conditions, sessions, data storage, admin panel) plus an open boundary for questionnaires, JavaScript tasks, or Python where each is the right fit.

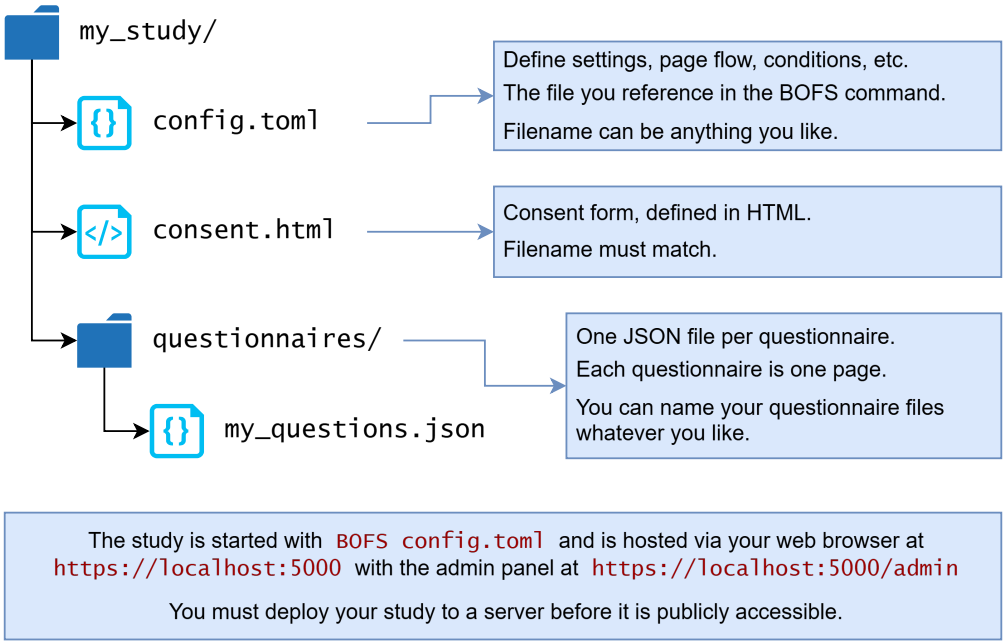
1.2 How BOFS development works

A BOFS project moves through three stages:

1. **Develop locally.** Build and run the experiment on your own machine. The project is a folder of configuration and content files you can edit freely.
2. **Test and debug.** Preview the experiment exactly as a participant will see it. The admin panel and debug tools surface errors before you go live.
3. **Deploy to a server.** Copy the project to a web server when you are ready to recruit participants. See *Deploying to a Server*.

1.3 A simple project

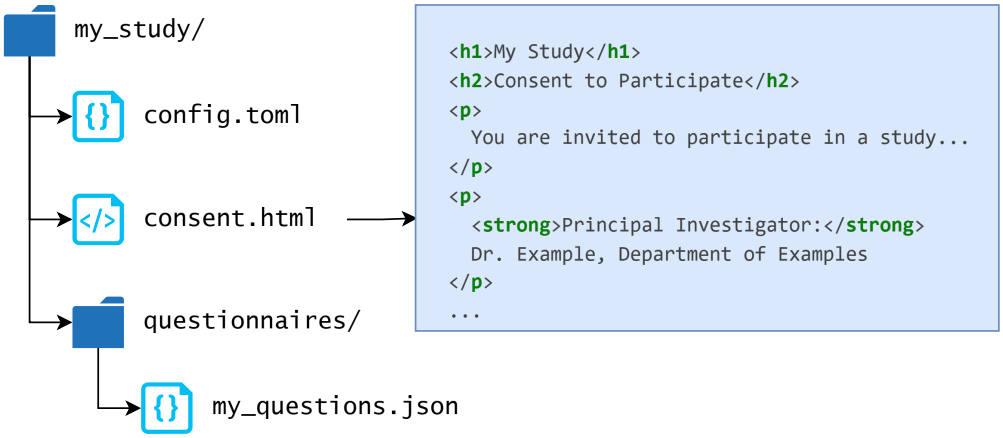
A BOFS project is a collection of files, each controlling one aspect of the experiment. The simplest BOFS project is a folder with three pieces:



1.3.1 The pieces

This file structure is all that is needed for questionnaire-based projects. It allows you to do the following:

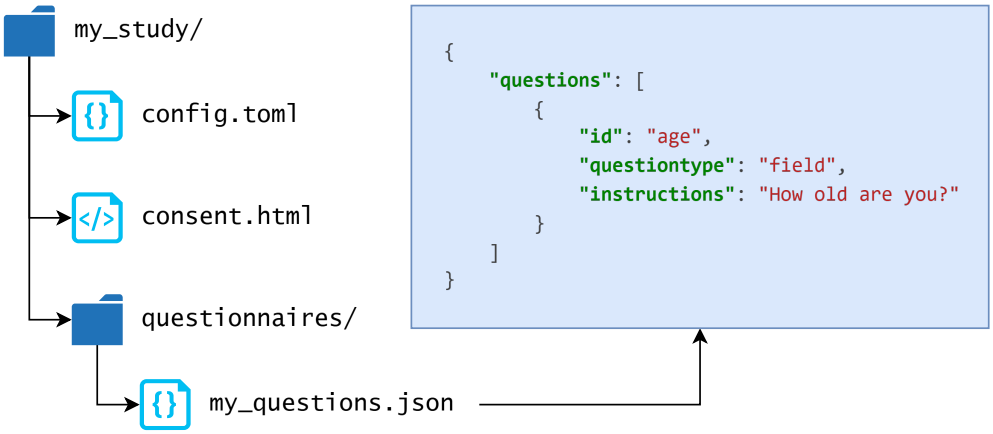
Show a consent form



The consent form is an HTML file at the project root. BOFS wraps it in a form with “I consent” and “I do not consent” buttons, records the response, and creates the participant entry in the database.

See *Consent Forms* for the consent flow, route variants, multi-stage consent, and IRB notes.

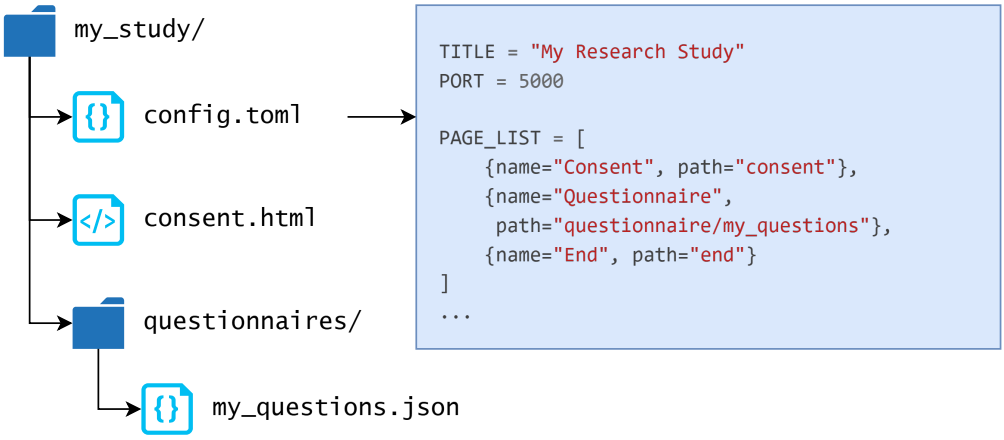
Ask questions



Questionnaires are JSON files — one file per page of questions. BOFS renders the form, validates required fields, and stores the responses automatically.

See *Adding Survey Questions* for question types, conditional visibility, and more.

Control the page order



`PAGE_LIST` in `config.toml` defines the sequence of pages a participant moves through, along with settings like the study title, port, and database. Add, remove, or reorder pages by editing the list.

See *Setting Up Your Page Flow* for required settings, page types, and running the same questionnaire multiple times.

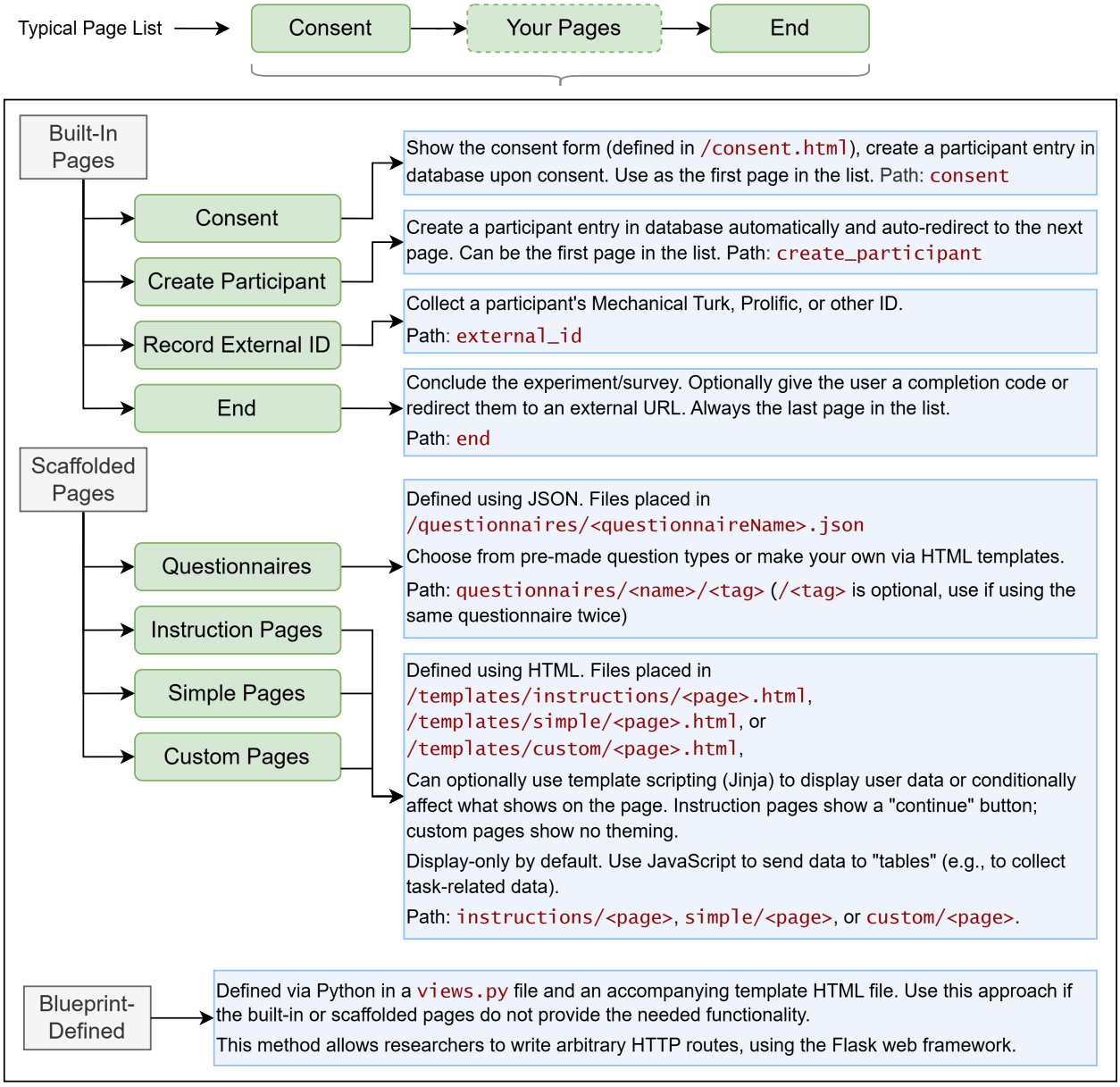
Even with these three files, BOFS already covers common research needs:

- **Random assignment.** Define groups in your config; BOFS assigns each participant and keeps the groups balanced as they enroll.
- **Item-order randomization.** Set "shuffle" on a rating grid or radio list to present items in a random order for each participant, controlling for position effects without any code.
- **Conditional questions.** Hide a question — or a whole page — when an earlier answer makes it irrelevant. Participants only see what applies to them.
- **Reusing earlier answers.** Embed a previous response into a later question's wording — for instance, show participants the rating they just gave and ask them to explain it.
- **Computed scores.** Sum a Likert block, reverse-score specific items, or define any calculation over a page's responses; BOFS evaluates it as the participant submits.

- **Repeated measurements.** Present the same questionnaire more than once — for pre/post within a single session. Or, have participants come back days or weeks later, and load their condition assignment or progress through the study based on an external ID.
- **Live monitoring.** The admin panel shows where each participant currently is, in real time.
- **Descriptive statistics on demand.** The admin panel shows N, mean, standard deviation, and median for every numeric question — broken down by condition — along with box plots, useful for inspecting pilot data without exporting to a stats package.

1.4 The page types

Every BOFS experiment is a sequence of pages defined in `PAGE_LIST` (inside `config.toml`). A typical experiment starts with consent, moves through your pages, and ends:



The diagram above shows all available page types, organized into three tiers:

Built-in Pages Require no files — just add them to `PAGE_LIST`. BOFS handles consent forms, participant creation, external ID collection, and experiment completion.

No code or markup required.

Scaffolded Pages Require an additional file to add. For questionnaires, a JSON file. For instruction, simple, or custom pages, an HTML file. BOFS automatically provides navigation, automatic validation and data storage, depending on the type of page.

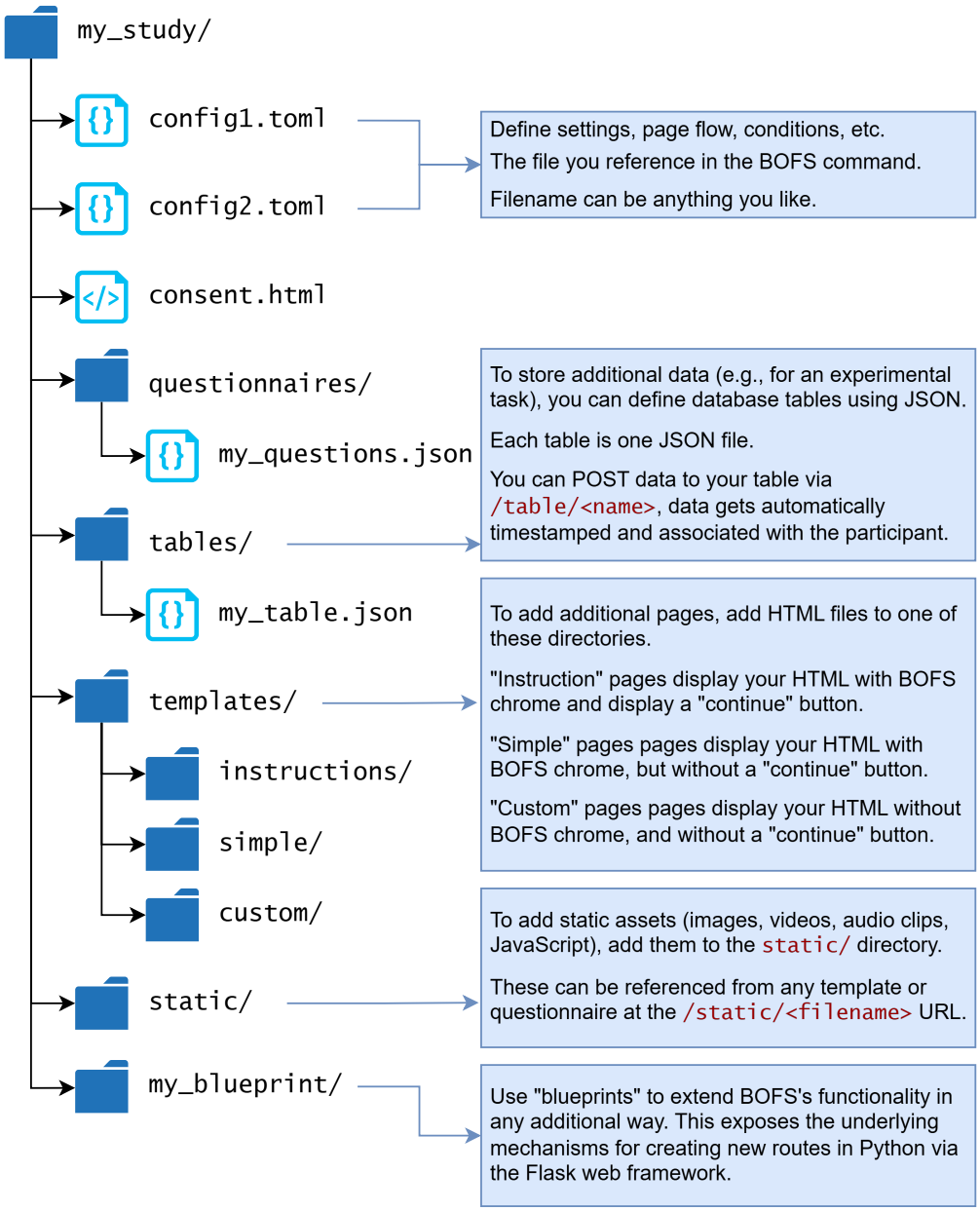
Some basic markup is required. To make full use of simple or custom pages, you can use HTML and JavaScript.

Blueprint-Defined Pages Require an additional directory and specific files to add to your project. For when the built-in and scaffolded options are not enough. You write Python code that responds at URLs you define — and BOFS auto-discovers them.

Knowledge of Python, HTML, and JavaScript is an asset.

1.5 More pieces

The simple project's three files cover questionnaire-based studies. To go past that, you add directories to the same folder — nothing about the simple project has to change.



Add your own pages

```
<h2>Welcome</h2>
<p>In this study you will read short scenarios and answer
   a few questions about each one.</p>
<p>Click <strong>Continue</strong> when you are ready.</p>
```

HTML files in `templates/instructions/`, `templates/simple/`, or `templates/custom/` become pages in your study. Instruction pages get an automatic Continue button; simple pages give you full control over navigation; custom pages drop the BOFS chrome entirely so you can host an embedded JavaScript task.

See *Adding Your Own Pages* for the differences between the three and how to wire navigation.

Serve static files

```
static/
├── stimulus_a.png
├── stimulus_b.png
├── instructions.pdf
└── p5.min.js
```

Files in `static/` are served at `/static/<path>`. Use it for images, audio, video, downloadable PDFs, and any JavaScript libraries your custom pages depend on.

See *Adding Your Own Pages* for embedding static assets in instruction, simple, and custom pages.

Store custom data

```
{
    "columns": {
        "score": "integer",
        "reaction_time": "float",
        "stimulus": "string"
    }
}
```

Custom tables are JSON-defined database tables for the data your study generates beyond questionnaire responses — task scores, trial-by-trial events, mouse movement. JavaScript on a page POSTs to `/table/<name>`; BOFS handles storage and exposes the rows in the admin panel.

See *Storing Custom Data* for column types, the JS read/write API, and calculated export fields.

Add Python routes

```
from flask import Blueprint, render_template
from BOFS.util import verify_correct_page

my_blueprint = Blueprint('my_blueprint', __name__,
                         template_folder='templates')

@my_blueprint.route('/task')
@verify_correct_page
def task():
    return render_template('my_blueprint/task.html')
```

A blueprint — Flask's term for a self-contained bundle of routes — is a folder at your project root containing `views.py` (and optionally its own `templates/`, `static/`, or `tables/`). BOFS auto-discovers it and loads the routes you define — for when you need server-side logic, custom database queries, or anything else beyond what scaffolded pages can do.

See *Blueprints and Routes* for blueprint layout, route decorators, and writing to custom tables from Python.

1.6 Where to go next

Read *Install BOFS* next to install BOFS, then *Initialize Your First Project* to generate a project with the `BOFS init` wizard and run it.

1.6.1 Install BOFS

The short version

If you already know your way around Python:

1. **Install Python 3.9+.**
2. **Install the framework:** `pip install bride-of-frankensystem`
3. **Test it:** run `BOFS` — you should see a help message.

That's the whole installation. If terms like `pip` or "virtual environment" are new to you, skip the short version — Steps 1 and 2 below walk through the same process and explain each command as it appears. Once BOFS is installed, *Initialize Your First Project* covers creating and running your first project.

BOFS runs on Windows, Mac, and Linux.

> **Note**
>
> To install the latest development release (which may be unstable), install the `bride-of-frankensystem-dev` package instead.

Step 1: Install Python

BOFS requires Python 3.9 or newer. The commands in this guide are typed into a terminal — a text window where you type a command and press Enter to run it. Check what's already installed:

- **Windows**: Open Command Prompt (search "cmd") and run `python --version`.
- **Mac/Linux**: Open Terminal and run `python3 --version`.

If you see `Python 3.9.x` or higher, continue. Otherwise, download Python from https://python.org.

Step 2: Install BOFS

Install BOFS into an isolated environment so its dependencies don't conflict with other Python projects on your machine. Pick whichever toolchain you're more comfortable with — both arrive at the same result.

> **Note**
>
> If you only ever plan to work on a single BOFS project, you can skip the isolated environment and install BOFS into your system Python with `pip install bride-of-frankensystem`. Working on multiple projects this way can cause dependency conflicts down the line.

pip + venv

The standard approach using Python's built-in tools.

2.1 Open your command line.

- **Windows**: Command Prompt or PowerShell
- **Mac**: Terminal (via Spotlight)
- **Linux**: your terminal application

2.2 Create the virtual environment.

```
python -m venv bofs_venv
```

This creates a `bofs_venv` folder containing its own copy of Python. Anything you install while the environment is active goes here instead of into your system Python, so different projects can use different package versions without conflicting.

2.3 Activate it.

- **Windows (Command Prompt)**: `.\bofs_venv\Scripts\activate.bat`
- **Windows (PowerShell)**: `.\bofs_venv\Scripts\Activate.ps1`
- **Mac/Linux**: `source bofs_venv/bin/activate`

Your prompt should now be prefixed with `(bofs_venv)`. Subsequent `python` and `pip` commands will use the virtual environment. Activate it again each time you open a new terminal.

2.4 Install BOFS.

```
pip install bride-of-frankensystem
```

`pip` is Python's package installer. This downloads BOFS and its dependencies from the Python Package Index. It may take a minute or two.

2.5 Test the installation. Run `BOFS`. You should see a help message listing the available commands.

uv

uv is a Python package manager from Astral. Its `uv tool` command installs Python applications into isolated environments and adds them to your PATH, so you don't need to activate anything before running BOFS.

2.1 Install uv. Follow the uv installation instructions for your operating system.

2.2 Install BOFS as a tool.

```
uv tool install bride-of-frankensystem
```

This creates a dedicated environment for BOFS behind the scenes and makes the `BOFS` command available globally. Open a new terminal afterwards so the updated PATH (the list of folders your system searches when you type a command) takes effect.

2.3 Test the installation. Run `BOFS`. You should see a help message listing the available commands.

Next step

Continue to *Initialize Your First Project* to create your first project with `BOFS init`.

Troubleshooting

- **"python not found"** — Make sure Python is installed and added to your system PATH (the list of folders your system searches when you type a command; the Python installer on Windows has an "Add to PATH" checkbox).
- **"pip not found"** — On Windows, allow the installer to add Python to your PATH. On Linux, `pip` may be a separate package.
- **"BOFS not found"** — If you used pip + venv, make sure the virtual environment is activated. As a fallback, `python -m BOFS run config.toml` is equivalent to `BOFS run config.toml`.
- **"Permission denied"** — Try your command line as administrator (Windows) or prefix the command with `sudo`, which runs it with administrator privileges (Mac/Linux).

1.6.2 Initialize Your First Project

This page assumes BOFS is installed — and, if you used pip + venv, that the virtual environment is active. See *Install BOFS* if not.

Step 1: Create a project with `BOFS init`

BOFS includes an interactive wizard that creates new projects. From your terminal, run:

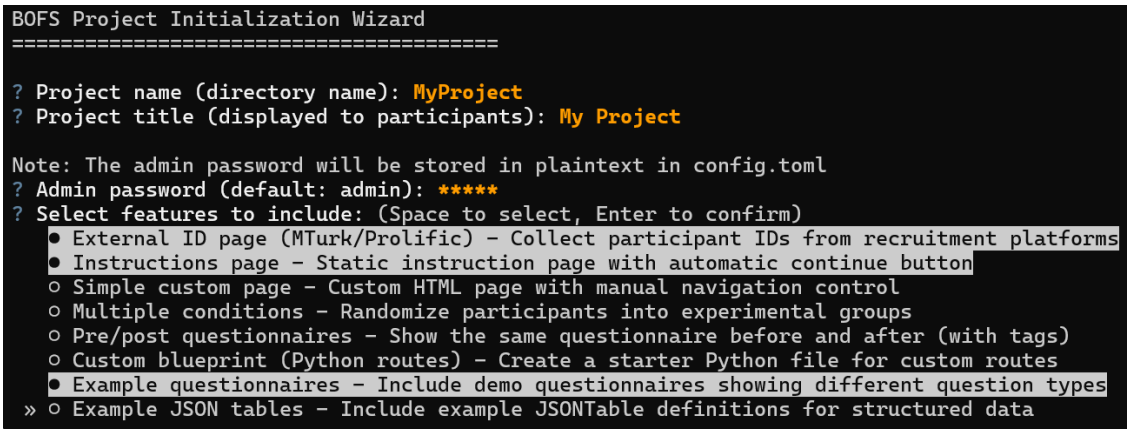
```
BOFS init
```

The wizard prompts you for:

1. **Project name** — the directory name (e.g., `my_experiment`).
2. **Project title** — what participants see in the page header (e.g., `My First Experiment`).
3. **Admin password** — for logging into `/admin`.
4. **Features** — toggle with Space, confirm with Enter.

For this walkthrough, select these features:

- ☑ External ID page (MTurk/Prolific)
- ☑ Instructions page
- ☑ Example questionnaires



When you confirm, the wizard creates the project and offers to start it for you. Choose **Yes** to launch the server and open your browser automatically.

Step 2: What the wizard generated

```
my_experiment/
├── config.toml                    # main configuration file
├── consent.html                   # consent form content
├── questionnaires/
│   ├── survey.json
│   ├── demographics.json
│   └── feedback.json
└── templates/
    └── instructions/
        └── welcome.html           # instructions page content
```

config.toml holds settings and the page flow:

```
SQLALCHEMY_DATABASE_URI = 'sqlite:///my_experiment.db'
TITLE = 'My First Experiment'
ADMIN_PASSWORD = 'admin'
PORT = 5000

PAGE_LIST = [
    {name='Consent', path='consent'},
    {name='External ID', path='external_id'},
    {name='Instructions', path='instructions/welcome'},
    {name='Survey', path='questionnaire/survey'},
    {name='End', path='end'},
]
```

`PAGE_LIST` defines what participants see and in what order. `SQLALCHEMY_DATABASE_URI` tells BOFS where to store participant data — the default is SQLite, a database kept in a single file (here `my_experiment.db`) inside your project folder. See *Setting Up Your Page Flow* to add or rearrange pages, and *Configuration Reference* for every available setting.

consent.html holds the consent text:

```
<h1>My First Experiment</h1>
<h2>Consent to Participate</h2>
<p>Welcome to this study. Please read the following information carefully.</p>
<!-- ... fill in purpose, procedures, contact info, IRB number ... -->
```

Edit this file with your real consent text before running a study with participants. See *Consent Forms* for the full picture, including the four first-page route variants.

questionnaires/survey.json is one of three example questionnaires:

```
{
    "title": "Survey",
    "instructions": "Please answer the following questions.",
    "questions": [
        {
            "questiontype": "radiogrid",
            "id": "agreement",
            "labels": ["Strongly disagree", "Disagree", "Neutral", "Agree", "Strongly␣
↪agree"],
            "questions": [
                {"id": "item1", "text": "Statement 1"},
                {"id": "item2", "text": "Statement 2"}
            ]
        }
    ]
}
```

See *Adding Survey Questions* to write your own.

Step 3: Run the project

If the wizard didn't start the project for you, start it manually:

```
cd my_experiment
BOFS run config.toml -d
```

The `-d` flag enables debug mode, which adds a debug toolbar at the bottom of every page and surfaces more detailed error messages. You'll see startup output like the following — the "blueprint" lines are BOFS loading its internal components:

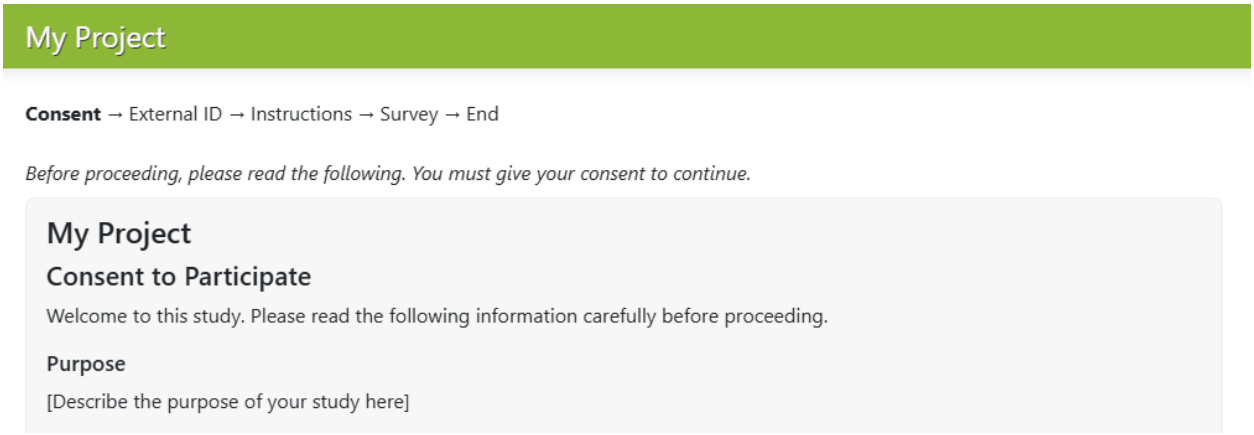
```
Loading blueprint: BOFS.admin
Loading blueprint: BOFS.default
BOFS.default: `models.py` loaded!
Listening on http://0.0.0.0:5000
Preview locally at http://127.0.0.1:5000
```

Open http://localhost:5000 in your browser — `localhost` and `127.0.0.1` both mean "this computer," so the address only works on the machine running BOFS. To stop the server, press **Ctrl+C** in the terminal.

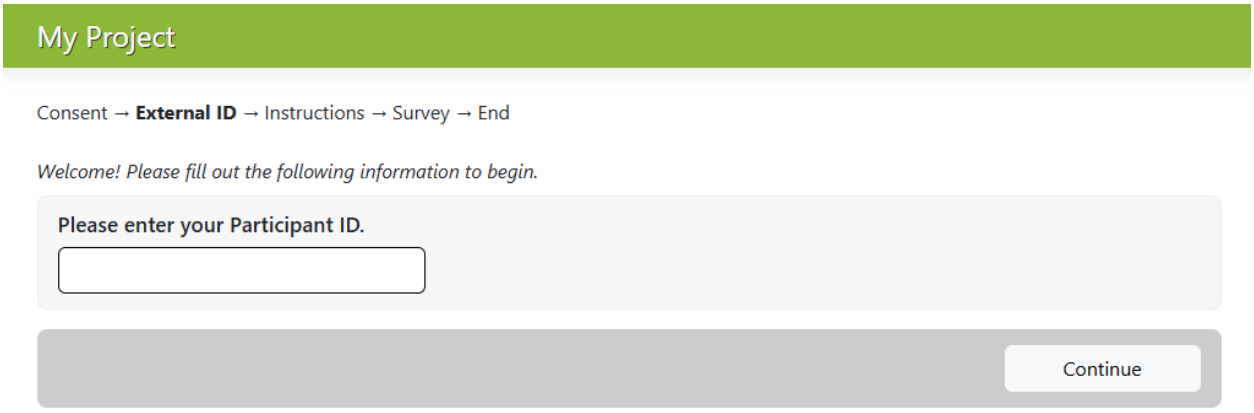
Step 4: Walk through the experiment

Each page below is one stop in the participant's journey through the project the wizard generated.

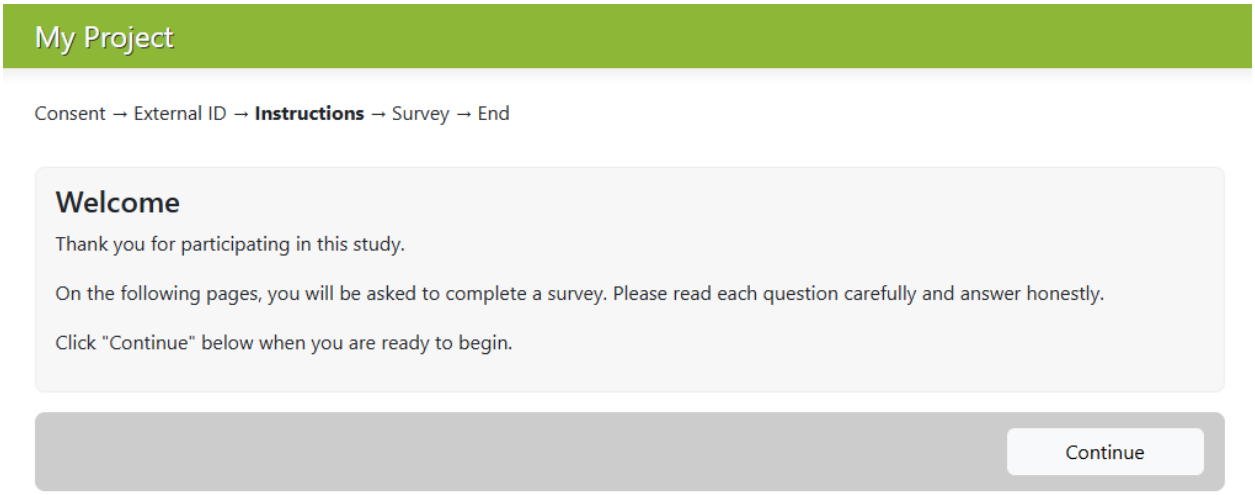
Consent page. The first page displays your consent form. Participants click "I Agree" to continue.



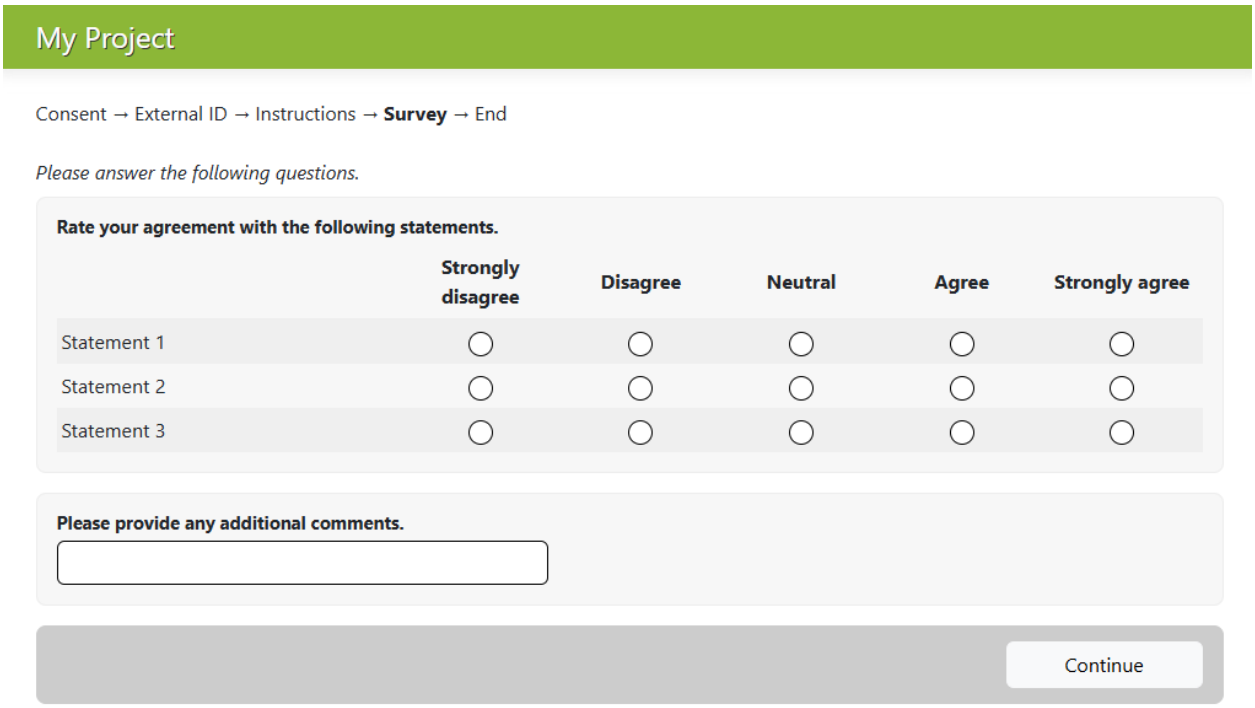
External ID page. Collects a participant ID, useful when recruiting from MTurk or Prolific. The prompt is configurable via `EXTERNAL_ID_PROMPT` in `config.toml`.



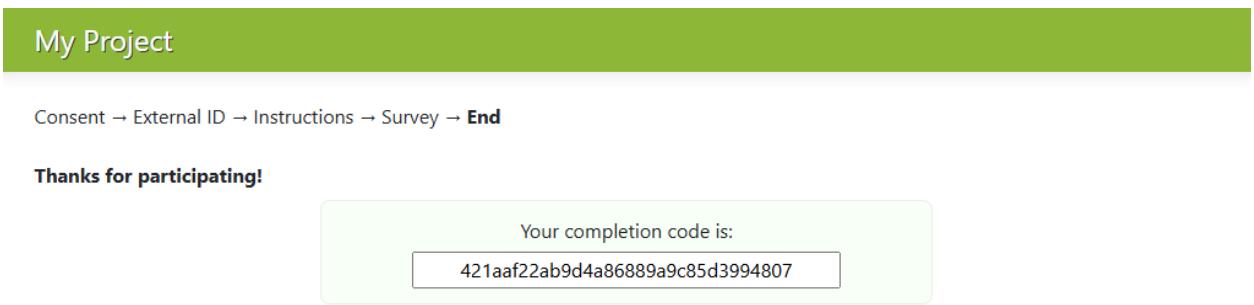
Instructions page. Renders the HTML in `templates/instructions/welcome.html`.



Survey page. Renders questions from the JSON questionnaire. Required questions block submission until they're answered.

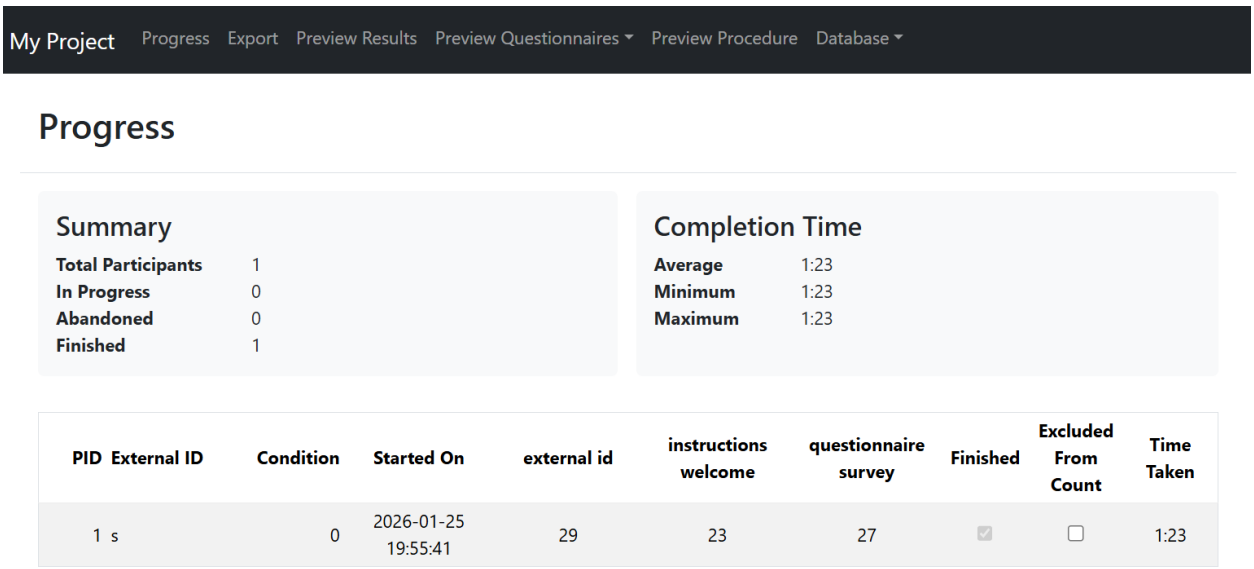


End page. Shows a generated completion code participants can use to verify participation.



The admin panel

Every BOFS project has an admin panel at `/admin` (so http://localhost:5000/admin for this project). Log in with the password you set in the wizard.



The admin panel shows live participant progress, exports the collected data as CSV, previews questionnaires without creating a participant record, and lets you browse the underlying database. Full details: *Monitoring and Exporting Data*.

Development tips

- **Replay the experiment.** Use a private/incognito window, or visit `/restart` to clear your session (BOFS's record of your progress through the study) and start over.
- **Debug toolbar.** The `-d` flag adds a toolbar at the bottom of every page with navigation controls and session information.
- **What restarts.** HTML and JSON changes reflect on the next page load. Editing `config.toml` requires restarting the server (Ctrl+C, then `BOFS run config.toml -d` again).

Where to go next

Three forks, depending on what you want to build:

- **Surveys, instruction pages, conditions** — read *Adding Survey Questions*, *Setting Up Your Page Flow*, and *Conditions and Branching* in order.
- **A JavaScript task** (p5.js, jsPsych, lab.js, PsychoJS, Unity) — read *Quickstart: Integrating a Custom Task* next.
- **See what else is possible** — *Quickstart: Integrating a Custom Task* doubles as a tour of BOFS's extensibility.

When you're ready to put the project in front of real participants, see *Deploying to a Server*.

Troubleshooting

- **"Address already in use"** — Another program is using the same port (the numbered channel a server listens on; BOFS defaults to 5000). Edit `config.toml` and set a different `PORT`.

1.6.3 Quickstart: Integrating a Custom Task

The previous page (*Initialize Your First Project*) got you to a working survey. This page is a tour of what BOFS can do beyond surveys — the patterns for embedding a JavaScript task, storing data the task generates, and stitching it into the page flow.

There's no step-by-step build here; the goal is to show you the shape of a custom-task project so you can pick the example that fits what you're trying to build and read the source. The examples live in the BOFS examples repository — download or clone it to run them locally.

The general pattern

A custom task in BOFS is three pieces:

- **A custom page** that hosts the task's HTML and JavaScript. BOFS serves your HTML from the project's `templates/` directory, and any static assets (libraries, images) from the project's `static/`.
- **A custom database table** that gives the task a place to write its data. You define the table in a small JSON file; BOFS creates the underlying SQL table at startup. The task sends its data to `/table/<name>` with a POST (a standard web request that carries data to the server) and the row appears in the database.
- **A** `PAGE_LIST` **entry** that puts the task page in sequence with consent, instructions, post-task questionnaires, and the end page.

The same three pieces compose for any in-browser task. There are two shapes of JavaScript library that fit:

- **Experiment frameworks** — jsPsych, lab.js, and PsychoJS. These ship the trial-loop scaffolding (timing, randomization, key capture, data accumulation). You write a timeline; the framework runs it. The integration with BOFS is the same in each case: include a copy of the library under your project's `static/` directory, host the task on a custom page, POST the per-trial data to a custom table at the end.
- **Stimulus and visualization libraries** — p5.js, D3.js, Three.js, plain Canvas/WebGL. These render and respond to input but do not impose a trial-loop structure. You write the trial logic yourself in vanilla JavaScript. Examples: `p5_example` for a five-second click counter, `ab_experiment` for a between-subjects D3.js menu task.

Unity WebGL is a third shape — a self-contained game build that is embedded as an asset rather than a JavaScript library.

What changes between examples is which library you load and what the per-trial data looks like.

The p5.js example

`p5_example` (in the BOFS examples repository) is the smallest illustration. The participant clicks as many times as they can in five seconds; the score gets posted to a custom table; the next page advances automatically.

The task page lives under `templates/simple/my_task.html` and is two script tags plus a `<main>` for p5 to attach a canvas to:

```
<script src="/static/p5.min.js"></script>
<script src="/static/my_task.js"></script>
<main></main>
```

Files in `templates/simple/` are served at `/simple/<filename>` (no `.html`), so this is reachable at `/simple/my_task`. The page is wrapped in BOFS's standard header, breadcrumbs, and styling — to drop that surrounding interface (typical for fullscreen tasks), use `templates/custom/` instead, served at `/custom/<filename>`.

The task script in `static/my_task.js` runs the click counter and, after five seconds, POSTs the result and advances:

```
fetch("/table/my_task", {
    method: "POST",
    headers: {"Content-Type": "application/json"},
```

(continues on next page)

(continued from previous page)

```
    body: JSON.stringify({ score: score })
}).then(function () {
    window.location.href = "/redirect_next_page";
});
```

`fetch` is JavaScript's built-in function for making web requests. `/table/my_task` is a built-in route that writes the posted data into the `my_task` custom table. `/redirect_next_page` is also built-in: it looks up the participant's current position in `PAGE_LIST` and sends them to whatever comes next, so you don't hard-code the URL of the page after the task.

The table definition in `tables/my_task.json` declares one column and two summary values computed across the participant's rows:

```
{
  "columns": {
    "score": { "type": "integer", "default": 0 }
  },
  "exports": [
    {
      "fields": {
        "average_score": "avg(score)",
        "high_score": "max(score)"
      }
    }
  ]
}
```

BOFS creates the `my_task` table at startup. The export fields show up per-participant in the admin export and are also usable in page-level `show_if` conditions as `tables.my_task.average_score` and `tables.my_task.high_score`.

`PAGE_LIST` ties consent, instructions, the task, and the end page together:

```
PAGE_LIST = [
    {name='Consent', path='consent'},
    {name='Instructions', path='instructions/task_instructions'},
    {name='Task', path='simple/my_task'},
    {name='End', path='end'}
]
```

That's the entire integration. No Python.

Other integration patterns

The example repository covers four more shapes of custom task. The PAGE_LIST and questionnaire layout are similar across them; what differs is which JavaScript library runs the trials and what the per-trial data looks like.

- **jsPsych** — the jspsych_example runs a Stroop task with jsPsych. Trial timing and key capture are handled by jsPsych; BOFS handles questionnaires, condition assignment, and storage. Per-trial data is POSTed in a single batch to `/table/jspsych_trials`. A copy of the jsPsych library is included under `static/jspsych/` so the example runs offline.
- **lab.js** — the labjs_example is the parallel of the jsPsych example with lab.js instead. The `PAGE_LIST` and questionnaires are identical; only the trial-running framework and the per-trial data shape change.
- **PsychoJS** — the psychojs_example wraps a Stroop task built in PsychoPy Builder and exported as PsychoJS. If you built your experiment in Builder, BOFS takes over the roles Pavlovia would otherwise play: it hosts the task,

supplies the participant ID, and receives the trial data. The Builder export is included verbatim apart from three marked edits: paths under `static/psychojs/`, skipping the participant info dialog (BOFS already knows the participant ID), and replacing `quitPsychoJS` to POST trial rows to `/table/psychojs_trials` and advance the BOFS page flow. A copy of the PsychoJS bundle is included under `static/psychojs/lib/`.

- **Unity WebGL** — the unity_example_2021.1 and unity_example_2023.2 projects host a Unity WebGL build inside a BOFS page. They demonstrate three layouts (BOFS-chrome, fullscreen, fully custom), pushing the participant ID into the running build, reading the assigned condition from inside Unity, posting data back to a custom table, and advancing the BOFS page flow from within Unity.
- **Embedded media** — the embedding_media_example shows every place BOFS can embed an image or video: in custom HTML, in questionnaire `instructions` fields, in `textview` questions, and as their own `video` question with optional "force watch" enforcement.

Where to learn more

To build the rest of an experiment around your task:

- *Adding Survey Questions* — pre/post-task questionnaires, question types, conditional questions.
- *Setting Up Your Page Flow* — adding pages, repeating questionnaires with different `tag` values, using multiple `.toml` files for dev/production splits or running several experiments out of one project.
- *Conditions and Branching* — A/B and multi-arm experiments, conditional routing, page-level `show_if`.
- *Storing Custom Data* — going deeper on custom tables, including JS read/write and Python access.
- *Monitoring and Exporting Data* — admin panel, exports, results.

If you want to write Python on top of BOFS — custom routes, server-side stimulus generation, complex data processing — see *The BOFS Architecture* and *Blueprints and Routes*.

CHAPTER

TWO

BUILDING YOUR EXPERIMENT

How-to guides for assembling a study: pages, questions, conditions, custom data, appearance, and monitoring. These pages cover all that is possible with BOFS without needing to write any additional server-side Python code.

If you're building your first study, read *Adding Survey Questions* and *Setting Up Your Page Flow* first — the rest can be read in any order as your study needs them.

2.1 Adding Survey Questions

Questionnaires in BOFS are JSON files — plain text in a structured `"name": value` format, editable in any text editor — stored in your project's `questionnaires/` directory. Each file is one questionnaire and renders as one page in the experiment. For multiple pages of questions, create multiple files.

Questionnaires can also live inside a blueprint at `<blueprint_name>/questionnaires/`. See *Blueprints and Routes* for how blueprint-scoped questionnaires are discovered.

2.1.1 A minimal example

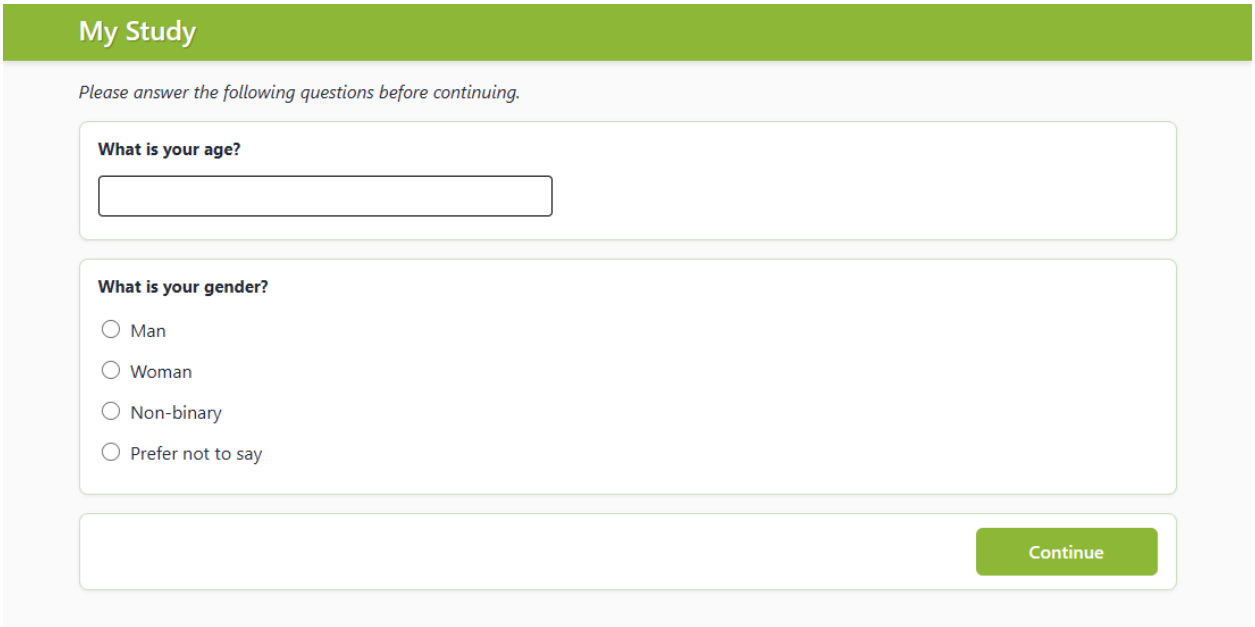
The following file — `questionnaires/demographics.json` — collects age and gender:

```
{
    "title": "Demographics",
    "instructions": "Please answer the following questions before continuing.",
    "questions": [
        {
            "id": "age",
            "questiontype": "num_field",
            "instructions": "What is your age?",
            "required": true
        },
        {
            "id": "gender",
            "questiontype": "radiolist",
            "instructions": "What is your gender?",
            "labels": ["Man", "Woman", "Non-binary", "Prefer not to say"]
        }
    ]
}
```

Every question has three required properties:

- `id` — a unique identifier that becomes a column name in the database and the column header in your exported CSV data. Use lowercase letters and underscores (e.g. `my_question`); avoid words that have special meaning in databases or Python, such as `order`, `select`, `class`, or `return`.
- `questiontype` — which question type to render (see below).
- `instructions` — the question text shown to participants (HTML is accepted).

`required` is optional and defaults to `false`.



2.1.2 Common question types

Single-line text (`field`)

Renders a single text input. Use `num_field` for numeric-only input.

```
{
    "id": "occupation",
    "questiontype": "field",
    "instructions": "What is your occupation?"
}
```

Single choice from a list (`radiolist`)

Renders a vertical list of radio buttons. Each entry in `labels` is one option.

```
{
    "id": "education",
    "questiontype": "radiolist",
    "instructions": "What is the highest level of education you have completed?",
    "labels": [
        "Less than high school",
        "High school diploma or equivalent",
        "Some college",
        "Bachelor's degree",
        "Graduate or professional degree"
```

(continues on next page)

(continued from previous page)

```
    ]
}
```

Rating multiple items on the same scale (`radiogrid`)

Renders a table where each row is an item and each column is a point on the scale. The `labels` list defines the column headers; the `questions` list defines the rows.

```
{
    "questiontype": "radiogrid",
    "instructions": "How much do you agree with each statement?",
    "labels": [
        "Strongly disagree",
        "Disagree",
        "Neutral",
        "Agree",
        "Strongly agree"
    ],
    "questions": [
        {"id": "enjoy_research", "text": "I enjoy participating in research studies."}
↪,
        {"id": "feel_informed", "text": "I feel well informed about this study."},
        {"id": "trust_researcher", "text": "I trust the research team."}
    ],
    "shuffle": true,
    "required": true
}
```

How much do you agree with each statement?

	Strongly disagree	Disagree	Neutral	Agree	Strongly agree
I enjoy participating in research studies.	○	○	○	○	○
I feel well informed about this study.	○	○	○	○	○
I trust the research team.	○	○	○	○	○

`shuffle: true` randomises the row order each time the page loads, controlling for item-order effects. `required: true` on a `radiogrid` requires every row to have a response before the participant can continue.

The full list of question types — checklists, dropdowns, sliders, multi-line text, and more — with every supported property, is in *Built-in Question Types*.

2.1.3 Adding a questionnaire to PAGE_LIST

Once the JSON file exists, add it to the `PAGE_LIST` in your `config.toml`. The path format is `questionnaire/filename` (without the `.json` extension):

```
PAGE_LIST = [
    {name="Consent", path="consent"},
    {name="Demographics", path="questionnaire/demographics"},
```

(continues on next page)

(continued from previous page)

```
    {name="End", path="end"}
]
```

See *Setting Up Your Page Flow* for PAGE_LIST syntax and ordering options.

2.1.4 Required vs optional questions

Questions are optional by default. Set `"required": true` on any question to force a response before the participant can continue:

```
{
    "id": "consent_check",
    "questiontype": "radiolist",
    "instructions": "I have read and understood the information above.",
    "labels": ["I agree", "I do not agree"],
    "required": true
}
```

2.1.5 Hiding questions based on other answers

A question can include a `show_if` property containing an expression. The question is shown only when that expression evaluates to true; otherwise it is hidden.

The following example shows a follow-up only when the participant reports being under 18:

```
{
    "questions": [
        {
            "id": "age",
            "questiontype": "num_field",
            "instructions": "What is your age?",
            "required": true
        },
        {
            "id": "parental_consent",
            "questiontype": "radiolist",
            "instructions": "Has a parent or guardian reviewed and consented to your␣
↪participation?",
            "labels": ["Yes", "No"],
            "show_if": "age < 18"
        }
    ]
}
```

The expression `age < 18` references the `id` of another question in the same questionnaire. The hidden question is also skipped by the `required` check — a question that is not visible cannot block submission.

`show_if` can also reference answers from earlier questionnaires and use boolean operators (`and`, `or`, `not`). Full expression syntax is in *Expressions: Calculations and Conditional Display*.

For hiding entire *pages* based on a participant's answers or condition, see *Conditions and Branching*.

2.1.6 Previewing from the admin panel

Before exposing a questionnaire to participants, preview it from the admin panel:

1. Start your project: `BOFS run config.toml -d`
2. Visit `http://localhost:5000/admin` and log in.
3. Click **Preview Questionnaire** and select the file to view.

The preview renders the questionnaire as participants would see it, reports any JSON syntax errors, and offers to add new database columns if the questionnaire's question IDs have changed since the last run.

> **Note**
>
> **Modifying questionnaires with existing data**
>
> The database schema for a questionnaire is derived from its question IDs. Changing the questionnaire after responses have been collected — adding an item once pilot data exists, or revising an instrument mid-study — requires care.
>
> *During development*, the simplest approach is to delete the `.db` file and restart BOFS — the schema is recreated from scratch on the next run.
>
> *With live participant data*, you have three options:
>
> 1. Use the admin panel preview. BOFS will offer to add columns for any new question IDs without touching existing data.
> 2. Drop the questionnaire's database table. Responses for that questionnaire are lost; everything else stays intact.
> 3. Alter the schema manually with SQL. This preserves all data but requires SQL knowledge.
>
> Renaming or removing an existing question ID breaks the link to any responses already collected. Back up the database before making changes.

> **Warning**
>
> Restart BOFS after editing a questionnaire JSON file — the files are loaded at startup, not on each request.

2.1.7 Further reading

- *Built-in Question Types* — all question types and their properties.
- *Questionnaire Properties* — `{{ }}` value substitution, `participant_calculations`, questionnaire naming, and custom question types.
- *Expressions: Calculations and Conditional Display* — full expression syntax for `show_if` and `{{ }}`.

2.2 Setting Up Your Page Flow

A BOFS project is configured by one TOML file — typically `config.toml` — that defines settings, the page sequence (`PAGE_LIST`), and condition assignment. This page covers the configuration concepts you'll use most often. The full setting-by-setting reference lives at *Configuration Reference*.

2.2.1 TOML basics

TOML is a plain text format with key-value pairs, # for comments, and square brackets for lists:

```
TITLE = "My Research Study"
PORT = 5000

# Comments start with #.

PAGE_LIST = [
    {name="Consent", path="consent"},
    {name="End", path="end"}
]
```

2.2.2 Required settings

Every BOFS project needs at least these:

Setting	What it does
`TITLE`	Study name shown in the browser tab and admin panel.
`PORT`	The port the project runs on (e.g., `5000` for `http://localhost:5000`).
`SQLALCHEMY_DATABASE_URI`	A URL-style address telling BOFS where to store participant data. `sqlite:///study.db` means "a SQLite file named `study.db` in the project folder" (see *Choosing a database* below).
`ADMIN_PASSWORD`	Password for `/admin`.
`PAGE_LIST`	The ordered sequence of pages participants see.

For the rest of the available settings — application options, admin options, security, completion codes, conditions, deployment — see *Configuration Reference*.

2.2.3 PAGE_LIST: defining the page sequence

`PAGE_LIST` is the ordered list of pages a participant moves through. Each entry has a `name` (the human-readable label, shown in admin progress and breadcrumbs) and a `path` (the URL or page-type pattern):

```
PAGE_LIST = [
    {name="Display Name", path="route/path"},
    {name="Another Page", path="different/route"}
]
```

The first page should be one of these participant-creation routes:

- `consent` — display `consent.html`, create the participant on agreement, assign a condition.
- `consent_nc` — display `consent.html`, create the participant on agreement, no condition assignment.
- `create_participant` — no consent screen displayed; create the participant immediately and assign a condition.
- `create_participant_nc` — no consent screen, no condition.

The four variants and when to pick which are covered in *Consent Forms*.

The last page must be `end`. This shows the completion message and (if configured) generates a completion code participants can paste back into a recruitment platform. A study can have more than one end page when different participants should finish differently — see *Multiple end pages* below.

Page types between first and last:

Path format	What it shows
`external_id`	A page asking the participant to enter their external ID (MTurk Worker ID, Prolific PID, etc.). Configurable via `EXTERNAL_ID_LABEL` or `EXTERNAL_ID_PROMPT`.
`questionnaire/<name>`	The questionnaire from `questionnaires/<name>.json`.
`questionnaire/<name>/<tag>`	The same questionnaire, recorded with a tag (used for repeated measures — see *Including the same questionnaire multiple times* below).
`instructions/<name>`	The HTML at `templates/instructions/<name>.html`, wrapped in BOFS chrome with an automatic Continue button.
`simple/<name>`	The HTML at `templates/simple/<name>.html`, wrapped in BOFS chrome but with no Continue button — you control navigation. See *Adding Your Own Pages*.
`custom/<name>`	The HTML at `templates/custom/<name>.html`, served without BOFS chrome (no header, breadcrumbs, or styling). For embedded JS tasks.
`end` / `end/<reason>`	A study-completion page. `end` is the default exit; `end/<reason>` is an alternate exit tagged with `<reason>` (e.g. `end/screened_out`). See *Multiple end pages* below.
`assign_condition`	Triggers condition assignment if the participant doesn't have one yet. Useful when consent was collected via `consent_nc` or `create_participant_nc`.
`<blueprint_endpoint>`	A custom Python route from one of your blueprints — only relevant if you're writing Python. See *Blueprints and Routes*.

A complete example:

```
PAGE_LIST = [
    {name="Consent",             path="consent"},
    {name="External ID",         path="external_id"},
    {name="Demographics",        path="questionnaire/demographics"},
    {name="Task Instructions",   path="instructions/task_intro"},
    {name="Practice Trials",     path="custom/practice_task"},
    {name="Main Task",           path="custom/main_task"},
    {name="Post-Task Survey",    path="questionnaire/post_task"},
    {name="Debrief",             path="simple/debrief"},
    {name="End",                 path="end"}
]
```

For conditional routing (different page sequences per condition) and per-page `show_if` expressions, see *Conditions and Branching*.

2.2.4 Multiple end pages

Not every participant should finish the same way. Someone screened out at the demographics stage, someone who fails an attention check, and someone who completes the full protocol may each need a different closing message — and, when recruiting from a platform, a different completion URL so your records distinguish a full completion from an early exit.

Add more than one end entry to `PAGE_LIST`. The default exit keeps `path="end"`; each alternate exit uses `path="end/<reason>"`, where `<reason>` is a short label you choose. Gate the alternate exits with `show_if` so a participant reaches the one that applies to them, and put the default `end` last as the fallback:

```
PAGE_LIST = [
    {name="Consent",       path="consent"},
    {name="Demographics", path="questionnaire/demographics"},
    {name="Screened out", path="end/screened_out", show_if="demographics.age < 18"},
    {name="Study",         path="questionnaire/main"},
    {name="End",           path="end"}
]
```

Here an under-18 participant skips the study and lands on `end/screened_out`; everyone else continues to the main questionnaire and finishes at `end`. Gating works exactly as it does for any other page (see *Conditions and Branching*).

When a participant reaches an end page, BOFS records the reason on their `Participant` row as `end_reason` — `"complete"` for the plain `end` page, or the `<reason>` label otherwise. This column appears in the admin progress page and the data export, so you can count or filter participants by how they exited.

What each end page shows

By default every end page shows the standard completion message. To customise the text for a specific reason, add `templates/end/<reason>.html` with just the body content — BOFS wraps it in the usual page chrome, the same way `simple/` pages work. A plain `templates/end.html` override changes the default exit.

To send participants to an external URL instead of showing a page — a platform completion URL, for instance — add an `outgoing_url` to the end entry:

```
PAGE_LIST = [
    {name="Consent", path="consent"},
    {name="Screened out", path="end/screened_out", show_if="demographics.age < 18",␣
↪outgoing_url="https://app.prolific.co/submissions/complete?cc=SCREENOUT1"},
    {name="Study", path="questionnaire/main"},
    {name="Complete", path="end", outgoing_url=
↪"https://app.prolific.co/submissions/complete?cc=C1ABC123"}
]
```

`outgoing_url` is valid only on entries whose path is `end` or `end/<reason>`; BOFS rejects it elsewhere at startup. The string is rendered through Jinja with the `participant` object in scope, so it can include participant-specific values (e.g. `...&pid={{ participant.externalID }}`). A per-entry `outgoing_url` takes precedence over the project-wide `OUTGOING_URL` setting (see *Recruiting via MTurk or Prolific*).

2.2.5 Including the same questionnaire multiple times

Pre-test/post-test designs and longitudinal studies often need the same questionnaire administered more than once. Add it to `PAGE_LIST` multiple times with different `tag` values:

```
PAGE_LIST = [
    {name="Consent",       path="consent"},
    {name="Mood (pre)",    path="questionnaire/mood/pre"},
    {name="Task",          path="custom/task"},
    {name="Mood (post)",   path="questionnaire/mood/post"},
    {name="End",           path="end"}
]
```

Each tagged submission is stored as a separate row and exported separately by the admin panel — no code needed. If you write your own templates or Python routes, a tagged response is read back with `participant.questionnaire("mood", "pre")`; see *Using Participant Data* for the full API. Longitudinal studies that reuse questionnaires across separate sessions are covered in *Longitudinal Experiments*.

2.2.6 Multiple config files

A BOFS project isn't tied to a single `config.toml`. Each `.toml` file is self-contained — its own settings, `PAGE_LIST`, `CONDITIONS`, database URI, port — and `BOFS run` takes the path to whichever one you want to load:

```
BOFS run config.toml
BOFS run pilot.toml -d
BOFS run study_a.toml
```

Two common reasons to keep more than one:

- **Dev vs. production.** `dev.toml` for local work and `config.toml` for the deployed instance. `PAGE_LIST`, `CONDITIONS`, and questionnaire references usually match across both; what differs is `PORT`, `SQLALCHEMY_DATABASE_URI`, `ADMIN_PASSWORD`, and `BEHIND_REVERSE_PROXY`.
- **Multiple experiments in one project.** Separate studies that share the same custom pages, blueprints, or questionnaires can each have their own `.toml` with a different `PAGE_LIST` and database. Switch between them by passing a different file to `BOFS run`.

2.2.7 Choosing a database

The `SQLALCHEMY_DATABASE_URI` setting tells BOFS which database to use:

SQLite is the recommended default: `sqlite:///study.db`. It's a single-file database with no setup, and is good for development, piloting, and small or medium studies (i.e., dozens of concurrent users, not hundreds; the total participant count is not a factor).

If you need to handle a larger volume of participants, consider using PostgreSQL or MySQL. These can be hosted on a separate server to spread server load. BOFS interacts with the database through SQLAlchemy (a Python database library), so any database SQLAlchemy supports is also supported by BOFS.

> **Warning**
>
> Schema changes — adding fields to a questionnaire, adding columns to a custom table, adding new questionnaires — are applied at startup. With a fresh database (or one with no existing data), BOFS creates or alters the tables to match your JSON definitions. With existing participant data, schema changes can fail or quietly leave you with an inconsistent database. During development, the safest reset is to delete the SQLite `.db` file. With live data, see *Questionnaire Properties* for the modification guide.

2.2.8 See also

- The minimal example — the smallest workable `PAGE_LIST`.

2.3 Consent Forms

BOFS provides four first-page route variants (with and without consent display, with and without condition assignment), a wrapper template you can override, and support for multi-stage consent. The consent flow records consent by creating the participant: a `Participant` row exists only for someone who agreed, and its `timeStarted` marks when. The radio choice itself is not stored separately — it gates the form, and agreement is what creates the row.

2.3.1 The default consent flow

When `PAGE_LIST` starts with `{name="Consent", path="consent"}`, BOFS:

1. Renders `consent.html` from your project root, wrapped in a form with two radio options ("I give my consent" / "I do not give my consent") and a Continue button.
2. On submission, validates that the participant chose to consent.
3. If they consented, creates a `Participant` row, assigns a condition (if any are configured), and routes them to the next page in `PAGE_LIST`.
4. If they declined, the form fails validation with a message and the participant cannot continue. No row is created and the consent value itself is not stored anywhere.

A minimal `consent.html` looks like:

```
<h1>My Study</h1>
<h2>Consent to Participate</h2>
<p>You are invited to participate in a study about ...</p>
<p><strong>Principal Investigator:</strong> Dr. Example, Department of Examples</p>
<p><strong>IRB Approval:</strong> 2026-001</p>
```

The radios and Continue button are added by BOFS — your file holds only the consent text.

2.3.2 Choosing a first-page route

Four routes can sit at the top of `PAGE_LIST`. They differ on two axes: whether `consent.html` is displayed, and whether a condition is assigned. The `_nc` suffix stands for "no condition."

Route	Shows consent.html?	Assigns condition?	Use when
`consent`	Yes	Yes	The default. Online recruitment with conditions.
`consent_nc`	Yes	No	You collect consent in BOFS but conditions are assigned later (or not at all).
`create_participant`	No	Yes	Consent collected externally (e.g., paper consent in a lab session). Participant arrives at the URL after already consenting.
`create_participant_nc`	No	No	Consent collected externally and no conditions, or conditions assigned later via `assign_condition`.

For all four, the Participant row is created on first arrival (or first agreement, for the consent-displaying variants).

Picking the variant is a one-line change in `PAGE_LIST`:

```
PAGE_LIST = [
    {name="Consent", path="consent_nc"},      # show consent, no condition
    {name="Survey",  path="questionnaire/survey"},
    {name="End",     path="end"}
]
```

2.3.3 What happens on decline

Declining the consent radio fails the form's required-field validation. The participant sees the form re-rendered with an error message ("You must provide your consent to continue"). They cannot advance, and closing the tab is their exit. No participant row is created for someone who declines — the row exists only once consent is given, which is what makes its presence a record of consent.

If your IRB wants the consented text captured per participant rather than inferred from the row, repeat the key consent statements as a one-question questionnaire right after the consent page (see *Multi-stage consent*); the questionnaire response is stored and exported like any other.

2.3.4 Writing consent.html

The file contains your consent text — BOFS adds the radio buttons and Continue button around it. A minimal example:

```
<h1>Study Title</h1>
<p><strong>Principal Investigator:</strong> Name (email)</p>
<p><strong>IRB Approval:</strong> 2026-001</p>

<h3>Purpose</h3>
<p>...</p>

<h3>Procedures</h3>
<p>...</p>

<h3>Contact for questions</h3>
<p>...</p>
```

To link a downloadable PDF copy, place `consent.pdf` in your project's `static/` directory and link it at `/static/consent.pdf`. Static files are served at `/static/<filename>`.

2.3.5 Multi-stage consent

Studies that record additional consents (media release, debrief acknowledgement, second-language consent) usually add a follow-up page after the main consent. The follow-up is a regular page in `PAGE_LIST` — typically a `simple/` page or a one-question questionnaire:

```
PAGE_LIST = [
    {name="Consent",        path="consent"},
    {name="Media Release",  path="questionnaire/media_release"},
    {name="Demographics",   path="questionnaire/demographics"},
    {name="End",            path="end"}
]
```

The secondary response *is* recorded (questionnaire submissions persist, unlike the primary consent radios) and appears in the admin export like any other questionnaire — no code needed. From templates or Python it is accessible via `participant.questionnaire("media_release")`. This is the practical workaround if you need an audit trail for a specific consent decision.

2.3.6 Customizing the consent wrapper

To change the agree/decline wording, the button text, or the layout of the form BOFS wraps around your `consent.html`, override the wrapper template by creating `templates/consent.html` in your project (note: `consent.html` *and* `templates/consent.html` are different — the first is your study's consent text, the second overrides the wrapper). See *Templates and Jinja2* for template lookup order and override patterns.

2.3.7 See also

- *Configuration Reference* for consent-adjacent settings (`GENERATE_COMPLETION_CODE`, `ADMIN_PASSWORD`, `EXTERNAL_ID_*`).
- The minimal example for the default consent flow in context.

2.4 Adding Your Own Pages

BOFS supports three kinds of custom pages that don't require any Python:

- **Instruction pages** — static HTML with an automatic "Continue" button.
- **Simple pages** — HTML rendered inside the BOFS chrome (header, breadcrumbs, project styling), but no automatic Continue button.
- **Custom pages** — the template is the entire HTML document with no BOFS wrapping.

All three are HTML files placed in your project's `templates/` directory and referenced in `PAGE_LIST` by their path. Typical uses: instruction pages for task instructions or debriefing text, simple pages for gated progression (a timed reading period, a comprehension check), and custom pages for hosting JavaScript-based tasks.

2.4.1 Instruction pages

An instruction page displays static HTML content. BOFS wraps it in the project's standard layout and adds a "Continue" button at the bottom that advances to the next page in `PAGE_LIST`.

Place the file in `templates/instructions/` and reference it in `PAGE_LIST` as `instructions/<filename>` (without the `.html` extension).

`templates/instructions/welcome.html`:

```
<h2>Welcome to Our Study</h2>

<p>Thank you for participating. This experiment will take approximately 15␣
↪minutes.</p>

<p>During this study, you will:</p>
<ul>
    <li>Answer some demographic questions</li>
    <li>Complete a short task</li>
    <li>Provide feedback about your experience</li>
</ul>

<p><strong>Important:</strong> Please complete this study in one sitting.</p>
```

Then add the page to your configuration:

```
PAGE_LIST = [
    {name="Consent", path="consent"},
    {name="Welcome", path="instructions/welcome"},
    {name="Demographics", path="questionnaire/demographics"},
    {name="End", path="end"}
]
```

Instruction pages are Jinja2 templates — HTML files that can include placeholders BOFS fills in when the page is served — so you can embed session variables and conditional logic. See *Templates and Jinja2* for details.

2.4.2 Simple pages

A simple page is rendered inside the BOFS chrome — the participant sees the same header and breadcrumbs as the rest of the experiment — but there is no automatic Continue button. Your HTML controls when and how the participant advances.

Place the file in `templates/simple/` and reference it in `PAGE_LIST` as `simple/<filename>`.

To advance the participant, link or redirect to one of these routes (URL paths BOFS responds to):

- `/redirect_next_page` — go to the next page in `PAGE_LIST`.
- `/redirect_to_page/<path>` — go to a specific page (e.g., `/redirect_to_page/questionnaire/demographics`).

See *Built-in Routes* for the full redirect route reference.

This example shows an instruction page where the Continue button only appears after a 10-second delay:

`templates/simple/timed_instructions.html`:

```
<h2>Task Instructions</h2>

<p>Read these instructions carefully before proceeding.</p>

<div id="continue-area" style="display: none;">
    <p>You may now continue.</p>
    <button onclick="location.href='/redirect_next_page'">I'm Ready to Begin</button>
</div>

<script>
setTimeout(function () {
    document.getElementById('continue-area').style.display = 'block';
}, 10000);
</script>
```

`PAGE_LIST` entry:

```
PAGE_LIST = [
    {name="Instructions", path="simple/timed_instructions"},
    {name="Task", path="custom/my_task"},
    {name="End", path="end"}
]
```

2.4.3 Custom pages

A custom page renders your template as the complete HTML document. BOFS adds no header, breadcrumbs, or stylesheet. Use this when the task needs full control over the page — for example, a jsPsych, lab.js, PsychoJS, p5.js, or Unity experiment that must occupy the entire viewport or supply its own `<head>`.

Place the file in `templates/custom/` and reference it in `PAGE_LIST` as `custom/<filename>`.

`templates/custom/my_task.html`:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
```

(continues on next page)

(continued from previous page)

```
    <title>My Task</title>
</head>
<body>
    <main></main>
    <script src="/static/my_task.js"></script>
</body>
</html>
```

Custom pages are still Jinja2 templates and have access to the same template variables (`session`, `participant`, `config`, `debug`) as instruction and simple pages — for example, `{{ session['condition'] }}` anywhere in the HTML is replaced with the participant's condition number when the page is served. The same redirect routes apply: `/redirect_next_page`, `/redirect_to_page/<path>`, or a POST to the page's own route.

See *Quickstart: Integrating a Custom Task* for a worked example of a JavaScript task using a custom page.

2.4.4 Serving static files

BOFS serves everything in your project's `static/` directory at the URL path `/static/`. Reference files from any template using `/static/<path>`.

Common uses: stimulus images, videos, audio files, downloadable PDFs, and JavaScript task bundles.

Example project layout:

```
your_project/
├── static/
│   ├── images/
│   │   └── stimulus.jpg
│   └── audio/
│       └── instructions.mp3
└── templates/
    └── instructions/
        └── welcome.html
```

Displaying an image in an instruction page:

```
<h2>Study Materials</h2>

<img src="/static/images/stimulus.jpg" alt="Stimulus image" width="400">
```

2.4.5 Choosing between the three page types

Type	Use when
Instruction	You want static informational content with the standard Continue button and project styling.
Simple	You need project styling but want to control navigation yourself (e.g., a timer, a quiz gate, a custom form).
Custom	The task needs the full HTML document — its own `<head>`, full-viewport layout, or a JavaScript framework that conflicts with the BOFS chrome.

For pages that display dynamic content based on session data, questionnaire responses, or configuration variables, see *Templates and Jinja2*.

For pages that need server-side logic (Python routes, form handling, database writes), see *Blueprints and Routes*.

2.5 Conditions and Branching

Most experiments need at least some branching: A/B comparisons assign each participant to one of several conditions; pre-screening studies skip irrelevant follow-up questions; longitudinal studies carry condition assignments across sessions. BOFS handles all three with three primitives — `CONDITIONS`, `conditional_routing`, and `show_if`. Together they support between-subjects designs, where each participant is assigned to one condition and sees only the content meant for it.

2.5.1 Defining conditions

Set `CONDITIONS` in `config.toml` to enable condition assignment:

```
CONDITIONS = [
    {label="Control",        enabled=true},
    {label="High Reward",    enabled=true},
    {label="Low Reward",     enabled=true}
]
```

A few rules to keep in mind:

- **Participants are assigned to whichever enabled condition has the fewest participants** at the moment of assignment, with ties going to the condition listed first. This keeps group sizes close even with rolling recruitment. Note that this is deterministic minimization, not simple random assignment — if your protocol or method section describes the randomization procedure, describe it as minimization.
- **Condition numbers start at 1**, in the order they appear in the list. `Control` is condition `1`, `High Reward` is `2`, `Low Reward` is `3`.
- **Participants who haven't been assigned yet have condition ``0``.** This includes participants in projects with no `CONDITIONS` block at all.
- **Abandoned participants aren't counted when balancing.** "Abandoned" means inactive longer than `ABANDONED_MINUTES` (default 5 minutes). To include them, set `COUNTS_INCLUDE_ABANDONED = true`.
- **Setting `enabled = false` removes a condition from rotation** without renumbering. Useful when one cell hits its target N before the others. You can also temporarily disable or enable conditions from the admin panel's progress page.

2.5.2 Where condition assignment happens

Five routes can trigger condition assignment:

- `consent` and `create_participant` — assign on first arrival.
- `consent_nc` and `create_participant_nc` — do **not** assign (the `_nc` suffix means "no condition").
- `assign_condition` — a standalone PAGE_LIST entry. Triggers assignment if the participant doesn't already have a condition. Use this when consent was collected via one of the `_nc` variants and you want to assign later in the flow.

Once a participant has a condition, it doesn't change. The same participant returning to the project always sees the same branch.

2.5.3 Carrying conditions across sessions

For longitudinal studies, the second-session participant needs the same condition they were assigned on day 0. Two configuration settings handle this:

- `CONDITIONS_FROM_DB = true` looks up the returning participant by their external ID and reuses their prior condition. The lookup runs against the project's own participant database, so it works without exporting any data.
- `CONDITIONS_FROM_CSV = "path/to/file.csv"` pre-assigns conditions from a CSV file keyed by external ID. The file is two columns (external ID, condition number) and is read at startup.

The full longitudinal pattern, including how the participant arrives back at the second session, is covered in *Longitudinal Experiments*.

2.5.4 Conditional routing

The `conditional_routing` block in `PAGE_LIST` shows different page sequences to different conditions. Each branch has a `condition` number (and/or a `show_if` expression) plus a nested `page_list`:

```
PAGE_LIST = [
    {name="Consent",       path="consent"},
    {name="Demographics", path="questionnaire/demographics"},

    {conditional_routing=[
        {condition=1, page_list=[
            {name="Control Instructions",     path="instructions/control"},
            {name="Control Task",             path="custom/task_control"}
        ]},
        {condition=2, page_list=[
            {name="High Reward Instructions", path="instructions/high_reward"},
            {name="High Reward Task",         path="custom/task_high_reward"}
        ]},
        {condition=3, page_list=[
            {name="Low Reward Instructions",  path="instructions/low_reward"},
            {name="Low Reward Task",          path="custom/task_low_reward"}
        ]}
    ]},

    {name="Post-Task", path="questionnaire/post_task"},
    {name="End",       path="end"}
]
```

Each participant follows exactly one branch. The branches can have different lengths and different page types, but they should converge back to a shared post-task and end page (otherwise different conditions complete at different points).

A `conditional_routing` branch can also key off prior questionnaire answers using `show_if` instead of (or in addition to) `condition`:

```
{conditional_routing=[
    {show_if="demographics.age < 18", page_list=[
        {name="Minor track", path="custom/task_minor"}
    ]},
    {show_if="demographics.age >= 18", page_list=[
        {name="Adult track", path="custom/task_adult"}
    ]}
]}
```

Both fields are optional. A branch matches when its `condition` matches (when set) and its `show_if` is true (when set); the first matching branch wins.

2.5.5 Page-level `show_if`

Any single PAGE_LIST entry — outside or inside a `conditional_routing` block — can carry a `show_if` expression. When it evaluates false against the participant's stored answers, that page is skipped:

```
PAGE_LIST = [
    {name="Consent",         path="consent"},
    {name="Demographics",    path="questionnaire/demographics"},
    {name="Parent contact", path="questionnaire/parent_contact", show_if=
↪"demographics.age < 18"},
    {name="End",             path="end"}
]
```

The expression has access to all of the participant's stored questionnaire fields, condition number, and custom-table export aggregates. The full syntax — operators, functions, qualified references for tagged questionnaires (`mood.pre.score`), table references (`tables.scores.high_score`) — is in *Expressions: Calculations and Conditional Display*.

2.5.6 Accessing the condition in templates and routes

This section applies only if you are writing your own templates or Python routes — `conditional_routing` and `show_if` cover condition-dependent flow without any code.

Inside a Jinja template:

```
{% if session['condition'] == 1 %}
  <p>Control instructions go here.</p>
{% elif session['condition'] == 2 %}
  <p>High-reward instructions go here.</p>
{% endif %}
```

Inside a Python blueprint route, read it from the session:

```
from flask import session
condition = session['condition']  # int starting at 1, or 0 if unassigned
```

For more on the participant object and template variables, see *Using Participant Data*.

2.5.7 A note on breadcrumbs

If `USE_BREADCRUMBS` is enabled (the default) and the project uses `conditional_routing` or page-level `show_if`, BOFS prints a startup warning. Breadcrumbs show every page in `PAGE_LIST` to every participant, which can reveal the structure of conditions or hidden follow-up pages — a participant who sees entries for other branches may infer the manipulation (demand characteristics). Either disable breadcrumbs (`USE_BREADCRUMBS = false`) or accept that participants will see entries for pages they won't actually visit.

2.5.8 See also

- The A/B experiment example — two conditions with conditional routing in a complete project.
- The branching example — question-level and page-level `show_if` in one project.
- *Longitudinal Experiments* for `CONDITIONS_FROM_DB` and `CONDITIONS_FROM_CSV` walkthroughs.
- *Expressions: Calculations and Conditional Display* for full expression syntax.

2.6 Longitudinal Experiments

A longitudinal study has the same participant returning across multiple sessions — hours, days, or weeks apart — and answering related questions or doing related tasks each time. Examples: a baseline survey followed by a one-week follow-up, a four-day diary study, a learning task with a 24-hour retention test.

BOFS supports two shapes for this. They share the same building blocks — a stable external ID and a database that persists between visits — but differ in whether each visit runs through the same `PAGE_LIST` or whether each session is its own deployment.

Pattern A — Same PAGE_LIST, repeated visits A single `config.toml` describing one visit's flow. When the participant returns with the same external ID, BOFS creates a new participant row but carries their condition forward, so they run through the same flow again as the same group.

Suits diary studies, daily check-ins, weekly mood surveys, or any design where every visit asks the same questions.

Pattern B — Separate per-session deployments A `day1.toml` and `day2.toml` (and possibly more), each with its own `PAGE_LIST` and its own database. Day 2 is a fresh session; BOFS uses the external ID to recover the participant's prior condition assignment from day 1's database (or a CSV).

Suits two-part studies where day 2's content differs from day 1's, separate recruitment links per session (e.g. a Prolific follow-up study), or longer gaps where you want each session's data isolated.

The longitudinal example linked at the bottom of this page is built as Pattern B. The rest of this page covers what each pattern needs and what they have in common.

2.6.1 How returning participants are recognized

Both patterns need a stable identifier the participant brings back with them. BOFS supports three sources:

- **A URL parameter.** When a participant arrives at `http://your-study.example/?external_id=abc123`, BOFS captures the value in the session as `externalID` (older projects may refer to it as `mTurkID` — the two names are aliases for the same value). Recruitment platforms like Prolific append the participant ID automatically — Prolific's `PROLIFIC_PID` parameter is also captured.
- **A manual entry page.** Adding `{name="Enter ID", path="external_id"}` to `PAGE_LIST` shows a form asking the participant to type their ID. Customizable via `EXTERNAL_ID_LABEL` and `EXTERNAL_ID_PROMPT`. With URL-parameter capture in place this page is optional — include it only as a fallback for participants who arrive without a platform-provided ID.
- **A blueprint route you write.** For custom recruitment flows, a Python view can call `set_external_id_in_session(value)` (from `BOFS.util`) to populate the external ID. Existing code that writes `session['mTurkID']` directly still works.

For details, see *Recruiting via MTurk or Prolific*.

2.6.2 Pattern A — Same PAGE_LIST, repeated visits

Two configuration settings drive this:

- `RETRIEVE_SESSIONS = true` — when a participant returns with the same external ID, BOFS looks up their past attempts and copies the prior condition forward into the new session.
- `ALLOW_RETAKES = true` — lets a participant who already finished start a fresh attempt. With `ALLOW_RETAKES = false`, a finished participant who returns is restored to their finished state and shown the end page again, so they can't run through the flow a second time.

Each return creates a new `Participant` row sharing the same `externalID`. All rows for that participant accumulate in the same database — questionnaire responses, custom-table rows, timestamps. The `Participant.timeStarted` column distinguishes attempts; admin-panel exports group by external ID.

Because each visit is its own participant row, the same questionnaire submitted on visit 1 and visit 2 doesn't collide — they're separate rows in the questionnaire's table, distinguishable by `participantID` and timestamp. No tagging required.

> **Note**
>
> **Repeated measures within a single visit.** If you instead need the same questionnaire administered twice in *one* run through `PAGE_LIST` (e.g. a pre-test and post-test on the same day), use the `tag` field on each `PAGE_LIST` entry — that's a separate mechanism from longitudinal returns. See *Including the same questionnaire multiple times* in the page-flow reference.

For the underlying session lifecycle and the IP-binding interaction, see *Sessions and Participant State*.

2.6.3 Pattern B — Separate per-session deployments

Each session runs as its own BOFS instance with its own `config.toml`, its own `PAGE_LIST`, and its own database. A typical day-2 configuration:

```
SQLALCHEMY_DATABASE_URI = 'sqlite:///day2.db'

RETRIEVE_SESSIONS = true
ALLOW_RETAKES = false

# Look up the returning participant in day 1's database and reuse their condition.
CONDITIONS_FROM_DB = 'sqlite:///day1.db'

CONDITIONS = [
    {label="Linear Menu",  enabled=true},
    {label="Marking Menu", enabled=true}
]

PAGE_LIST = [
    {name="Consent",        path="consent_nc"},
    {name="Participant ID", path="external_id"},
    {name="",               path="assign_condition"},
    # ... day-2 content ...
    {name="End",            path="end"}
]
```

Three things are unique to Pattern B:

- **Carrying conditions across deployments.** Either `CONDITIONS_FROM_DB` or `CONDITIONS_FROM_CSV` tells day 2 where to look up the condition assigned on day 1. `CONDITIONS` itself must list the same conditions in the same order as the source — the lookup is by integer position.
  - `CONDITIONS_FROM_DB = '<sqlalchemy-uri>'` — points at another BOFS database (typically day 1's). The returning participant's external ID is matched against the `Participant` table there and their condition is reused.
  - `CONDITIONS_FROM_CSV = "path/to/file.csv"` — pre-assigns conditions from a CSV file keyed by external ID (two columns: external ID, condition number). Useful when condition assignment is decided outside BOFS. If both `CONDITIONS_FROM_CSV` and `CONDITIONS_FROM_DB` are set, the CSV is consulted first and the DB is the fallback.
- **Consent without re-randomizing.** Plain `consent` runs condition assignment at submission time, which would

burn a random condition on day 2 before the external ID had been collected. Use `consent_nc` (no condition) on day 2 instead.

- **Explicit ``assign_condition`` after the external ID.** Place `{name='', path='assign_condition'}` *after* `external_id` in `PAGE_LIST` so the lookup has something to match on. A participant entering an unrecognized ID is shown an "ID Not Recognized" page and cannot proceed.

`RETRIEVE_SESSIONS` is still useful within each deployment — if a participant drops off partway through day 2 and returns later, BOFS resumes them in day-2's `PAGE_LIST`. `ALLOW_RETAKES` should usually stay `false` for the same reason as in Pattern A.

For the underlying lifecycle and the IP-binding interaction, see *Sessions and Participant State*.

2.6.4 Branching by prior responses

Page-level `show_if` predicates and `conditional_routing` blocks both operate on the *current* participant row. `has_questionnaire('survey')` checks the rows on the participant currently in session — it will not see questionnaire submissions from a prior visit (Pattern A, separate participant row) or from a prior deployment (Pattern B, separate database).

To branch on cross-visit data, query it from a custom blueprint route — for Pattern A, by external ID against the same database; for Pattern B, by external ID against day 1's database (the same one `CONDITIONS_FROM_DB` points at). Branching by the `condition` value works in both patterns, since the condition is carried forward.

The expression syntax is described in *Expressions: Calculations and Conditional Display*. The full branching pattern is in *Conditions and Branching*.

2.6.5 Storing custom task data across sessions

Custom tables defined in `tables/*.json` accumulate rows across sessions. Every row carries the participant's `participantID` and a `timeSubmitted` timestamp. See *Storing Custom Data*.

In Pattern A, each visit is a separate `participantID`, so rows naturally separate by visit — group by `participantID` then look up the shared `externalID` on the `Participant` table to combine across visits. In Pattern B, custom-table data lives in each deployment's own database, so analysis that spans days requires reading from both.

2.6.6 Bringing participants back

This part is out of scope for BOFS. Email reminders, scheduling, calendar invites, and SMS notifications need to come from outside the framework. The practical pattern is:

1. Export the participant ID list from the admin panel after day 1.
2. Send each participant a reminder (email, SMS, recruitment-platform message) with the study URL and their external ID.
3. They click the link, the URL parameter populates the external ID, and BOFS recognizes them.

Recruitment platforms (Prolific, MTurk) handle this for you — Prolific in particular has built-in support for two-part studies that auto-message participants.

2.6.7 Testing during development

Walking through a multi-session flow on your own machine takes a bit of setup:

- **Inspecting a participant's progress.** The admin panel's participant detail view shows the current page, the questionnaires submitted, and the timestamps. Use it to confirm a returning session resumed at the right spot.

- **Faking a return (Pattern A).** Complete the flow with `external_id=test1`, then reload with the same external ID — a new participant row is created, the prior condition is carried forward, and you can run through the flow again.
- **Running both deployments side by side (Pattern B).** Start each `.toml` in its own terminal: `BOFS run day1.toml -d` on one port, `BOFS run day2.toml -d` on another. Complete a participant in day 1 first so day 2's `CONDITIONS_FROM_DB` lookup has something to find.

2.6.8 See also

- The longitudinal example — a two-day HCI menu-learning study built as Pattern B, using `CONDITIONS_FROM_DB` to carry condition from day 1 to day 2. The example also includes a `conditions.csv` to demonstrate the `CONDITIONS_FROM_CSV` alternative.

2.7 Storing Custom Data

Questionnaire responses are stored automatically — BOFS creates the database columns from your question IDs and writes each submission for you. Task data — the trial-level measurements an embedded task produces, such as reaction times, accuracy, scores, or event sequences — doesn't have that automatic path. Custom tables give it one.

A custom table is a JSON file that describes the columns you want. BOFS creates the corresponding database table at startup and accepts rows at the address `/table/<name>`. The JavaScript running your task sends each row there as a standard web request (a POST); the examples below use JavaScript's built-in `fetch` function.

2.7.1 Defining a Table

Place a `<name>.json` file in the `tables/` directory at your project root. The file name (without the `.json` extension) becomes the table name used in routes and Python access.

```
my_study/
├── config.toml
├── tables/
│   └── trials.json
└── questionnaires/
```

Tables can also live inside a blueprint at `<blueprint_name>/tables/`. See *Blueprints and Routes* for how blueprint-scoped tables are discovered.

Column Types

Each entry in the `"columns"` object is one column. Valid types are `integer`, `float`, `boolean`, `string`, `datetime`, and `json`. If `"type"` is omitted, the column defaults to `string`.

A minimal table for a reaction-time task:

```
{
  "columns": {
    "score": {"type": "integer", "default": 0},
    "reaction_time": {"type": "float", "default": 0}
  }
}
```

Save this as `tables/trials.json` and BOFS creates the table the next time the project starts.

Two columns are added automatically to every custom table and should not appear in your file or in POST payloads:

- `participantID` — links the row to the participant who submitted it.
- `timeSubmitted` — the server's UTC time at the moment of insert. Timestamps come from the server, not the participant's machine, so they are comparable across participants regardless of time zone or clock settings.

The full list of column types, default-value rules, and naming constraints is in *Custom Tables*.

2.7.2 Writing Data from JavaScript

POST a JSON object to `/table/<name>` from your task code. The server attaches `participantID` and `timeSubmitted` automatically.

```
fetch('/table/trials', {
    method: 'POST',
    headers: {'Content-Type': 'application/json'},
    body: JSON.stringify({score: 42, reaction_time: 312.5})
});
```

This code goes wherever your task logic runs — typically inside the `<script>` block of a simple or custom page (see *Adding Your Own Pages*). Call it whenever you have a row to record: once per trial for trial-level data, or once at the end of the task for summary data.

On success the request returns status `204 No Content` — the row was stored and there is nothing to send back. A malformed payload returns `400 Bad Request`.

The `"json"` column type accepts a JavaScript object or array directly in the payload — useful for per-trial event logs, mouse trajectories, or keystroke timings where a flat column per measurement would be impractical:

```
fetch('/table/trials', {
    method: 'POST',
    headers: {'Content-Type': 'application/json'},
    body: JSON.stringify({
        score: 42,
        events: [{t: 0, x: 512, y: 300}, {t: 16, x: 510, y: 298}]
    })
});
```

When you later read the row back, `events` is already a JavaScript array — no `JSON.parse` needed.

For fast-paced tasks, you can also collect rows locally and send them in one request at the end (batch insert); see *Custom Tables* for batch inserts and form-encoded POSTs.

2.7.3 Reading Data Back

Reading is useful when a page needs the participant's earlier rows — to display a score, show performance feedback, or check whether a practice block met a threshold before continuing. A GET request to `/table/<name>` returns a JSON array of all rows belonging to the current participant:

```
fetch('/table/trials')
    .then(r => r.json())
    .then(rows => {
        // rows is an array of objects, e.g. [{score: 42, reaction_time: 312.5}, ...]
        console.log(rows);
    });
```

Query-string parameters are applied as exact-match filters:

```
fetch('/table/trials?score=42')
    .then(r => r.json())
    .then(rows => console.log(rows));
```

Note

Query-string filters use string-cast equality. Range queries and more complex filtering must be done in Python or post-processed on the client.

2.7.4 Calculated Export Fields

The admin panel's per-table CSV export gives you raw rows. Calculated export fields let you define per-participant summary values — a trial count, a mean reaction time — computed from those rows. Once defined, they appear as columns on the admin **Export** page (see *Monitoring and Exporting Data*) and are accessible from page templates.

Add an `"exports"` block to the table's JSON file. Each entry in the array defines one set of fields, each computed by a summary (aggregate) function over the participant's rows:

```
{
  "columns": {
    "score": {"type": "integer"},
    "reaction_time": {"type": "float"}
  },
  "exports": [
    {
      "fields": {
        "trial_count": "count(score)",
        "avg_rt": "avg(reaction_time)"
      }
    }
  ]
}
```

The supported aggregate functions are `MIN`, `MAX`, `SUM`, `COUNT`, and `AVG` — smallest, largest, total, number of rows, and mean. Optional `filter`, `group_by`, `order_by`, and `having` keys are available for more complex exports.

Calculated fields are reachable in page templates through `participant.table('<name>')`:

```
{% set trials = participant.table('trials') %}
<p>Trials completed: {{ trials.trial_count }}</p>
<p>Average RT: {{ trials.avg_rt }}</p>
```

Scalar export fields are also reachable in `show_if` conditions via the `tables.<name>.<column>` reference form — for example, to skip a page unless the participant's average reaction time crossed a threshold. See *Expressions: Calculations and Conditional Display* for expression syntax.

The complete reference — all export keys, `group_by` behaviour, the `@page_tables` decorator for the participant detail view, and the Python `db` API — is in *Custom Tables*.

2.7.5 Further Reading

- *Custom Tables* — all column types, export keys, naming rules, batch inserts, Python API, and the `@page_tables` decorator.
- *Expressions: Calculations and Conditional Display* — `tables.<name>.<column>` reference form for `show_if` and `participant.evaluate()`.
- *The Database Layer* — SQLAlchemy session and model details.
- *Monitoring and Exporting Data* — admin export panel.

2.8 Monitoring and Exporting Data

Every BOFS project ships with an admin panel for monitoring participants, exporting data, previewing questionnaires, and inspecting the underlying database.

2.8.1 Accessing the Panel

The admin panel lives at `/admin`. For a project running locally on port 5000, that's http://localhost:5000/admin.

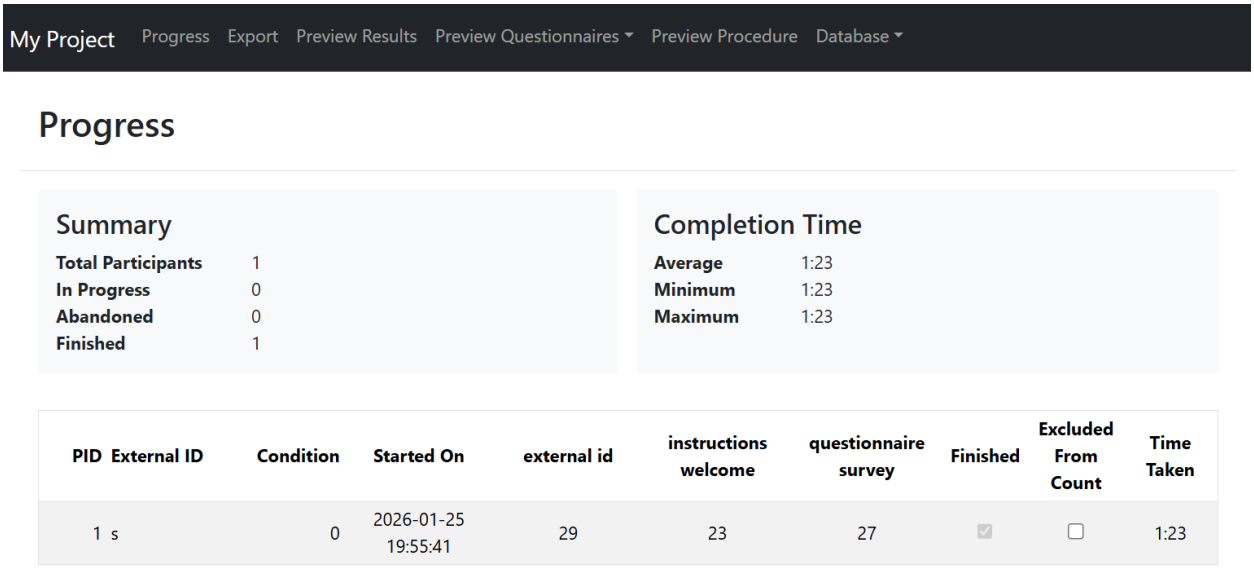
Set the password in your `.toml` file:

```
ADMIN_PASSWORD = 'your_secure_password_here'
```

This password protects all participant data and any destructive controls (such as clearing the database), so use a strong, unique value rather than a placeholder.

2.8.2 Progress Monitoring

`/admin/progress` shows each active participant's position in the experiment. The page updates every 5 seconds automatically and includes per-participant exclusion checkboxes.



The same page reports summary statistics, broken down by experimental condition:

Table 1: Progress Statistics

Metric	Description
Total Participants	Count by experimental condition
Abandoned Participants	Those inactive longer than `ABANDONED_MINUTES`; not counted when balancing conditions by default (see *Conditions and Branching*)
In-Progress Participants	Currently active participants
Finished Participants	Those who have completed all pages
Average Duration	Mean completion time by condition
Min/Max Duration	Fastest and slowest completion times

When a study uses more than one end page (see *Multiple end pages*), the progress page also breaks participants down by `end_reason` — how each one exited — so you can see, for example, how many were screened out versus completed.

Participant Detail View

Clicking a participant ID on the progress dashboard opens that participant's detail view (`/admin/participant/<id>`). This page reconstructs the participant's run through the experiment as a timeline, with one card per page in their `PAGE_LIST` (condition-aware, so participants in different conditions see only the pages that apply to them).

Each timeline card shows:

- **Status**: Completed, In Progress, or Not Reached.
- **Timing**: when the page was started, when it was submitted, and the duration on the page.
- **Submitted data**, broken down by page type:
  - *Questionnaire pages* — every field's prompt, field ID, and the participant's response, plus any calculated fields defined on the questionnaire.
  - *Custom pages* writing to a JSONTable — the calculated export fields from the table's `exports` block, scoped to this participant. Opting a page in applies only to blueprint-defined Python pages: decorate the view function with `@page_tables('<table_name>')`; see *Custom Tables* for details.

The page header also shows the participant's external ID (e.g. Prolific PID), assigned condition, total duration, last-active timestamp, and an Excluded badge when applicable.

2.8.3 Data Export

`/admin/export` downloads questionnaire responses as CSV. Options include excluding unfinished or excluded participants, previewing the table in HTML before downloading, and automatic timestamping of the filename. The export includes each participant's `source` (recruitment-channel tag) and `end_reason` (how they exited), so you can filter or group by either in your analysis.

For questionnaire interaction data (only collected when `LOG_QUESTIONNAIRE_INTERACTIONS` is enabled), use `/admin/export_item_timing`. This exports a flat event log — one row per event — with participantID, externalID, questionnaire, tag, questionID, eventType, timestamp, and value. See *Data Quality* for what the events capture and how to use them.

Any database table — built-in (`Participant`, `Progress`, `Response`) or defined by a custom blueprint — can be exported individually via `/admin/table_csv/<table_name>`.

2.8.4 Results Analysis

`/admin/results` groups every response field by condition and splits them into two tables, both cached for two minutes:

- **Summary Statistics** — continuous measurements (sliders, number fields, numeric calculations). Shows N, min/max, mean, median, standard deviation, standard error, and variance.
- **Response Counts** — discrete responses (true/false, radio-list and drop-down selections, checklist flags, radio-grid cells, and free text). Shows the number of distinct responses and the total. Free-text "other" answers each appear as their own response.

Selecting a field opens `/admin/results/<field_name>`, an interactive Plotly.js chart by condition with zoom, pan, and hover tooltips: a box plot (with outliers) for a continuous field, or a grouped bar histogram for a count field, alongside a per-condition table.

2.8.5 Questionnaire Management

`/admin/preview_questionnaire/<name>`
: Renders a questionnaire as participants would see it. Surfaces JSON parsing errors, lets you switch conditions for previewing conditional questions, and marks questionnaires that already have live participant data with an asterisk.

`/admin/questionnaire_html/<name>`
: Plain HTML rendering with no admin chrome — useful for embedding, printing, or sharing.

`/admin/preview_procedure`
: Generates a flowchart of your `PAGE_LIST`, including the per-condition routes from any `conditional_routing` blocks — useful for checking the flow against your protocol, or for a figure in an ethics application.

2.8.6 Database Management

`/admin/table_view/<table_name>` shows the live contents of any database table with automatic refresh and column-type detection.

For SQLite databases only:

- `/admin/database_download` downloads the full database file — useful for backups or offline analysis.
- `/admin/database_delete` clears the database. The action is password-protected; only the table structure survives. It attempts to copy the SQLite file to a timestamped sibling first, but treat that copy as a courtesy, not a backup — it is not verified, does not include WAL/SHM sidecar files, and may not be reachable from where you're running BOFS (e.g. inside a container).

> **Warning**
>
> Database deletion is irreversible. Take your own backup (`/admin/database_download`, or copy the `.db` file directly) before using it.

2.8.7 Configuration

Table 2: Admin Configuration

Variable	Type	Description
ADMIN_PASSWORD	string	**Required.** Password for admin panel access.
USE_ADMIN	boolean	Enable or disable the admin panel entirely (default: `true`).
LOG_QUESTIONNAIRE_INTE	boolean	Log questionnaire interaction events for export (default: `false`).
ADDITIONAL_ADMIN_PAGES	list	Custom admin pages from blueprints (see below).

Adding Custom Admin Pages

`ADDITIONAL_ADMIN_PAGES` adds entries to the admin navigation. Each entry is either a Flask route from one of your blueprints or an external URL:

```
ADDITIONAL_ADMIN_PAGES = [
    {name = "Custom Analysis", route = "my_blueprint.custom_analysis"},
    {name = "External Tool", url = "https://example.com/tool"}
]
```

To protect a custom Flask route with the same authentication as the built-in admin pages, decorate it with `@verify_admin`:

```
from BOFS.admin.util import verify_admin
from flask import Blueprint

my_blueprint = Blueprint('my_blueprint', __name__)

@my_blueprint.route('/admin/my_custom_page')
@verify_admin
def my_custom_admin_page():
    return render_template('my_admin_page.html')
```

2.8.8 Security

The admin panel exposes every participant's data and includes destructive controls. For production deployments:

- Pick a strong, unique password for `ADMIN_PASSWORD`.
- Serve the site over HTTPS.
- Back up the database before using `/admin/database_delete`.
- Be deliberate when exporting data — the CSV files contain anything participants entered.

2.8.9 Setup Diagnostics

At startup, BOFS validates your configuration, questionnaires, tables, database, and blueprints, and records what it finds as a set of diagnostics. Each diagnostic has one of three severity levels:

- **Error** — a problem that stops the study from running. The experiment is not accessible to anyone until every error is resolved.
- **Warning** — something that is likely a mistake but doesn't stop the study, such as a questionnaire referencing an image file that isn't on disk, or a questionnaire whose `questions` array is empty.
- **Notice** (info) — a change or condition BOFS wants you to know about but that needs no action, such as a `SECRET_KEY` migrated from your config into the project database, a `SQLALCHEMY_BINDS` entry that no questionnaire or table references, or a database column that exists but is no longer defined in its questionnaire JSON (its old data is preserved; new submissions store NULL).

Diagnostics are grouped into five sections — Config, Questionnaires, Tables, Database, and Blueprints — and appear in three places.

Console output at startup

Every diagnostic is written to the application log as BOFS starts, at a level matching its severity. After validation, BOFS also prints a summary: when there are errors it prints a banner noting the experiment will not be accessible until they're fixed; when there are only warnings it prints a one-line count and directs you to open the app for details. On a deployed server these lines appear in the service log (see *Deploying to a Server*).

Admin navbar pills and modal

When a project has warnings or notices, the admin navbar shows a pill for each — a "warnings" pill and a "notices" pill. Clicking either opens the Setup Diagnostics modal, which lists every diagnostic grouped by section and, within a section, by the file or setting that produced it. A questionnaire's diagnostics group under its filename; config-level diagnostics group under labels such as `PAGE_LIST`, `Database bindings`, and `Security`. Errors don't get a pill, because an error blocks access to the admin panel entirely (see below).

Participant-facing setup pages

When there are **errors**, BOFS replaces every page — including the admin panel — with a setup-error page that lists the errors and their suggested fixes. This is what anyone visiting the study sees until the errors are resolved, so participants can't reach a broken experiment.

When there are only **warnings**, BOFS shows a one-time interstitial on the study's entry pages listing the warnings, with a link to continue into the study. This interstitial only appears to you as the researcher — it renders when you're running in debug mode (`-d`) or viewing from the same machine, and only before any participant session exists. Real participants recruited to the live study never see it. Following the "Continue to study" link dismisses the interstitial for the rest of that browser session.

Resolving diagnostics

Diagnostics are evaluated once, when BOFS starts. Fix the underlying config, questionnaire, table, or database issue, then restart the project to re-evaluate. Dismissing the warning interstitial hides it for the session but doesn't clear the warning — it reappears after the next restart until the cause is addressed.

2.8.10 Troubleshooting

Login problems
: Confirm `ADMIN_PASSWORD` is set correctly. Try an incognito window to rule out a stale session, and check that cookies are enabled.

Export problems
: For large studies, check available disk space and that the application directory is writable. For interaction timing exports specifically, `LOG_QUESTIONNAIRE_INTERACTIONS` must be enabled.

Slow results page
: Results are cached for two minutes — wait for the cache to refresh before assuming new data isn't loading. For very large datasets, add database indexes on frequently queried columns.

SQLite-only features unavailable
: `/admin/database_download` and `/admin/database_delete` only work when the project uses SQLite. With PostgreSQL or another backend, use the database's own tools.

2.9 Data Quality

Every online study collects some responses you'd rather leave out: submissions from automated bots, the same person participating more than once, and answers given too carelessly or quickly to be usable. BOFS offers several ways to reduce and detect these; some run automatically, others you turn on when your study needs them.

2.9.1 Automatic bot screening

Two protections run on every study with no configuration.

User-agent check. BOFS checks each new participant's user agent against a list of known web crawlers (automated programs that visit pages, such as search-engine indexers). A visitor that matches is flagged and left out of condition-balancing counts.

Honeypot field. The consent page includes a decoy text field hidden from view. A person filling out the form never sees it, but some automated bots fill in every field they find. If the decoy input is given a value, BOFS treats the submission as a bot and does not create a participant.

These checks are not foolproof. A determined bot can spoof a normal user agent and skip hidden fields. However, they do exclude some automated traffic (such as web crawlers) without affecting real participants.

2.9.2 Logging how participants respond

BOFS can record *how* a participant fills out each questionnaire, not just their final answers. This is off by default. Turn it on in your config:

```
LOG_QUESTIONNAIRE_INTERACTIONS = true
```

> **Note**
>
> Earlier versions called this setting `LOG_GRID_CLICKS`. The old name still works, but `LOG_QUESTIONNAIRE_INTERACTIONS` is the current spelling.

With logging on, BOFS records a timestamped event each time a participant interacts with a questionnaire input. The event types are:

- `focus` and `blur` — entering and leaving an input.
- `change` — a changed answer.
- `paste` — text pasted into a field (rather than typed).
- `drop` — text dragged and dropped into a field (rather than typed).

- `paste_blocked` / `drop_blocked` — a paste or drop that was prevented because paste is disabled for that field (see **Disabling paste** under *Keeping participants engaged with materials* below). The attempt and its character count are still recorded, but no text was inserted.
- `visibility` — the browser tab being hidden or shown, which happens when the participant switches away to another tab or window.

Together these let you measure how much time participants spent on each item, whether they revised answers, whether they pasted responses in, and whether they left the page mid-questionnaire. These signals can be used to gauge attentiveness and effort.

To get the data, open the admin **Export** page and download the questionnaire interaction log (see *Monitoring and Exporting Data*). It is a flat table with one row per event: `participantID`, `externalID`, `questionnaire`, `tag`, `questionID`, `eventType`, `timestamp`, and `value`.

> **Note**
>
> The `value` column records the input's contents at the time of the event, so the log can contain participants' own responses. Treat it as participant data: store and share it under the same conditions as the rest of your dataset, and check your IRB's guidance before exporting.

2.9.3 Preventing repeat submissions

When you recruit with a participant identifier (a Prolific or MTurk ID, or a code you assign), BOFS can keep the same person from completing the study twice. Setting `ALLOW_RETAKES = false` blocks an identifier that has already finished from starting a fresh attempt, while `RETRIEVE_SESSIONS = true` lets someone who was interrupted resume where they left off. This catches repeat submissions from the same identifier; it does not catch one person returning under a new identifier. The full behaviour is described in *Recruiting via MTurk or Prolific* and *Sessions and Participant State*.

2.9.4 Keeping participants engaged with materials

Two questionnaire features help ensure participants actually engage with what you present:

- **Required questions** (`"required": true`) prevent a participant from advancing past a question they have left blank. See *Adding Survey Questions*.
- **Force-watch media** (`"force_watch": true` on a `video` or `audio` question) keeps the Continue button disabled until a video clip has been played. See *Built-in Question Types*.
- **Disabling paste** stops participants from pasting (or dragging) text into a field, so free-text answers have to be typed. Set `DISABLE_PASTE = true` in your config to apply this to every text input across the study, or set `"disable_paste": true` on an individual text question to apply it to just that question. When interaction logging is on, blocked attempts are recorded as `paste_blocked` / `drop_blocked` events (distinct from genuine `paste` / `drop` events), so you can see that a participant tried to paste even though nothing was inserted.

2.9.5 Reviewing and excluding participants

After data collection, the admin panel helps you spot and remove low-quality records:

- **Completion times.** The progress page reports each participant's duration, by condition (see *Monitoring and Exporting Data*).
- **Abandoned participants.** Participants who go inactive past `ABANDONED_MINUTES` are flagged as abandoned and left out of condition balancing (see *Sessions and Participant State*).
- **Manual exclusion.** The progress page has a per-participant exclusion checkbox, and the export can omit excluded or unfinished participants (see *Monitoring and Exporting Data*).

2.10 Testing Your Study Before Launch

A study collects data from paid participants exactly once — a broken page or a missing database column discovered mid-collection can't be fixed retroactively. The checks below catch those problems while the only participant is you. They use pieces documented elsewhere (the admin panel, the debug toolbar, the export); this page puts them in launch order.

2.10.1 1. Preview every questionnaire

From the admin panel (`/admin`), open **Preview Questionnaire** for each JSON file. The preview renders the questionnaire as participants will see it, reports JSON syntax errors, and offers to add database columns when question IDs have changed. Check the wording, the scale labels, and that `show_if` questions appear and disappear when you change the answers they depend on. Startup problems in questionnaires, tables, or the config also appear as warning and notice pills in the admin navbar — resolve those first (see *Setup Diagnostics*).

2.10.2 2. Check the flow against your protocol

`/admin/preview_procedure` draws a flowchart of your `PAGE_LIST`, including the per-condition routes from `conditional_routing` blocks. Compare it against your study protocol — every condition path should reach an end page. The flowchart also works as a figure for an ethics application.

2.10.3 3. Walk through the experiment — once per condition

Run the project in debug mode (`BOFS run config.toml -d`) and complete the study as a participant. The `-d` flag adds a toolbar with navigation controls and session information.

- Repeat the walkthrough until you have seen every condition. Condition assignment balances automatically, so consecutive runs rotate through them; the debug toolbar shows which condition you received.
- To start over, open a private/incognito window or visit `/restart`, which clears your session.
- Answer every required question and try to skip them too — confirm validation blocks the submit.
- If the study is multi-session, fake a return visit with a test external ID; see the testing notes in *Longitudinal Experiments*.

2.10.4 4. Verify the data landed

After each walkthrough:

- Open your own run in the participant detail view (click your participant ID on `/admin/progress`). Every page should show as Completed with plausible timings, and each questionnaire's responses should match what you entered.
- Preview the export (`/admin/export`) in HTML and check the column headers — one column per question ID, plus any `participant_calculations`. Analysis scripts are usually written against these headers, so this is the moment to catch a misnamed ID.
- If a custom task posts to a table, inspect it via `/admin/table_view/<table_name>` and confirm one row per trial with the participant ID attached.

2.10.5 5. Delete the test data

Your walkthroughs are now rows in the database, and they will appear in the export and the condition-balancing counts alongside real participants. Clear them before recruiting:

- *During development*, the simplest reset is deleting the `.db` file (SQLite) and restarting — the schema is recreated on the next run.

- Once the deployment is final, use `/admin/database_download` to take a backup, then `/admin/database_delete` to clear all rows while keeping the table structure. The deletion is irreversible; take the backup first.
- If you need one more sanity check after clearing, do a single walkthrough and delete again — or mark your own run with the per-participant **Excluded** checkbox on `/admin/progress`, which keeps the row out of counts and can be filtered from the export.

2.10.6 6. Pre-launch checklist

Immediately before sharing the study URL:

- `ADMIN_PASSWORD` is strong and unique — it protects all participant data.
- The server setup matches *Deploying to a Server*: HTTPS via a reverse proxy, `BEHIND_REVERSE_PROXY = true`, and BOFS running without `-d`.
- The completion code or `outgoing_url` fires on the end page (walk through one final time on the production URL).
- External IDs are captured — check the `external_id` column after your production walkthrough (see *Recruiting via MTurk or Prolific* for the platform-specific setup).
- Test rows are deleted or excluded.

Running a small paid pilot (a handful of participants) before the full launch catches what solo walkthroughs can't: real devices, real network conditions, and instructions that only confuse people who aren't you.

2.11 Customizing the Appearance

BOFS pages use a default stylesheet defined with CSS custom properties — reusable named values declared once at the top of the stylesheet, so changing one updates it everywhere it's used. Two levels of customization are available without editing any page templates: a single config setting for the header color, and a full stylesheet override for broader changes. Template-level customization (changing the HTML itself) is covered in *Templates and Jinja2*.

2.11.1 Quick: header color from config

Set `HEADER_COLOR` in your project's `.toml` file to change the title bar background without creating a custom stylesheet:

```
HEADER_COLOR = "#003366"   # hex, named color, rgb(), rgba(), hsl(), or hsla()
```

When `HEADER_COLOR` is unset, the default green from `style.css` is used. The accent also trickles into the question and navigation card borders (`--bofs-card-border` uses `color-mix()` to produce a subtle tint), so a single color setting keeps the page visually cohesive.

See *Configuration Reference* for the full setting description.

2.11.2 Override the stylesheet

Placing a `style.css` in your project's `static/` directory replaces the BOFS default entirely. BOFS detects this file at startup and serves it in place of its own.

Copy the default as a starting point so you keep everything that already works:

```
cp /path/to/BOFS/static/style.css ./static/style.css
```

Then edit `./static/style.css` and restart BOFS (`BOFS run config.toml -d`).

Note

The exact path to the BOFS `static/` directory depends on how you installed the package. In a virtual environment it is typically something like `venv/lib/python3.x/site-packages/BOFS/static/style.css`. You can also find it by running `python -c "import BOFS, os; print(os.path.join(os.path.dirname(BOFS.__file__), 'static', 'style.css'))"` from the project directory.

Warning

A copied stylesheet will not receive updates automatically when BOFS is upgraded. Note the changes you make so you can reapply them to a fresh copy after an upgrade.

2.11.3 CSS custom properties

The BOFS stylesheet declares its custom properties on `:root`, the stylesheet's top level, where they apply to every page. Overriding a subset of them in your `style.css` is the most targeted way to restyle colors, fonts, and layout. When changing colors, keep the contrast between text and its background high — participants complete studies on a wide range of screens, and low-contrast text affects readability.

The full set of variables from the default stylesheet:

```
:root {
    /* Layout */
    --contents-width: 967px;

    /* Colors */
    --top-bar-color: #8CB737;
    --bs-border-color: black;

    /* Card surfaces */
    --bofs-card-bg: #ffffff;
    --bofs-card-border: color-mix(in srgb, var(--bofs-accent) 20%, #e2e5e7);

    /* Typography (rem values at 16px base) */
    --font-size-main: 1rem;             /* 16px */
    --font-size-h1: 1.5rem;             /* 24px */
    --font-size-h2: 1.3125rem;          /* 21px */
    --font-size-h3: 1.0625rem;          /* 17px */
    --font-size-h4: 0.9375rem;          /* 15px */
    --question-title-font-size: 1.0625rem;
    --question-instructions-font-size: 0.9375rem;
}
```

To change only a few values, have your `static/style.css` import the default and redeclare just the variables you want to change:

```
@import url('/BOFS_static/style.css');

:root {
    --top-bar-color: #003366;
    --font-size-main: 1.125rem;
```

(continues on next page)

(continued from previous page)

```
    --contents-width: 800px;
}
```

This keeps everything else — borders, question layout, mobile breakpoints — from the default.

> **Note**
>
> The `@import` approach works when your `static/style.css` is meant to layer on top of the default rather than replace it. If you are making extensive changes, copying the full default and editing it directly may be easier to maintain.

2.11.4 Further customization

For changes that require modifying HTML structure — overriding the base template, changing how question types render, or adding custom fonts and images — see *Templates and Jinja2*. That page covers:

- Template lookup order (project `templates/` vs. BOFS defaults)
- Overriding `template.html` to change the page layout
- Custom question type templates
- Serving custom assets (fonts, images, JavaScript)

CHAPTER

THREE

THE BOFS ARCHITECTURE

This section is for developers who want to go past the built-in patterns — write Python routes, render dynamic templates, query the database directly, or understand how sessions work. If you're configuring an experiment with TOML, JSON, and HTML and don't need any of that, you can focus on the "Building Your Experiment" section and "Reference" section.

3.1 What BOFS actually is

Under the hood, BOFS is a Python web application built on Flask, with an extended `BOFSFlask` class on top that adds three things:

- **TOML configuration loading**, so settings live in human-friendly `.toml` files instead of Python.
- **Blueprint auto-discovery**, so a folder containing a `views.py` at your project root is registered automatically without an explicit Python import.
- **Database-backed sessions**, replacing Flask's default file-based session interface so participants don't lose state when a worker process restarts.

If you've used Flask before, that's the whole story: a Flask app with conventions baked in. If you haven't, you don't need to learn Flask first — most projects never touch it directly. But if you do start writing Python, what you're writing *is* Flask, and the same Flask documentation applies.

3.2 The request lifecycle

When a participant visits a URL, here's the order of operations:

1. **Session lookup.** BOFS's database-backed session middleware (`BOFSSession.py`) reads the session cookie and loads the corresponding row. New visitors get a fresh session.
2. **Routing.** Flask matches the URL against the registered routes — those defined in BOFS's built-in default blueprint (`BOFS.default`), the admin blueprint (`BOFS.admin`), and any project-level blueprints discovered at startup.
3. **Page-flow validation.** Routes that participate in `PAGE_LIST` use the `@verify_correct_page` decorator to confirm the URL matches the participant's current position. A mismatch redirects them back to where they should be.
4. **View execution.** The route function runs. It may read questionnaire data from the participant record, query custom tables, render a template, or process a form submission.
5. **Template rendering.** Templates extend `BOFS/templates/template.html` and inherit a context that already includes `participant`, `session`, `config`, `debug`, and `flat_page_list`.
6. **Storage.** Form submissions write to a participant-specific row in the questionnaire's auto-generated table; `$.post` calls to `/table/<name>` write to a custom table.

The participant then gets a response, advances, and the loop repeats.

3.3 Where BOFS's code ends and yours begins

BOFS exposes a small number of well-defined extension points. Each has its own page later in this section.

Extension point	What you write	Where it lives
Questionnaires	JSON files	`questionnaires/*.json` at the project root, or `<blueprint>/questionnaires/*.json`
Custom tables	JSON files	`tables/*.json` at the project root, or `<blueprint>/tables/*.json`
Static pages	HTML files	`templates/instructions/`, `templates/simple/`, `templates/custom/`
Custom routes	Python (Flask blueprint)	A folder with `__init__.py` and `views.py` at the project root
Template overrides	HTML/Jinja2	`templates/` files matching BOFS's default template names
Visual styling	CSS	`static/style.css`

A project that uses only the first three rows is a "no Python" project. Adding the fourth opens up server-side logic. Templates and CSS overrides apply at any level.

3.4 The project folder as a configuration surface

A BOFS project is a folder. The contents of that folder, plus any Python blueprints inside it, define the entire experiment. Nothing about the experiment is configured in BOFS itself. This is what makes a BOFS project portable — copying the folder copies the experiment.

A representative project at the framework level looks like:

```
my_study/
├── config.toml                  # settings, PAGE_LIST, conditions
├── consent.html                 # consent form
├── questionnaires/
│   └── *.json                   # one per survey page
├── templates/
│   ├── instructions/*.html      # static instruction pages
│   ├── simple/*.html            # custom pages with BOFS chrome
│   ├── custom/*.html            # custom pages without BOFS chrome
│   ├── questions/*.html         # custom question type templates (optional)
│   └── template.html            # base template override (optional)
├── tables/
│   └── *.json                   # custom database tables
├── static/
│   ├── style.css                # CSS overrides (optional)
│   ├── *.js                     # JavaScript task files
│   └── images, audio, video, etc.
└── my_blueprint/                # custom Python blueprint (optional)
    ├── __init__.py
    ├── views.py
    ├── templates/               # blueprint-scoped templates
    ├── static/                  # blueprint-scoped static files
```

(continues on next page)

(continued from previous page)

```
    ├── tables/                  # blueprint-scoped custom tables
    └── questionnaires/          # blueprint-scoped questionnaires
```

Each subdirectory is auto-discovered by BOFS at startup. You don't import anything from BOFS in your blueprint's `__init__.py`; you just declare a Flask `Blueprint` object and write routes in `views.py`. BOFS finds it.

3.5 How the pieces connect

A complete request flow, end-to-end:

1. `config.toml` declares `PAGE_LIST` — the ordered sequence of pages.
2. The participant arrives, consent is recorded (or skipped, depending on the first-page route), and a row is created in the `Participant` table.
3. The `@verify_correct_page` decorator routes them through the pages declared in `PAGE_LIST` in order. Each page is one of: a built-in route (`consent`, `external_id`, `end`), a questionnaire, a static HTML page (`instructions/`, `simple/`, `custom/`), or a custom blueprint route.
4. Form submissions populate per-questionnaire database tables (auto-generated from the JSON definitions). JavaScript tasks POST to `/table/<name>` and populate custom tables (auto-generated from your `tables/*.json` definitions).
5. The `/admin` panel reads from these same tables to display progress, preview questionnaires, and export data.

The rest of this section walks each layer in turn:

- *Blueprints and Routes* — Flask blueprints in BOFS terms, `views.py`, route decorators.
- *Templates and Jinja2* — template inheritance, Jinja2 features, all available variables, override patterns.
- *Using Participant Data* — the `participant` object, accessing questionnaire and table data, expressions in templates.
- *The Database Layer* — SQLAlchemy in BOFS, built-in models, custom table internals, querying, export field mechanics.
- *Sessions and Participant State* — database-backed sessions, session lifecycle, recovery, IP binding.

See also: the advanced example in the example projects repo — a worked project that touches most of these extension points in one place.

3.5.1 Blueprints and Routes

A blueprint is a folder of Python code that BOFS auto-discovers at startup, registers as a Flask blueprint, and integrates into the experiment flow. Use a blueprint when an experiment page needs server-side logic — generating stimuli on the fly, processing form submissions, writing to a custom table from Python, or talking to an external service.

What "blueprint" means here

In Flask, a Blueprint is a way to group related routes, templates, and static files into a reusable unit. BOFS uses Flask's blueprint system, plus a discovery convention: at startup, BOFS walks your project root looking for folders that contain a `views.py`. Each one is registered automatically — you don't import or list anything.

The auto-discovery rule is "folder with a `views.py`." A folder containing only `__init__.py` (or only templates) is not picked up.

Blueprint layout

A blueprint named `my_blueprint` lives at the project root:

```
my_study/
├── config.toml
├── consent.html
└── my_blueprint/
    ├── __init__.py        # marks the folder as a Python package; can be empty
    ├── views.py           # route definitions (required for discovery)
    ├── templates/         # Jinja2 templates, scoped to this blueprint
    ├── static/            # static files, served at /my_blueprint/<path>
    ├── tables/            # custom table definitions, scoped to this blueprint
    └── questionnaires/    # questionnaire JSON files, scoped to this blueprint
```

The `templates/`, `static/`, `tables/`, and `questionnaires/` subdirectories inside a blueprint work the same way as the equivalent directories at the project root. Files inside them are merged into the project's overall template lookup, static-file serving, table registry, and questionnaire registry — see *Templates and Jinja2*, *The Database Layer*, and *Adding Survey Questions* for details.

The views.py boilerplate

A minimal `views.py` looks like this:

```
from flask import Blueprint, render_template, request, redirect, session
from BOFS.util import verify_correct_page, verify_session_valid
from BOFS.globals import db

# The variable name should match the folder name.
my_blueprint = Blueprint(
    'my_blueprint', __name__,
    static_url_path='/my_blueprint',
    template_folder='templates',
    static_folder='static',
)
```

The `Blueprint()` constructor arguments rarely need adjusting beyond the name. The rest of the file is your route definitions.

Creating routes

A route is a Python function decorated with `@<blueprint>.route`. BOFS adds two of its own decorators on top of Flask's:

```
@my_blueprint.route("/task", methods=['POST', 'GET'])
@verify_correct_page
@verify_session_valid
def task():
    if request.method == 'POST':
        log = db.answers()                                  # custom table from /
↪tables/answers.json
        log.participantID = session['participantID']
        log.answer = request.form['answer']
        db.session.add(log)
        db.session.commit()
```

(continues on next page)

(continued from previous page)

```
        if log.answer.lower() == "linux":
            return redirect("/redirect_next_page")
        return render_template("task.html", incorrect=True)

    return render_template("task.html")
```

Three decorators stacked, in order:

- `@my_blueprint.route("/task", methods=['POST', 'GET'])` registers the URL and accepts both GETs (loading the page) and POSTs (form submission).
- `@verify_correct_page` (from `BOFS.util`) prevents participants from visiting this page out of order. They must reach it through `PAGE_LIST`.
- `@verify_session_valid` (from `BOFS.util`) redirects participants to the first page of `PAGE_LIST` if their session is missing the expected fields.

Both BOFS decorators are no-ops on routes that aren't reached through `PAGE_LIST` — internal admin endpoints, for example. See *Helper Functions* for the full decorator reference.

GET vs POST

A typical task route handles both: GET renders the page, POST processes the form submission. The pattern in the example above (`if request.method == 'POST': ... return render_template ...`) is standard Flask.

Writing to a custom table

Custom tables defined in `tables/*.json` are accessible as ORM classes on `db`. The class name matches the JSON filename (without `.json`). Create an instance, set attributes, add to the session, and commit — standard SQLAlchemy:

```
log = db.answers()
log.participantID = session['participantID']
log.answer = request.form['answer']
db.session.add(log)
db.session.commit()
```

The `participantID` and `timeSubmitted` columns are added to every custom table automatically, but Python code is responsible for populating `participantID` (the JS API populates it from the session automatically — Python doesn't). See *The Database Layer* for the broader picture and *Custom Tables* for column types and export keys.

Redirecting participants

Three built-in routes handle navigation:

- `/redirect_next_page` — advance to the next entry in `PAGE_LIST`.
- `/redirect_to_page/<path>` — jump to a specific page.
- `/redirect_from_page/<path>` — used internally; rarely called from your code.

Use `redirect("/redirect_next_page")` rather than hard-coding the next URL — that way the route works regardless of how the experiment's `PAGE_LIST` is reorganized later.

The full route reference is at *Built-in Routes*.

Templates in blueprints

Templates live in `my_blueprint/templates/` and are rendered with `render_template("filename.html", ...)`. They typically extend BOFS's base template:

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

{% block contents %}
    <h2>Click the button when you're ready.</h2>
    <form method="POST">
        <input type="text" name="answer">
        <button type="submit">Submit</button>
    </form>
    {% if incorrect %}<p>Try again.</p>{% endif %}
{% endblock %}
```

Static files inside the blueprint are served at `/my_blueprint/<path>`. Reference them either with the URL path directly (`/my_blueprint/myscript.js`) or via Flask's `url_for`:

```
<script src="{{ url_for('my_blueprint.static', filename='myscript.js') }}"></script>
```

Either form works. The `url_for` form survives a base-URL change (e.g., when `APPLICATION_ROOT` is set for hosting at a subpath); the literal path doesn't.

Template lookup order, override patterns, and the full set of available template variables are covered in *Templates and Jinja2*.

Showing data on the participant detail page

The admin panel's participant detail view shows each page the participant has visited and (for questionnaire pages) what they submitted. To make a custom-page route surface its data the same way, decorate it with `@page_tables('<table_name>')`:

```
from BOFS.util import page_tables

@my_blueprint.route("/task", methods=['POST', 'GET'])
@verify_correct_page
@verify_session_valid
@page_tables('answers')
def task():
    ...
```

The participant detail page will run the `answers` table's calculated export fields scoped to the participant and display the result inline with the page entry. The decorator can list multiple tables. See *Custom Tables* for export-field syntax and *Helper Functions* for the decorator reference.

Reading questionnaire data from a route

To read the participant's questionnaire responses inside a route, look them up by session ID and use the `questionnaire()` method:

```
from BOFS.globals import db
from flask import session

@my_blueprint.route("/results")
```

(continues on next page)

(continued from previous page)

```
@verify_correct_page
@verify_session_valid
def results():
    participant = db.Participant.query.get(session['participantID'])
    demographics = participant.questionnaire('demographics')

    return render_template(
        "results.html",
        age=demographics.age,
        condition=participant.condition,
    )
```

For repeated questionnaires (pre/post designs), pass the tag as the second argument: `participant.questionnaire('mood', 'pre')`. The full participant API is at *Participant Data API*. For the broader question of "how do I show participant data in templates?", see *Using Participant Data*.

Activity polling on custom pages

Custom pages reached through `PAGE_LIST` automatically poll `/user_active` every 30 seconds. The poll updates the participant's last-active timestamp, which the admin panel uses to mark a participant as in-progress versus abandoned (after `ABANDONED_MINUTES` of silence).

Polling is fine for almost every page, but two cases need to opt out:

- A page that uses `window.beforeunload` to capture data before the participant leaves — the poll's network activity can race with the unload handler.
- A long-running task that explicitly manages its own activity heartbeat through a custom table.

Decorate the route with `@suppress_activity_polling` to disable the script for that page:

```
from BOFS.util import suppress_activity_polling

@my_blueprint.route("/long_task")
@verify_correct_page
@verify_session_valid
@suppress_activity_polling
def long_task():
    return render_template("long_task.html")
```

Implementation note: the decorator sets `g.bofs_skip_activity_polling = True`, which the response-writing layer reads to skip injecting the polling `<script>`.

See also

- The advanced example — a worked blueprint with a task page, a custom table, condition-aware routing, and `@page_tables` integration.
- The forthcoming custom_blueprint_example — a smaller, single-purpose blueprint walkthrough specifically for this page.
- *Helper Functions* for the full decorator reference.
- *Built-in Routes* for the navigation routes.

3.5.2 Templates and Jinja2

Every page BOFS renders — built-in pages (consent, end, questionnaires) and your own custom ones — goes through Jinja2, the template engine that comes with Flask. This page covers what BOFS does with Jinja2: the lookup order, the base template and its blocks, the variables every template has access to, and the patterns for overriding any of it. Read it when you want to change what participants see beyond content — adding an IRB number or institutional logo to every page, showing condition-specific text, or overriding the consent form layout.

Lookup order

When BOFS renders a template by name (`render_template("foo.html")`), Flask searches three places in this order:

1. **Project ``templates/`` ** at the project root. This is where your overrides go.
2. **Blueprint ``templates/`` directories**, one per discovered blueprint. Templates here are scoped to that blueprint by default but available to other code via the same lookup.
3. **BOFS's bundled defaults** under `BOFS/templates/`. The fallback for everything you haven't overridden.

The first match wins. Dropping `templates/consent.html` into your project replaces BOFS's default consent template; nothing else changes.

The base template (`template.html`)

BOFS ships with a `template.html` that defines the page layout — the `<html>` shell, the header bar, the content area, the script tags, and (when enabled) the breadcrumb. Every BOFS-rendered page either *is* this template or extends it:

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

{% block contents %}
    <h2>Your content here.</h2>
{% endblock %}
```

The blocks defined in the bundled `template.html` are:

- `head` — additional `<head>` content (extra stylesheets, `<meta>` tags).
- `top` — content rendered above the page title (rare).
- `contents` — the main page body (this is the block most pages override).
- `bottom` — content rendered below the page body, before the activity-polling script.
- `scripts` — additional `<script>` tags at the end of `<body>`.

You don't need to know the bundled template's full source to use it — overriding `contents` is enough for almost everything. To customize the page layout itself (header, footer, IRB-required watermark), copy `template.html` into your project's `templates/` directory and edit your copy.

Available template variables

BOFS injects four variables into every template's context, plus Flask's standard `session`:

- `participant` — the current participant object, or `None` outside a session (admin previews, error pages). Guard with `{% if participant %}` if your template can render in those contexts.
- `session` — the Flask session dict, populated as the participant moves through the experiment.
- `config` — the project's TOML config, accessed as `config['KEY']` or `config.KEY`.
- `debug` — `True` when BOFS is running with `-d`.

- `flat_page_list` — the participant's filtered page sequence, with `show_if` and conditional routing already applied.

Each is detailed in *Participant Data API*. The most common pattern is reading a questionnaire response or branching on the assigned condition:

```
{% if session['condition'] == 1 %}
    <p>Control instructions go here.</p>
{% elif session['condition'] == 2 %}
    <p>Treatment instructions go here.</p>
{% endif %}

<p>You answered: {{ participant.questionnaire('demographics').age }}.</p>
```

For a usage-oriented walkthrough of these variables, see *Using Participant Data*.

Jinja2 features beyond `{{ }}` and `{% if %}`

Most templates only need variable substitution and conditionals. Three more Jinja2 features come up regularly:

Loops — iterating over a list:

```
<ul>
{% for entry in flat_page_list %}
    <li>{{ entry }}</li>
{% endfor %}
</ul>
```

Variables in templates — assigning a name once for reuse:

```
{% set demo = participant.questionnaire('demographics') %}
<p>You are {{ demo.age }} years old.</p>
<p>You identified as {{ demo.gender }}.</p>
```

Filters — transformations after the `|`:

```
<p>{{ participant.questionnaire('feedback').comments | length }} characters of␣
↪feedback.</p>
<p>Started at: {{ participant.timeStarted | string }}</p>
```

The full filter list is in Jinja2's documentation.

Template override patterns

There are four levels of override, from least to most invasive:

1. CSS-only changes. Drop `static/style.css` in your project. See *Customizing the Appearance*.

2. Override a page template. Copy a single page template — `consent.html`, `end.html`, `questionnaire.html` — from BOFS's default into your project's `templates/` and edit it. The other pages stay unchanged. The template files BOFS ships are:

```
BOFS/templates/
├── template.html              # base template (header, layout)
├── consent.html               # consent form page
├── external_id.html           # external ID collection
├── questionnaire.html         # questionnaire page
```

(continues on next page)

(continued from previous page)

```
├── questionnaire_macro.html   # rendering helpers used by questionnaire.html
├── instructions.html          # instruction page wrapper
├── simple.html                # simple-page wrapper
├── end.html                   # completion page
└── questions/                 # one file per built-in question type
    ├── radiolist.html
    ├── radiogrid.html
    ├── slider.html
    └── ...
```

3. Override the base template. Copy `template.html` to `templates/template.html` and edit it to add a study-branded header, footer, IRB number, or institutional logo. Every page that extends `template.html` (which is most of them) inherits the change.

A minimal customized base might look like:

Listing 1: templates/template.html

```
{% from "macros.html" import adminControls, checkUserActive %}
<!DOCTYPE html>
<html>
<head>
    <title>{{ config['TITLE'] }}</title>
    <link rel="stylesheet" href="{{ url_for('BOFS_static', filename='bootstrap.min.css
↪') }}">
    <link rel="stylesheet" href="{{ style_url }}">
    <script src="{{ url_for('BOFS_static', filename='js/jquery-3.7.1.min.js') }}"></
↪script>
    {% block head %}{% endblock %}
</head>
<body>
    <header class="study-header">
        <h2>Psychology Department Research Study</h2>
        <p>IRB Protocol #2026-001</p>
    </header>

    <main class="content">
        {% block top %}{% endblock %}
        {% block contents %}{% endblock %}
        {% block bottom %}{% endblock %}
    </main>

    <footer>
        <p>Questions? Contact research@example.edu</p>
    </footer>

    {{ adminControls() }}
    {{ checkUserActive() }}
</body>
</html>
```

4. Custom question types. A `templates/questions/<type>.html` file in your project (or in a blueprint) defines a new question type. Set `"questiontype": "<type>"` on a question to use it. The full mechanics — what variables the template receives, naming rules, the multiple-IDs pattern for templates that emit several form fields — are documented

in *Questionnaire Properties*.

> **Warning**
>
> Customizations made by overriding bundled templates may stop working when BOFS is upgraded if the upstream templates change. Track the changes you make so you can re-apply them on a fresh copy after upgrading.

Adding custom assets

Anything in your project's `static/` directory is served at `/static/<path>`. The same applies to a blueprint's `static/` directory, served at `/<blueprint_name>/<path>`.

For custom fonts, declare `@font-face` in `static/style.css`:

```
@font-face {
    font-family: 'UniversityFont';
    src: url('/static/fonts/UniversityFont.woff2') format('woff2');
    font-weight: normal;
}

body {
    font-family: 'UniversityFont', 'Segoe UI', Arial, sans-serif;
}
```

For images, audio, video, PDFs, downloadable consent forms, JavaScript libraries — drop them in `static/` and reference them with their URL path:

```
<img src="/static/images/stimulus.png" alt="...">
<a href="/static/consent.pdf">Download consent form</a>
<script src="/static/p5.min.js"></script>
```

See also

- *Customizing the Appearance* for the no-code styling options (`HEADER_COLOR`, CSS-only overrides).
- *Participant Data API* for the full template-variable reference.
- *Using Participant Data* for usage-oriented examples of reading data in templates.
- *Questionnaire Properties* for custom question type templates.

3.5.3 Using Participant Data

This page covers how to read participant data — questionnaire responses, custom-table rows, condition assignment, session state — from inside templates and Python routes. The exhaustive method-level reference is at *Participant Data API*; here we focus on the patterns you'll actually reach for.

The participant object

The current participant is available as the `participant` variable in every BOFS-rendered template, and as `db.Participant.query.get(session['participantID'])` in custom blueprint routes.

The participant object has a small set of database-backed attributes (`participantID`, `condition`, `externalID`, `timeStarted`, `finished`, `code`) and a few methods that return their related data. `externalID` is also accessible as `mTurkID` — both names refer to the same column, with `mTurkID` kept as an alias for backward compatibility.

- `participant.questionnaire(name, tag="")` — the response row, with each field accessible as an attribute.
- `participant.has_questionnaire(name, tag="")` — boolean; was the questionnaire submitted at all.
- `participant.questionnaire_interactions(name, tag="")` — the interaction event log (when `LOG_QUESTIONNAIRE_INTERACTIONS = true`).
- `participant.table(name)` — a `TableAccessor` for a custom JSONTable; iterates rows, reads export aggregates.
- `participant.evaluate(expression)` — runs an expression DSL (the same one used by `show_if`) against the participant's data.

Reading questionnaire responses

The `questionnaire()` method returns a row whose fields are accessible as attributes. The field names are the question IDs from the questionnaire JSON:

```
{% set demo = participant.questionnaire('demographics') %}
<p>Age: {{ demo.age }}, Education: {{ demo.education }}.</p>
```

If the participant hasn't submitted that questionnaire yet, the call still returns a row — one with default values (empty strings, zeros). To distinguish between "not yet submitted" and "submitted but empty," use `has_questionnaire()`:

```
{% if participant.has_questionnaire('feedback') %}
    Feedback received: {{ participant.questionnaire('feedback').comments }}
{% else %}
    Feedback not yet collected.
{% endif %}
```

Tagged (repeated) questionnaires

For pre/post designs and longitudinal studies, the same questionnaire is submitted multiple times under different tags (see *Setting Up Your Page Flow* for tagging in PAGE_LIST). Pass the tag as the second argument:

```
{% set pre  = participant.questionnaire('mood', 'pre') %}
{% set post = participant.questionnaire('mood', 'post') %}
<p>Mood improved from {{ pre.score }} to {{ post.score }}.</p>
```

Reading custom-table data

`participant.table(name)` returns a `TableAccessor` that handles three things:

1. **Iterate the raw rows** the participant submitted to that table.
2. **Read the calculated export fields** declared in the table's `exports` block.
3. **Compute scoped aggregates** — every export is automatically scoped to this participant.

```
{% set trials = participant.table('trial_data') %}

{# Iterate raw rows #}
<ul>
{% for row in trials %}
    <li>Trial {{ loop.index }}: {{ row.score }}</li>
{% endfor %}
</ul>
```

(continues on next page)

(continued from previous page)

```
{# Read scalar export aggregates #}
<p>Total trials: {{ trials.trial_count }}</p>
<p>Average RT: {{ trials.avg_rt | round(1) }} ms</p>
```

For exports declared with `group_by`, the accessor returns a dict keyed by the group value:

```
{% set scores = participant.table('game') %}
<ul>
{% for level, deaths in scores.totalDeathCount.items() %}
    <li>{{ level }}: {{ deaths }} deaths</li>
{% endfor %}
</ul>
```

The full `TableAccessor` API — `.rows`, `.exports`, indexing, caching behavior — is in *Participant Data API*. The export-key syntax is in *Custom Tables*.

Running expressions from templates

`participant.evaluate(expr)` runs an expression in the same DSL `show_if` and `participant_calculations` use. Useful for one-off calculations that are inconvenient to add to a questionnaire's `participant_calculations` block:

```
{% set total = participant.evaluate("scale_1 + scale_2 + scale_3") %}
<p>Your composite score: {{ total }}.</p>
```

Returns `None` on parse error or when a referenced field doesn't exist. The full expression syntax is at *Expressions: Calculations and Conditional Display*.

Session variables

Flask's `session` is available in every template. BOFS populates it with six fields:

- `session['participantID']` — the current participant's ID.
- `session['condition']` — assigned condition number (1+, or 0 if unassigned).
- `session['currentUrl']` — the page the participant should be on.
- `session['externalID']` — the external ID (regardless of source: MTurk, Prolific, manual entry). Also available as `session['mTurkID']`, an alias kept for backward compatibility with code written before the rename.
- `session['source']` — the recruitment channel string (e.g. `"prolific"`, `"reddit"`, `"email"`), or absent when none was provided.
- `session['code']` — the completion code, populated near the end.

Branching on condition is the most common use:

```
{% if session['condition'] == 1 %}
    <p>Control instructions go here.</p>
{% elif session['condition'] == 2 %}
    <p>Treatment instructions go here.</p>
{% endif %}
```

Configuration access

The project's TOML config is exposed as the `config` template variable. Access fields with bracket or attribute notation:

```
<p>Welcome to {{ config['TITLE'] }}.</p>
<p>Contact: {{ config.SUPPORT_EMAIL }}</p>
```

Custom (non-BOFS) keys you set in `config.toml` are available the same way — useful for project-specific values you want to thread through templates without hard-coding them.

Reading data in custom blueprint routes

In a Python route, look up the participant by session ID:

```
from BOFS.globals import db
from flask import session, render_template

@my_blueprint.route("/results")
@verify_correct_page
@verify_session_valid
def results():
    participant = db.Participant.query.get(session['participantID'])
    demo = participant.questionnaire('demographics')

    return render_template(
        "results.html",
        name=demo.first_name,
        condition_label=["Control", "Treatment"][session['condition'] - 1],
    )
```

The participant object you get from the database query has all the same methods (`questionnaire()`, `table()`, `evaluate()`, etc.) as the template variable.

Worked example: confirmation page

A confirmation step that shows participants what they submitted before final review:

Listing 2: templates/simple/review.html

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

{% block contents %}
    {% set demo = participant.questionnaire('demographics') %}
    {% set survey = participant.questionnaire('survey') %}

    <h2>Please review your answers</h2>

    <h3>About you</h3>
    <ul>
        <li>Age: {{ demo.age }}</li>
        <li>Education: {{ demo.education }}</li>
    </ul>

    <h3>Survey responses</h3>
    <ul>
```

(continues on next page)

(continued from previous page)

```
        <li>Question 1: {{ survey.q1 }}</li>
        <li>Question 2: {{ survey.q2 }}</li>
    </ul>

    <a href="/redirect_next_page">Looks good — continue</a>
    <a href="/redirect_to_page/questionnaire/survey">Go back and edit</a>
{% endblock %}
```

Worked example: performance feedback from multiple sources

A debrief page combining a questionnaire scale score, a custom-table aggregate, and a condition-specific message:

Listing 3: templates/simple/debrief.html

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

{% block contents %}
    {% set scores = participant.table('trial_data') %}
    {% set self_report = participant.questionnaire('post_task') %}

    <h2>Your results</h2>

    <p>You completed {{ scores.trial_count }} trials with an average reaction
    time of {{ scores.avg_rt | round(1) }} ms.</p>

    <p>You rated your effort {{ self_report.effort }} out of 7
    and your confidence {{ self_report.confidence }} out of 7.</p>

    {% if session['condition'] == 1 %}
        <p>Thank you for completing the control task.</p>
    {% else %}
        <p>The training task is part of an ongoing study —
        your data will help us refine the next version.</p>
    {% endif %}

    <a href="/redirect_next_page">Continue</a>
{% endblock %}
```

See also

- *Participant Data API* — exhaustive method reference.
- *Expressions: Calculations and Conditional Display* — expression syntax for `evaluate()` and embedded `{{ }}` substitution.
- *Custom Tables* — table definition format and export keys.
- *Templates and Jinja2* — template lookup, blocks, override patterns.
- *Blueprints and Routes* — accessing participant data from a custom Python route.

3.5.4 The Database Layer

BOFS uses SQLAlchemy — Python's standard ORM — for everything that touches the database: built-in tables (Participant, Progress, response rows), questionnaire-derived tables (one per JSON file), and custom tables (one per `tables/*.json`). This page covers how those layers fit together and how to read or write from Python.

The exhaustive reference for custom-table JSON syntax (column types, exports, naming rules) is at *Custom Tables*. This page is the conceptual orientation — it's for researchers writing Python routes that query or insert data directly; if you work only with JSON and TOML, you can skip it.

How BOFS uses SQLAlchemy

A BOFS process holds one global `db` object, exposed as `BOFS.globals.db`. It carries:

- The SQLAlchemy `session` — the unit-of-work used for queries, inserts, and commits.
- The model classes for built-in tables (`Participant`, `Progress`, `QuestionnaireInteraction`, `ResponseLog`).
- One auto-generated model class per loaded questionnaire (named after the questionnaire file).
- One auto-generated model class per loaded custom table (named after the table JSON file).

Custom blueprint code reaches them by import:

```
from BOFS.globals import db
from flask import session

participant = db.Participant.query.get(session['participantID'])
trials = db.trial_data.query.filter_by(participantID=session['participantID']).all()
```

Built-in models

Four SQLAlchemy models are part of BOFS itself, not derived from user JSON. They live in `BOFS/default/models.py`.

- **Participant.** One row per consenting participant. Columns include `participantID` (PK), `condition`, `externalID` (also accessible as `mTurkID`, an alias kept for backward compatibility), `source`, `timeStarted`, `timeEnded`, `finished`, `end_reason`, `code`. The `Participant` class also defines the methods used through the `participant` template variable — `questionnaire()`, `table()`, `has_questionnaire()`, `evaluate()`, `display_duration()`. See *Participant Data API*.
- **Progress.** One row per page transition. Used by the admin panel to show timestamps for each page in a participant's flow.
- **QuestionnaireInteraction.** One row per UI event (focus, blur, change, paste, visibility) when `LOG_QUESTIONNAIRE_INTERACTIONS = true`. For text fields, additional rows record per-input authenticity signals (keystroke counts, paste lengths, focus duration).
- **ResponseLog.** One row per consent submission, keyed to the participant.

You usually don't query these directly — the participant object's methods cover the common cases. Querying is appropriate when you're writing admin tooling or doing custom analytics.

Questionnaire-derived models

Each questionnaire JSON file produces a model class at startup. The class name matches the filename (without `.json`), and the columns are derived from each question's `id` plus type:

- `field` and `radiolist` → `String`

- `num_field` → `Integer`
- `checklist` → `Boolean` (one column per option)
- `radiogrid` → `Integer` (one column per row)
- `slider` → `Integer`
- …and so on

Plus three columns added automatically: `participantID` (FK to `Participant`), `timeStarted`, `timeEnded`, and `tag` for repeated submissions.

Reading from Python is the same as any SQLAlchemy model:

```
demographics_row = db.demographics.query.filter_by(
    participantID=session['participantID']
).first()

age = demographics_row.age
```

The simpler `participant.questionnaire('demographics').age` does this lookup for you in one call. Use the raw SQLAlchemy query when you need filtering across all participants (e.g., for an admin tool).

Custom tables

Every `tables/*.json` file (at the project root or inside a blueprint) becomes a model class at startup. The columns come from the `columns` block in the JSON definition; the export-field aggregates come from the `exports` block. `participantID` and `timeSubmitted` are added automatically.

The *Custom Tables* page covers the JSON format. The patterns most likely to come up in Python:

Insert one row.

```
row = db.trial_data()
row.participantID = session['participantID']
row.score = 42
row.reaction_time = 312.5
db.session.add(row)
db.session.commit()
```

Insert many rows efficiently. `db.session.add_all([...])` plus a single commit:

```
rows = [
    db.trial_data(
        participantID=session['participantID'],
        score=score,
        reaction_time=rt,
    )
    for score, rt in zip(scores, rts)
]
db.session.add_all(rows)
db.session.commit()
```

Query by participant.

```
trials = db.trial_data.query.filter_by(
    participantID=session['participantID']
).order_by(db.trial_data.timeSubmitted).all()
```

Aggregate in SQL. Faster than iterating in Python for large tables:

```
from sqlalchemy import func

avg_rt = db.session.query(
    func.avg(db.trial_data.reaction_time)
).filter_by(participantID=session['participantID']).scalar()
```

For most aggregations, declaring an `exports` block in the table JSON and accessing it via `participant.table('trial_data').avg_rt` is simpler — the SQL is generated for you and the result is cached on the accessor. Direct queries are appropriate for one-off or admin-only computations that don't fit cleanly into a per-participant export.

Calculated export fields in depth

The `exports` block on a table is BOFS's way of defining per-participant aggregates without writing SQL. Each entry takes a `fields` dict (column-name → SQL expression) and optional `filter`, `group_by`, `order_by`, and `having` keys:

```
{
  "columns": {
    "score": {"type": "integer"},
    "reaction_time": {"type": "float"},
    "block": {"type": "string"}
  },
  "exports": [
    {
      "fields": {
        "trial_count": "count(score)",
        "avg_rt": "avg(reaction_time)"
      }
    },
    {
      "filter": "block = 'practice'",
      "fields": {
        "practice_avg": "avg(score)"
      }
    },
    {
      "group_by": "block",
      "fields": {
        "block_avg": "avg(score)"
      }
    }
  ]
}
```

At runtime, BOFS expands this into SQL queries scoped to each participant. The results appear three places:

1. As columns in the admin panel's per-participant CSV export.
2. As attributes on `participant.table('trial_data')` in templates and routes.
3. As reachable references in expressions (`tables.trial_data.avg_rt` in a `show_if` predicate).

Full reference: *Custom Tables*.

Showing table data in the participant detail view

The admin panel's per-participant page shows each page in the participant's flow with whatever data was submitted on that page. Questionnaire pages show responses automatically. Custom-page routes don't, by default — to surface data from a custom-page route, decorate the view with `@page_tables('<table_name>')`:

```
from BOFS.util import page_tables

@my_blueprint.route("/task")
@verify_correct_page
@verify_session_valid
@page_tables('trial_data')
def task():
    ...
```

The decorator can list multiple tables: `@page_tables('trial_data', 'event_log')`. The participant detail page runs each table's calculated export fields scoped to the participant and displays the result inline. See *Helper Functions*.

Custom SQLAlchemy models

JSONTable handles the common case: one flat table per concept, scoped to participants, with simple aggregations on top. When that fits, it fits well — declarative, no Python required. When it doesn't, BOFS exposes the underlying SQLAlchemy directly through a `models.py` file in a blueprint.

This feature is for developers already comfortable with relational databases and SQLAlchemy. Researchers without that background usually do fine with JSONTable; the cases below are where `models.py` adds something the JSONTable shape can't:

- **Relationships between tables.** Foreign keys, one-to-many, many-to-many. JSONTable rows are flat per-participant logs; if you need a stimulus catalog joined to trial responses, that's a relationship.
- **Indexes and database-level constraints.** Compound indexes, unique constraints, check constraints.
- **Custom methods on the model.** Computed properties, validation logic, helper methods that operate on a row.
- **Mapping existing tables.** Reading from a database BOFS didn't create — for example, a stimulus pool that an external tool populates.

The factory pattern

A blueprint's `models.py` exports a `create(db)` function that returns one model class or a list of them. BOFS auto-discovers `models.py` alongside `views.py`, calls `create(db)` at startup, and attaches each returned class to `db` so blueprint code can use it like any other model:

Listing 4: my_blueprint/models.py

```
def create(db):
    class StimulusItem(db.Model):
        __tablename__ = 'stimulus_item'
        id = db.Column(db.Integer, primary_key=True)
        word = db.Column(db.String(64), nullable=False, index=True)
        category = db.Column(db.String(32), nullable=False)
        difficulty = db.Column(db.Float, default=0.5)

    class Trial(db.Model):
        __tablename__ = 'trial'
        id = db.Column(db.Integer, primary_key=True)
```

(continues on next page)

(continued from previous page)

```
        participantID = db.Column(
            db.Integer, db.ForeignKey('participant.participantID'), nullable=False
        )
        stimulus_id = db.Column(
            db.Integer, db.ForeignKey('stimulus_item.id'), nullable=False
        )
        response = db.Column(db.String(64))
        reaction_time = db.Column(db.Float)

        stimulus = db.relationship('StimulusItem')

    return [StimulusItem, Trial]
```

The `db.Model` base, `db.Column`, `db.relationship`, and the type names (`db.Integer`, `db.String`, etc.) are all standard Flask-SQLAlchemy — BOFS doesn't add a layer here, just hands you the configured `db` object.

After startup, the returned classes are available on `db` by class name:

Listing 5: my_blueprint/views.py

```
@my_blueprint.route("/trial")
@verify_correct_page
@verify_session_valid
def trial():
    item = db.session.query(db.StimulusItem).order_by(db.func.random()).first()

    trial = db.Trial(
        participantID=session['participantID'],
        stimulus_id=item.id,
    )
    db.session.add(trial)
    db.session.commit()

    return render_template("trial.html", word=item.word)
```

Schema creation and changes

BOFS calls `db.create_all()` at startup, so a new model's table is created automatically the first time the project runs after `models.py` is added. Schema *changes* — adding a column, changing a type — are not applied automatically; that's the same trade-off as JSONTable. For changes after a project has live data, use Alembic or hand-written migration SQL.

Coexisting with JSONTable and questionnaires

Three sources contribute models to the same `db` namespace and the same database:

- **Questionnaires.** Each `questionnaires/<name>.json` produces a model class named after the filename (`db.demographics` from `demographics.json`).
- **JSONTable.** Each `tables/<name>.json` produces a model class named after the filename (`db.trial_data` from `trial_data.json`).
- **Custom models.** Each class returned from a `create(db)` function in a `models.py` is attached to `db` under its class name.

Pick custom-model names that don't shadow a questionnaire or a JSONTable. `class Demographics(db.Model)` while `questionnaires/demographics.json` exists will collide on `db.Demographics`/`db.demographics` — case differs but the underlying database table name (set by `__tablename__` for custom models, or the filename for questionnaires/JSONTable) needs to be unique across all three.

System tables use the `bofs_` prefix, questionnaires use `questionnaire_`, and JSONTables use `table_`. Custom models should avoid all three prefixes. A name clash (between a JSON file and a system table or an existing model) now produces a startup error with a diagnostic message rather than failing silently.

A typical split: JSONTable for participant-generated data (trial responses, event logs), `models.py` for shared study-level data (stimulus pools, condition manifests, scoring rubrics) that participant data joins against. The two reference each other via foreign keys.

Limitations

- `models.py` must define a `create(db)` function.
- Models are loaded at startup. Adding a new model means restarting the project.
- The `db.<ModelName>` registration uses the class name, so two models with the same name across different blueprints will collide.

Splitting data across multiple databases

A questionnaire or custom table can write to a database other than the project's main one, for example, to keep personally identifiable information (follow-up emails, contact details) in a file separate from the experiment data.

Define the additional database in `config.toml` using TOML dotted keys:

```
SQLALCHEMY_DATABASE_URI = "sqlite:///study.db"
SQLALCHEMY_BINDS.pii = "sqlite:///pii.db"
SQLALCHEMY_BINDS.archive = "sqlite:///archive.db"
```

Then opt a questionnaire or table into that bind by adding a top-level `"database"` field to its JSON:

```
{
  "database": "pii",
  "questions": [
    {"questiontype": "field", "id": "email", "title": "Email"}
  ]
}
```

Without the field, the questionnaire or table writes to the main DB.

Limitations and Considerations

The `Participant` row stays in the main DB regardless of any per-bind config. Rows in a non-default bind link to `Participant` by integer `participantID`, but there is no database-level foreign key across binds. When deleting/clearing databases, failure to clear out/delete any secondary databases may result in data loss, errors, or data integrity issues (i.e., it won't be clear if the stored data is truly associated with your participants).

When writing custom blueprints, `Participant.questionnaire('contact')` and `has_questionnaire('contact')` still work, however, without a valid foreign key, accessing the questionnaire via SQLAlchemy's `participant.questionnaire_<name>` backref will not work.

Each SQLite bind is a separate file. The admin panel's "Download Database" button (under the Database dropdown) bundles every SQLite bind into a single zip. "Delete Database" similarly clears rows from every bind in one shot, after writing a timestamped backup zip to the project root.

Partial state from out-of-band editing (a researcher restoring `main.db` from an old snapshot while `pii.db` keeps the current data, for example) leaves rows on the bound DB whose `participantID` values may no longer exist on the default bind. SQLAlchemy can't enforce a FK across engines, so the framework can't reject the orphan rows outright. Instead, BOFS scans every cross-bind table with a `participantID` column at startup and logs a warning naming any orphan IDs. Rotate or delete the orphans before letting new participants sign up; otherwise the next participant assigned a reused ID will silently inherit the orphan row's data.

Export endpoints are per-bind

The admin `/admin/export/download` endpoint still produces today's single CSV (with everything merged via outer joins). For each bind referenced by at least one questionnaire or table, the export page adds a `/admin/export/download/<bind>` button — that endpoint emits a CSV containing only that bind's questionnaire / table columns, leading with `participantID` as the join key. The PII CSV does not contain the experiment data, and the experiment-only CSV at `/admin/export/download` does not contain PII (when the researcher consumes the per-bind endpoints separately).

Mixing dialects works — the main DB can be SQLite while a bind is Postgres, or vice versa. Each engine is independent, so type compatibility is the researcher's problem; the framework just routes queries.

Automatic schema migration is also per-bind: when a questionnaire or table definition gains a new field, BOFS adds the missing column on the engine that questionnaire's bind points at, using dialect-appropriate `ALTER TABLE` statements (SQLite, PostgreSQL, MySQL, and MariaDB).

Switching from SQLite to PostgreSQL (or another DBMS)

For development, SQLite is built in and easy to use. For production with many concurrent participants, PostgreSQL is an option. It handles concurrent writers cleanly where SQLite serialises them.

The only change is the connection string in `config.toml`:

```
# Development:
SQLALCHEMY_DATABASE_URI = "sqlite:///study.db"

# Production:
SQLALCHEMY_DATABASE_URI = "postgresql://user:password@host:5432/dbname"
```

Schema creation runs at startup either way. Migrating live data between databases is a separate operation — typically a `pg_dump`/`pg_restore` after exporting the SQLite content with `sqlalchemy`-native or third-party tools.

The main limitation of this is that from the admin panel you cannot download the entire database as a file, nor can you delete the database.

See *Deploying to a Server* for production deployment context.

See also

- *Custom Tables* — the JSON definition syntax, all column types, all export keys.
- *Participant Data API* — the methods on the `Participant` model.
- *Helper Functions* — `@page_tables` and other decorators.
- *Blueprints and Routes* — the blueprint context for Python database access.

3.5.5 Sessions and Participant State

Web frameworks use *sessions* to remember who a user is across requests. BOFS uses sessions to track each participant's place in `PAGE_LIST`, their assigned condition, the external ID they arrived with, and a few other fields. This page covers what's in a session, how it's created and torn down, and the configuration knobs around recovery and IP binding. Read it when configuring session recovery for a crowdsourced or longitudinal study, troubleshooting participants who lose their place, or deciding on IP-binding settings before going live.

Database-backed, not file-based

Flask's default session implementation stores session data in a signed cookie or in a temporary file on disk. BOFS replaces it with `BOFSSession`, a database-backed implementation that:

- Persists session state in the project's own database (the same one that holds participant data).
- Survives restarts of the BOFS process — the next request looks the session up by ID and continues.
- Supports the recovery flow used by longitudinal and crowdsourced studies (see below).

The implementation lives in `BOFS/BOFSSession.py`. You don't interact with it directly — Flask's standard `session` proxy works the same way it always does.

What's in a session

Six fields cover the participant lifecycle. Each is populated as the participant moves through `PAGE_LIST`:

- `session['participantID']` — the database PK of the current participant. Set when the participant row is created (on consent or on a `create_participant` route).
- `session['condition']` — the assigned condition number (1+, or 0 if no condition has been assigned). Set at the same time as `participantID` for `consent` and `create_participant`; later for the `_nc` variants if and when the participant hits `assign_condition`.
- `session['currentUrl']` — the page the participant should be on according to `PAGE_LIST`. Updated on every page navigation.
- `session['externalID']` — the external ID, regardless of source. Captured from URL parameters (`PROLIFIC_PID`, `external_id`) on the consent page, or set on the manual `external_id` page. Also available as `session['mTurkID']` — both keys are kept in sync at every BOFS write site for backward compatibility with code written before the rename.
- `session['source']` — the recruitment channel, set from a `?source=` URL parameter. Inferred as `"prolific"` when `PROLIFIC_PID` is present and no explicit source is given. Free-form string; expression code (`show_if = "source == 'prolific'"`) can branch on it. Absent when the participant arrived without any source hint — treat `None` as "unknown source," not as a specific value.
- `session['code']` — the completion code, set near the end of the experiment when `GENERATE_COMPLETION_CODE = true`.

Admin sessions also exist (a separate session row marking an authenticated admin) but don't share these fields — they're orthogonal to participant sessions.

Lifecycle

The session is created on the first POST to a participant-creation route — `consent`, `consent_nc`, `create_participant`, or `create_participant_nc`. Before that, a participant browsing the consent page has only an unauthenticated session row with no `participantID`.

After creation, every page navigation:

1. Reads the session from the database via the cookie's session ID.

2. Compares `session['currentUrl']` against the URL being requested. `@verify_correct_page` redirects to `currentUrl` if they don't match.
3. Updates `session['currentUrl']` after a successful submission, advancing to the next entry in `PAGE_LIST` (with conditional routing and `show_if` applied).

The session ends in one of three ways:

- The participant reaches the `end` page; `session['code']` is populated and the participant's `finished` flag is set.
- `ABANDONED_MINUTES` of silence go by without an activity ping; the participant is treated as abandoned. The session row remains in the database but is not counted for condition balancing (unless `COUNTS_INCLUDE_ABANDONED = true`). Abandoned participants still appear in data exports — filter on the `finished` flag when analysing completed sessions only.
- The participant explicitly visits `/restart`, which clears the session and routes them back to the first page.

Session recovery

Two configuration settings control what happens when a participant returns with an external ID that's already been seen.

- `RETRIEVE_SESSIONS = true` (default) — when an incoming request carries an external ID (URL parameter or manual entry) that matches an existing participant, BOFS loads that participant's session and resumes from their current page. The participant doesn't see consent again; their condition assignment is preserved.
- `ALLOW_RETAKES = false` (default) — a participant whose session is already marked `finished` is blocked from starting over. They see a "you've already completed this study" message.

Both default to a configuration appropriate for crowdsourced single-session studies: workers who closed their browser can resume; double submissions are blocked.

For longitudinal studies, the same defaults apply — recovery is what enables day-2 participants to land directly in their day-2 page sequence rather than redo day 1. See *Longitudinal Experiments*.

Implementation: the recovery logic lives in `BOFS/services/session_recovery.py`. The lookup happens at the consent and external-ID routes, before the form is shown.

IP binding

To reduce session-hijacking risk, BOFS binds sessions to the IP they were created on. Two settings control this:

- `SESSION_BIND_TO_IP_PARTICIPANT = true` (default) — a participant session is invalidated if a request arrives from a different IP than the one that created it. Set this to `false` for studies where participants legitimately switch networks mid-session — for example, mobile users moving between cellular and wifi, where each network change produces a new public IP.
- **Admin sessions are always IP-bound.** There is no opt-out. The trade-off — that an admin moving between networks has to re-authenticate — is accepted because an admin compromise has more impact.

When `BEHIND_REVERSE_PROXY = true`, BOFS reads the client's real IP from the `X-Forwarded-For` header instead of the direct socket connection. Without this, IP binding compares against the reverse proxy's address, which is the same for every client — defeating the binding entirely. See *Deploying to a Server* for the deployment context.

Brute-force protection on admin login is built on the same IP infrastructure. See *Deploying to a Server* for the brute-force settings (`BRUTE_FORCE_PROTECTION`, `BRUTE_FORCE_MAX_ATTEMPTS`, etc.).

Recovering from self-lockout

If you (the admin) get locked out — wrong password attempts from your own IP, or a misconfigured `SESSION_BIND_TO_IP_PARTICIPANT` after a network change — there are three escapes:

- `BRUTE_FORCE_AUTO_TRUST_ADMIN = true` (default) adds successful admin IPs to a persistent allowlist (`admin_trusted_ip` table) so they're exempted from future bans. If you've ever logged in from this IP successfully, you're in.
- `TRUSTED_IPS = ["..."]` is an explicit allowlist that bypasses brute-force protection entirely. Add your office or VPN IP here.
- `BRUTE_FORCE_PROTECTION = false` is the kill-switch. Disable it temporarily to log in, then re-enable. Restart BOFS after the change.

For the per-setting reference, see *Configuration Reference*.

See also

- *Longitudinal Experiments* for the longitudinal/multi-session walkthrough that depends on `RETRIEVE_SESSIONS`.
- *Recruiting via MTurk or Prolific* for how external IDs from MTurk and Prolific feed into the session.
- *Deploying to a Server* for IP binding, brute-force protection, and reverse-proxy configuration.
- *Configuration Reference* for the full per-setting reference.

CHAPTER

FOUR

DEPLOYING TO A SERVER

This page covers production deployment: choosing a web server, configuring a reverse proxy with HTTPS, selecting a database, and securing the admin panel. For local development setup, see *Install BOFS*. For crowdsourcing-platform integration (MTurk, Prolific), see *Recruiting via MTurk or Prolific*.

Warning

Production servers hold participant data and may be subject to IRB requirements, GDPR, HIPAA, or institutional policy — consult your institution's IT and compliance teams before collecting data. Use a strong admin password, terminate TLS, and back up the database regularly.

4.1 Why Not the Development Server?

The built-in development server (`BOFS run config.toml -d`) is single-threaded, has no TLS, has no process manager to restart it after a crash, and exposes debugging information that shouldn't reach participants. Production needs:

- HTTPS, with traffic routed through a reverse proxy
- A process manager (systemd) that restarts BOFS automatically on failure
- A database sized to the study (SQLite for small studies, PostgreSQL for larger ones)
- A strong admin password and a unique `SECRET_KEY`

4.2 System Requirements

Small studies (less than a few dozen concurrent participants) can run on a single shared VM:

- Ubuntu 22.04 or newer (or another recent Linux distribution)
- Python 3.9 or newer
- 1 GB RAM, 1 vCPU, 10 GB or more storage
- SQLite (bundled)

Large studies (50+ concurrent participants) benefit from a dedicated database. Two layouts work:

- *Single server*: 4 GB+ RAM, 2+ cores, PostgreSQL on the same machine.
- *Two servers*: 2 GB / 2 cores for the BOFS application server; 2 GB+ / 2 cores for a dedicated PostgreSQL server.

Extremely large studies can scale horizontally: multiple BOFS instances behind a load balancer, a shared PostgreSQL with connection pooling, and a CDN for static files. Doing this is beyond the scope of the documentation. Most research studies do *not* reach this point — *a single-server setup is capable of handling hundreds of concurrent users*.

4.3 Setting Up the Server

This is not a complete server administration tutorial; it covers the BOFS-specific steps. Adapt commands to your distribution and hosting environment.

Before starting, you'll need: SSH access to a Linux server (from your department's IT group or a cloud provider), permission to run commands as administrator (`sudo`), and — for HTTPS — a domain name pointing at the server.

4.3.1 Install BOFS in a Virtual Environment

```
python3 -m venv bofs_production
source bofs_production/bin/activate
pip install --upgrade pip
pip install bride-of-frankensystem
```

4.3.2 Copy Your Experiment to the Server

Use `scp` or `rsync` (command-line tools that copy files to a server over SSH), or `git clone` from a private repository:

```
mkdir -p ~/experiments/my_study
scp -r /local/path/to/experiment/* user@server:~/experiments/my_study/
```

4.3.3 Verify the Installation

```
source bofs_production/bin/activate
BOFS --help
cd ~/experiments/my_study
BOFS run production.toml
```

`BOFS --help` confirms BOFS is installed and on the path. `BOFS run production.toml` starts the server; once it prints `Listening on http://...` the config has loaded successfully. Stop it with `Ctrl+C` — the next sections set it up to run as a managed service behind a reverse proxy.

4.4 Web Server Configuration

4.4.1 Run BOFS with Waitress

BOFS ships with a Waitress-based production server that handles concurrent requests. Start it in production mode by running BOFS **without** the `-d` flag:

```
BOFS run production.toml
```

This binds to `0.0.0.0` — meaning BOFS accepts connections from any network interface, not just the local machine — on the port specified in your config, so the project is reachable at `http://your-ip-address:<PORT>`. The connection is unencrypted at this point — anything participants submit travels in plaintext. For studies recruiting real participants, put a reverse proxy in front of BOFS so traffic is encrypted under HTTPS.

4.4.2 Run BOFS Under systemd

A systemd unit ensures BOFS starts on boot and restarts on failure. Create `/etc/systemd/system/bofs-experiment.service`:

```
[Unit]
Description=BOFS Experiment Server
After=network.target

[Service]
Type=simple
User=YOUR_USERNAME
WorkingDirectory=/path/to/your/experiment
Environment=PATH=/path/to/bofs_production/bin
ExecStart=/path/to/bofs_production/bin/python -m BOFS run production.toml
Restart=always
RestartSec=10

[Install]
WantedBy=multi-user.target
```

Enable, start, and inspect the service (`journalctl` streams the service's log in real time — useful for catching startup errors):

```
sudo systemctl enable bofs-experiment
sudo systemctl start bofs-experiment
sudo systemctl status bofs-experiment
sudo journalctl -u bofs-experiment -f
```

4.4.3 Reverse Proxy

A reverse proxy sits in front of BOFS, terminates TLS so traffic is encrypted as HTTPS, and forwards requests to BOFS over the local connection.

Caddy

Caddy provisions and renews TLS certificates automatically via Let's Encrypt — no separate certbot step.

Install Caddy (installation instructions) and put this in `/etc/caddy/Caddyfile`:

```
yourdomain.com, www.yourdomain.com {
    reverse_proxy 127.0.0.1:5000

    handle_path /static/* {
        root * /path/to/your/experiment/static
        file_server
        header Cache-Control "public, immutable, max-age=31536000"
    }

    header {
        X-Frame-Options "SAMEORIGIN"
        X-Content-Type-Options "nosniff"
    }
}
```

Reload Caddy to pick up the change:

```
sudo systemctl reload caddy
```

Caddy obtains a TLS certificate the first time someone hits the domain.

Nginx

Two pieces are involved: installing Nginx and obtaining a TLS certificate for your domain.

Install Nginx. Follow the official Nginx installation instructions. On Debian/Ubuntu:

```
sudo apt update
sudo apt install nginx
```

Obtain a TLS certificate. The Nginx config below references certificate and key files. The simplest route is Let's Encrypt's Certbot:

```
sudo apt install certbot python3-certbot-nginx
sudo certbot --nginx -d yourdomain.com -d www.yourdomain.com
```

Certbot can modify your Nginx config in place or just produce cert files (typically under `/etc/letsencrypt/live/yourdomain.com/`). For commercial certificates, follow your CA's installation guide.

Configure the site. Create `/etc/nginx/sites-available/bofs-experiment`:

```
server {
    listen 80;
    server_name yourdomain.com www.yourdomain.com;
    return 301 https://$server_name$request_uri;
}

server {
    listen 443 ssl http2;
    server_name yourdomain.com www.yourdomain.com;

    ssl_certificate /etc/letsencrypt/live/yourdomain.com/fullchain.pem;
    ssl_certificate_key /etc/letsencrypt/live/yourdomain.com/privkey.pem;
    ssl_protocols TLSv1.2 TLSv1.3;
    ssl_ciphers HIGH:!aNULL:!MD5;

    add_header X-Frame-Options "SAMEORIGIN" always;
    add_header X-Content-Type-Options "nosniff" always;

    location / {
        proxy_pass http://127.0.0.1:5000;
        proxy_set_header Host $http_host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;
    }

    location /static {
        alias /path/to/your/experiment/static;
        expires 1y;
        add_header Cache-Control "public, immutable";
    }
}
```

Enable the site:

```
sudo ln -s /etc/nginx/sites-available/bofs-experiment /etc/nginx/sites-enabled/
sudo nginx -t
sudo systemctl restart nginx
```

Note

Brotli-compressed Unity builds require HTTPS to decompress correctly in the browser. Serving them over plain HTTP will cause the game to fail to load.

4.5 Database Configuration

Set `SQLALCHEMY_DATABASE_URI` in your TOML config. BOFS supports anything SQLAlchemy supports; the common options are:

```
# SQLite — file-based, no separate service needed
SQLALCHEMY_DATABASE_URI = "sqlite:///study.db"

# PostgreSQL
SQLALCHEMY_DATABASE_URI = "postgresql://username:password@localhost/study_db"

# MySQL
SQLALCHEMY_DATABASE_URI = "mysql+pymysql://username:password@localhost/study_db"
```

For PostgreSQL, install the Python adapter into the BOFS virtual environment:

```
pip install psycopg2-binary
```

PostgreSQL installation and database creation are outside the scope of this guide — see your hosting provider's documentation or the official PostgreSQL documentation. For connection pooling, see the Flask-SQLAlchemy configuration guide.

See *Configuration Reference* for all database-related settings.

4.6 Securing the Admin Panel

The admin panel exposes participant data and includes destructive controls.

Set a long, random `ADMIN_PASSWORD` in your TOML config:

```
ADMIN_PASSWORD = "a-long-random-password-here"
```

To restrict `/admin` to a specific IP at the reverse-proxy layer:

Caddy

```
yourdomain.com {
    @admin_block {
        path /admin*
        not remote_ip 203.0.113.5/32
    }
    respond @admin_block 403
```

(continues on next page)

(continued from previous page)

```
    reverse_proxy 127.0.0.1:5000
}
```

Nginx

```
location /admin {
    allow 203.0.113.5/32;
    deny all;

    proxy_pass http://127.0.0.1:5000;
    proxy_set_header Host $http_host;
    proxy_set_header X-Real-IP $remote_addr;
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_set_header X-Forwarded-Proto $scheme;
}
```

4.7 Brute-Force Protection and Session Security

BOFS includes IP-based protection on the admin login, IP binding for sessions, and probe-URL / hostile-user-agent traps. All of these are on by default. See *Configuration Reference* for the full list of security settings.

4.7.1 Admin Login Protection

Failed admin logins are tracked per IP. After `BRUTE_FORCE_MAX_ATTEMPTS` failures (default: 5) within `BRUTE_FORCE_WINDOW_MINUTES` (default: 15 minutes), the IP is banned. Bans use a progressive schedule defined by `BRUTE_FORCE_BAN_SCHEDULE` (default: 1 min → 2 min → 5 min → 15 min → 1 hr → 6 hr → 1 day → 7 days), so the first ban is brief and repeat offenders escalate to multi-day bans.

4.7.2 Session IP Binding

Admin and participant sessions each bind to the IP that created them. See *Sessions and Participant State* for session lifecycle details.

- **Admin session**: when an admin logs in, the session records the request IP. Subsequent admin requests from a different IP are rejected and the session is cleared. Admins should not roam networks while logged in.
- **Participant session**: a participant's session is invalidated if a request comes from a different IP than the one that created the session. Disable with `SESSION_BIND_TO_IP_PARTICIPANT = false` for studies where participants may switch networks (cellular to Wi-Fi) mid-session.

4.7.3 Probe URL Traps and Hostile User Agents

Any request to a path in `BRUTE_FORCE_PROBE_URLS` (e.g., `/.env`, `/wp-admin`, `/.git`) immediately bans the source IP. If a custom blueprint serves a path that appears in the default list, edit the list in your TOML config.

Any request whose `User-Agent` contains a pattern from `BRUTE_FORCE_HOSTILE_UA_PATTERNS` (`sqlmap`, `nikto`, `nmap`, etc.) is also immediately banned. This is easily evaded — attackers can pass a different agent string — so it is defense in depth, not a primary control.

4.7.4 Auto-Trust for Known Admin IPs

Once an admin has logged in successfully from a given IP, that IP is added to a persistent allowlist (the `admin_trusted_ip` table) and exempted from future bans. To disable this, set `BRUTE_FORCE_AUTO_TRUST_ADMIN = false`.

4.7.5 BEHIND_REVERSE_PROXY Setting

If BOFS runs behind Caddy or Nginx, set `BEHIND_REVERSE_PROXY = true` in your TOML so BOFS reads the real client IP from `X-Forwarded-For` rather than seeing `127.0.0.1` for every request. Without this, every IP-based check treats the proxy itself as the client. The Caddyfile and Nginx configs above already pass the correct headers.

```
BEHIND_REVERSE_PROXY = true
```

4.7.6 Recovery from Self-Lockout

The auto-trust list covers the common case. If a fresh admin (no prior successful login on record from their IP) exhausts the attempt budget, three recovery paths are available:

1. Add their IP to `TRUSTED_IPS` in `config.toml` and restart BOFS.
2. Set `BRUTE_FORCE_PROTECTION = false` in `config.toml` and restart; log in successfully (which adds the IP to `admin_trusted_ip`); re-enable `BRUTE_FORCE_PROTECTION` and restart.
3. Edit the database directly: `DELETE FROM banned_ip WHERE ipAddress = '...';`

4.7.7 Limits of IP-Based Protection

This is application-layer protection calibrated for casual brute-force attackers. It does not stop a determined attacker rotating IPs through Tor or residential-proxy networks, and it can lock out legitimate users behind CGNAT (mobile carriers, university networks) when one bad actor on the same network trips a ban.

For DDoS protection and broader rate limiting, use proxy-layer controls. Caddy's caddy-ratelimit plugin can rate-limit by path:

```
# Caddyfile snippet — requires the caddy-ratelimit plugin
rate_limit {
    zone participant_create {
        match {
            path /consent /consent_nc /create_participant /create_participant_nc
        }
        key {client_ip}
        events 10
        window 1m
    }
}
```

4.8 Troubleshooting

Service won't start. Check the journal first: `sudo journalctl -u bofs-experiment --lines=50`. Common causes are TOML syntax errors, database connection failures, file-permission problems, and a port already in use.

SSL certificate expired. With Let's Encrypt and Certbot: `sudo certbot renew && sudo systemctl reload nginx`. With Caddy, renewal is automatic.

PostgreSQL connection refused. Confirm the service is running (`sudo systemctl status postgresql`) and check `/var/log/postgresql/postgresql-*-main.log`.

PostgreSQL "too many connections". Tune the pool in your TOML:

```
[SQLALCHEMY_ENGINE_OPTIONS]
pool_size = 5
max_overflow = 10
```

High CPU or memory. Check for runaway processes or query patterns first. If the workload genuinely justifies it, scale up the VM or add swap. For slow responses, add indexes on frequently queried columns and enable compression at the proxy layer.

4.9 After Deployment

Run a small pilot study before opening the project to real participants. Walk through the experiment yourself, verify rows appear in the database, confirm the completion code or redirect fires at the end page, check that external IDs are being captured, and confirm the logs are clean. For recruiting participants via crowdsourcing platforms, see *Recruiting via MTurk or Prolific*.

4.9.1 Recruiting via MTurk or Prolific

Most of the wiring is the same across platforms: capture the participant's ID from a URL parameter, optionally let them resume an interrupted session, and either show a completion code or redirect them back to the platform when they're done. The platform-specific parts are limited to URL formats and the completion-redirect target.

For server deployment (TLS, reverse proxy, systemd), see *Deploying to a Server*.

External IDs

An *external ID* is the participant identifier issued by the recruiting platform (MTurk Worker ID, Prolific PID, etc.). BOFS stores it on the `Participant.externalID` column (also accessible as `mTurkID`, an alias kept for backward compatibility) regardless of which platform you use. There are two ways to capture it.

URL parameter (automatic). When a participant arrives via a link that includes a recognised parameter, BOFS stores it on the participant row at consent without showing them anything. The recognised parameters are:

- `PROLIFIC_PID` — Prolific
- `external_id` — generic

A `?source=` URL parameter alongside lets you tag participants by recruitment channel (e.g. `?source=reddit`, `?source=email`). The value is free-form and lands on `Participant.source`, where expression code can branch on it (`show_if = "source == 'reddit'"`). When `PROLIFIC_PID` is present without an explicit `?source=`, BOFS infers `source="prolific"`.

Manual entry page. Add the `external_id` page to your `PAGE_LIST` to prompt the participant to type their ID:

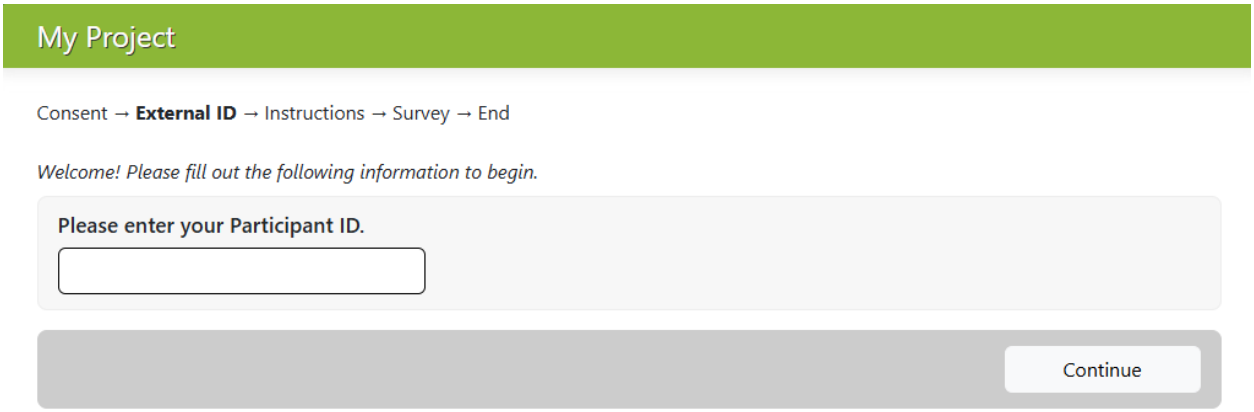
```
PAGE_LIST = [
    {name="Consent", path="consent"},
    {name="Participant ID", path="external_id"},
    {name="Study", path="questionnaire/main"},
    {name="End", path="end"}
]
```

With URL-parameter capture in place, this page is optional. It pre-fills from the URL parameter if one was present, so include it only as a fallback for participants who might arrive without one.

Customise the label and prompt shown on the manual entry page:

```
EXTERNAL_ID_LABEL = "Your Platform Participant ID"
EXTERNAL_ID_PROMPT = "Please enter the participant ID provided by the research␣
↪platform."
```

The defaults provide a MTurk-specific prompt, so you will want to override both when using Prolific or a generic platform. If you override *EXTERNAL_ID_PROMPT*, then *EXTERNAL_ID_LABEL* will not be used.



Session Management

Two settings control what happens when a participant returns with an external ID that has already been seen:

```
RETRIEVE_SESSIONS = true    # Resume incomplete sessions for known IDs
ALLOW_RETAKES = false       # Prevent the same ID from completing twice
```

With `RETRIEVE_SESSIONS = true`, BOFS checks each external ID against existing participants, loads the previous session if it is incomplete, resumes from the last completed page, and preserves the original condition assignment. `ALLOW_RETAKES = false` blocks a participant whose session is already marked finished from starting again. These two together are the standard configuration for crowdsourced studies: workers who closed their browser accidentally can return and finish instead of being lost to attrition, but repeat submissions are rejected.

For a deeper look at how session recovery works, see *Sessions and Participant State*.

Completion

There are three options for how a participant's run ends.

Generated completion code — a unique code per participant:

```
GENERATE_COMPLETION_CODE = true
COMPLETION_CODE_MESSAGE = "Please copy this completion code and paste it into the HIT␣
↪to receive payment:"
```

BOFS generates a UUID-derived code automatically. Use this for MTurk, where each worker pastes the code into the HIT form to claim payment. Because each code is unique per participant, you can verify that a submitted code came from someone who actually reached the end page.

Static completion code — the same code for everyone:

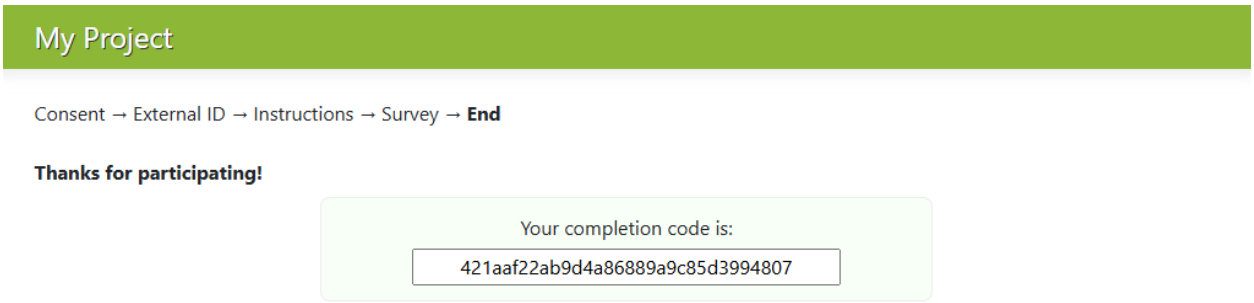
```
GENERATE_COMPLETION_CODE = false
STATIC_COMPLETION_CODE = "STUDY2024"
COMPLETION_CODE_MESSAGE = "Your completion code is: STUDY2024"
```

Redirect — skip the code entirely and send the participant back to the platform:

```
GENERATE_COMPLETION_CODE = false
OUTGOING_URL = "https://app.prolific.co/submissions/complete?cc=C1ABC123"
```

> **Note**
>
> `GENERATE_COMPLETION_CODE = true` and `OUTGOING_URL` are mutually exclusive — only one of them takes effect.

`OUTGOING_URL` sets one redirect target for the whole study. When different participants need different completion URLs — separate codes per recruitment source, or a distinct exit for screened-out participants — give each end page its own `outgoing_url` instead. See *Multiple end pages* in the page-flow guide; the `outgoing_url` string is rendered through Jinja with the `participant` in scope, so it can carry per-participant values.



MTurk

MTurk workers paste a completion code into the HIT form to claim payment, so the generated-code approach is the standard fit.

HIT URL. When creating the HIT in MTurk, set the URL to pass the Worker ID as a parameter:

```
https://yourdomain.com/consent?external_id=${mturk.workerId}
```

You can also leave the URL as `https://yourdomain.com/consent` and rely on the manual `external_id` page if you prefer the worker to confirm their ID.

Example TOML for MTurk:

```
TITLE = "Research Study"
SQLALCHEMY_DATABASE_URI = "sqlite:///mturk_study.db"
ADMIN_PASSWORD = "changeme"

EXTERNAL_ID_LABEL = "MTurk Worker ID"
EXTERNAL_ID_PROMPT = "Please enter your MTurk Worker ID exactly as it appears in your␣
↪dashboard."

GENERATE_COMPLETION_CODE = true
COMPLETION_CODE_MESSAGE = "Please copy this completion code and paste it into the␣
↪MTurk HIT to receive payment:"

RETRIEVE_SESSIONS = true
ALLOW_RETAKES = false
```

(continues on next page)

(continued from previous page)

```
PAGE_LIST = [
    {name="Consent", path="consent"},
    {name="Worker ID", path="external_id"},
    {name="Study", path="questionnaire/main"},
    {name="End", path="end"}
]
```

If you want to redirect workers to MTurk's external-submit endpoint instead of showing a code, set `OUTGOING_URL` to the MTurk submit URL and set `GENERATE_COMPLETION_CODE = false`.

Prolific

Prolific provides each study with a completion URL containing a completion code. Participants must reach that URL for the platform to credit their submission, so the redirect approach is the natural fit.

Study URL. Configure the Prolific study URL as:

```
https://yourdomain.com/consent?PROLIFIC_PID={{%PROLIFIC_PID%}}
```

BOFS picks up `PROLIFIC_PID` from the URL and stores it as the external ID.

Example TOML for Prolific:

```
TITLE = "Research Study"
SQLALCHEMY_DATABASE_URI = "sqlite:///prolific_study.db"
ADMIN_PASSWORD = "changeme"

EXTERNAL_ID_LABEL = "Prolific ID"
EXTERNAL_ID_PROMPT = "Your Prolific ID should be automatically detected. If not,␣
↪please enter it manually."

GENERATE_COMPLETION_CODE = false
OUTGOING_URL = "https://app.prolific.co/submissions/complete?cc=C1ABC123"

RETRIEVE_SESSIONS = true
ALLOW_RETAKES = false

PAGE_LIST = [
    {name="Consent", path="consent"},
    {name="Prolific ID", path="external_id"},
    {name="Study", path="questionnaire/main"},
    {name="End", path="end"}
]
```

Replace `C1ABC123` with your study's actual completion code from Prolific. If you want a custom completion page instead of an immediate redirect, leave `OUTGOING_URL` unset and override `templates/end.html` with Prolific-specific instructions.

Security

External IDs are personally identifiable. MTurk Worker IDs and Prolific PIDs can be linked back to the participant's account and to other studies they have participated in. They are stored in the database alongside the participant's responses. Check your IRB's data-retention guidance and consider whether your analysis pipeline needs to anonymise them before export.

The `?source=` parameter is self-reported and arrives unauthenticated in the URL. A participant recruited from one channel can claim to be from another by editing the URL. Treat `source` as a hint for filtering and reporting, not as a credential. Where the distinction matters — for example, per-source completion URLs that gate payment — gate on the presence of a platform-issued parameter like `PROLIFIC_PID`, not on the `source` string.

> **Warning**
>
> Never commit `ADMIN_PASSWORD` to public version control.

Troubleshooting

Debug mode surfaces detailed error messages, request flow, and database query logs, and adds a debug toolbar to the bottom of each page:

```
BOFS run config.toml -d
```

Admin panel. The `Participant` table shows stored external IDs; the `Progress` table shows session flow. Watching for duplicate IDs there is a fast way to catch configuration mistakes before they affect a live cohort.

Longitudinal studies

For multi-session studies (e.g., a day-0 and day-1 wave), the standard pattern is to capture `PROLIFIC_PID` on day 0 via `RETRIEVE_SESSIONS` and then look up the same participant on day 1 using `CONDITIONS_FROM_DB`, which reads the prior study's database to assign the returning participant to the same condition they were in before — keeping the manipulation consistent across waves.

See *Longitudinal Experiments* for a full walkthrough of this pattern, including example TOML for both waves and how `CONDITIONS_FROM_DB` and `CONDITIONS_FROM_CSV` work.

See also

- *Configuration Reference* — complete list of all TOML settings
- *Sessions and Participant State* — session lifecycle, `RETRIEVE_SESSIONS`, `ALLOW_RETAKES`
- *Longitudinal Experiments* — multi-wave studies with `CONDITIONS_FROM_DB`
- *Deploying to a Server* — production deployment

CHAPTER

FIVE

REFERENCE

Setting-by-setting and API-level reference for the framework.

5.1 Configuration Reference

Every option BOFS reads from a project's `.toml` file. For a guided introduction, see *Setting Up Your Page Flow*.

5.1.1 Required Settings

Variable	Type	Description
`SQLALCHEMY_DATABASE_U`	string	A URL-style address telling BOFS where your database file or server is. Use `sqlite:///filename.db` for SQLite or `postgresql://user:pass@host/db` for PostgreSQL.
`SQLALCHEMY_BINDS`	table	Optional. Additional databases the project can route questionnaires / custom tables into. Each entry maps a bind name to a connection string. A questionnaire or table opts into a bind by adding a `"database": "<bind-name>"` field to its JSON. See *The Database Layer* for the trade-offs (no cross-bind FK, separate backup discipline).
`TITLE`	string	Study title shown in browser tab and page headers.
`PORT`	integer	Port number for the web server (e.g., `5000`).
`ADMIN_PASSWORD`	string	Password for accessing the admin panel at `/admin`.
`PAGE_LIST`	list	Defines the sequence of pages participants see. See *PAGE_LIST Configuration* below.

5.1.2 Application Settings

Variable	Type	Default	Description
`APPLICATION_ROOT`	string	`""`	URL prefix if hosting at a subpath (e.g., `/study1`). Leave unset when hosting at the root.
`USE_BREADCRUMBS`	boolean	`true`	Show a breadcrumbs-style progress indicator to participants.
`USE_LOGO`	boolean	`true`	Display the BOFS logo in the page header.
`HEADER_COLOR`	string	*(unset)*	Background color of the title bar. Accepts a CSS hex color (e.g., `"#8CB737"`), a named color (e.g., `"navy"`), or `rgb()`/`rgba()`/`hsl()`/`hsla()` notation. When unset, the default green from the stylesheet is used.
`CHECK_FOR_UPDATES`	boolean	`true`	When `true`, BOFS checks PyPI for a newer version at startup and periodically while the admin panel is open. Set to `false` to disable (e.g., on air-gapped servers).
`WAITRESS_THREADS`	integer	`16`	How many simultaneous requests the production (non-debug) server can handle; Waitress is the server BOFS uses in that mode. See *Deploying to a Server*.
`SECRET_KEY`	string	*(auto-generated)*	Flask session signing key. BOFS generates a random key on first run and stores it in the `app_meta` database table; the generated key persists across restarts. A value in the config is migrated into the database on first run and then ignored. Documented here for completeness — do not set this manually.

5.1.3 Admin Panel Settings

Variable	Type	Default	Description
`USE_ADMIN`	boolean	`true`	Enable or disable the admin panel entirely.
`ADDITIONAL_ADMIN_PAGES`	list	`[]`	Custom admin pages contributed by blueprints. See *Monitoring and Exporting Data*.
`LOG_QUESTIONNAIRE_INT`	boolean	`false`	Log focus, blur, change, paste, drop, and visibility events for every input on every questionnaire (plus `paste_blocked` / `drop_blocked` events when paste is disabled). Text inputs additionally record per-field authenticity signals (keystrokes, backspaces, pastes, pasted character count, blocked pastes, drops, dropped character count, blocked drops, final length, total focus duration, time-to-first-keystroke). See *Monitoring and Exporting Data*.
`DISABLE_PASTE`	boolean	`false`	Block paste and drag-drop into all text inputs throughout the experiment (questionnaires and custom pages). Individual questions can opt in independently via the `disable_paste` question property even when this is `false`. See *Data Quality*.
`LOG_GRID_CLICKS`	boolean	*(deprecated)*	Deprecated alias for `LOG_QUESTIONNAIRE_INTERACTIONS`. If both keys are absent, `LOG_QUESTIONNAIRE_INTERACTIONS` is used. If only `LOG_GRID_CLICKS` is present, its value is copied to `LOG_QUESTIONNAIRE_INTERACTIONS` and a warning is printed. Rename to `LOG_QUESTIONNAIRE_INTERACTIONS` to silence the warning.
`EXPORT`	list	`[]`	Custom export definitions for blueprint-defined tables. Each entry is a dict describing a table, fields, and optional grouping. Populated automatically from loaded blueprints; can also be set in the project config.

5.1.4 Session and Participant Settings

Variable	Type	Default	Description
`RETRIEVE_SESSIONS`	boolean	`true`	If the external ID was used before, attempt to restore the participant's session and redirect to where they left off.
`ALLOW_RETAKES`	boolean	`false`	When `false`, prevents the same external ID from being used twice.
`ABANDONED_MINUTES`	integer	`5`	Minutes of inactivity before a participant is considered abandoned.
`COUNTS_INCLUDE_ABANDO`	boolean	`false`	Include abandoned participants when balancing condition assignment. Abandoned participants are not counted when balancing conditions by default.

5.1.5 Security Settings

These settings control IP-based brute-force protection on the admin login, binding sessions to the IP they were created on, how BOFS resolves the real client IP behind a reverse proxy, and request/cookie limits. See *Deploying to a Server* for deployment context.

Variable	Type	Default	Description
BRUTE_FORCE_PROTECTION	boolean	true	Master kill-switch. When false, IP banning, login-attempt tracking, and session IP binding are all bypassed. Useful for emergency recovery if an admin locks themselves out.
BRUTE_FORCE_AUTO_TRUST_AD	boolean	true	When true, a successful admin login adds the IP to a persistent allowlist (the admin_trusted_ip table) and exempts it from future bans. Acts as the primary self-service safeguard against admin self-lockout.
BRUTE_FORCE_MAX_ATTEMPTS	integer	5	Failed admin logins per IP per window before a ban is issued.
BRUTE_FORCE_WINDOW_MINUTE	integer	15	Sliding window for counting admin login failures.
BRUTE_FORCE_BAN_SCHEDULE	list of integers	[1, 2, 5, 15, 60, 360, 1440, 10080]	Progressive ban duration in minutes. The IP's prior ban count indexes into the list (1m, 2m, 5m, 15m, 1h, 6h, 1d, 7d). The final entry sticks for any further bans. Historical ban rows are kept so the count is accurate across time.
BRUTE_FORCE_PROBE_URLS	list of strings	*(curated list, see below)*	Paths that instantly ban any visiting IP — used to catch scanners hitting well-known attack targets like /.env or /wp-admin. Match is exact-or-prefix. If a custom blueprint legitimately serves any of these paths, remove that entry.
BRUTE_FORCE_HOSTILE_UA_PA	list of strings	["sqlmap", "nikto", "nmap", "dirbuster", "gobuster", "masscan", "WPScan", "acunetix", "nessus"]	Case-insensitive substring matches against the request's User-Agent. A match instant-bans the IP. This is defense-in-depth rather than a primary control, since scanners can mask their user-agent.
SESSION_BIND_TO_IP_PARTICIPA	boolean	true	When true, a participant session is invalidated if the IP it was created from differs from the current request's IP. Set to false for studies with mobile users who legitimately switch networks (cellular to wifi) mid-session, since each network change produces a new public IP. Admin sessions are always bound — there is no opt-out.
TRUSTED_IPS	list of strings	[]	Static allowlist that bypasses all IP-based protection. Combines with the runtime admin_trusted_ip table.
BEHIND_REVERSE_PROXY	boolean	false	Set to true if BOFS runs behind Caddy or nginx so the real client IP is read from X-Forwarded-For (via Werkzeug's ProxyFix). When false, X-Real-IP and X-Forwarded-For are ignored — they are spoofable when nothing trusted is in front of the app.
MAX_CONTENT_LENGTH	integer	8388608 (8 MB)	Maximum request body size in bytes.

The default `BRUTE_FORCE_PROBE_URLS` list:

```
BRUTE_FORCE_PROBE_URLS = [
    "/.env",
    "/.git",
    "/.aws",
    "/wp-admin",
    "/wp-login.php",
    "/wp-includes",
    "/wp-content",
    "/xmlrpc.php",
    "/phpmyadmin",
    "/phpMyAdmin",
    "/administrator",
    "/admin.php",
    "/server-status",
    "/actuator",
    "/vendor/phpunit",
    "/cgi-bin",
    "/.DS_Store",
    "/.htaccess",
    "/.svn",
]
```

Recovery from a self-imposed lockout (the auto-trust list takes care of the common case once an admin has logged in successfully even once from the IP they're using):

1. Add the IP to `TRUSTED_IPS` in `config.toml` and restart.
2. Set `BRUTE_FORCE_PROTECTION = false` in `config.toml` and restart.
3. Edit the database directly: `DELETE FROM banned_ip WHERE ipAddress = '...';` and optionally `DELETE FROM admin_trusted_ip WHERE ipAddress = '...';`.

5.1.6 External ID Settings (MTurk/Prolific)

These settings control the `/external_id` page for collecting participant IDs from recruitment platforms. See *Recruiting via MTurk or Prolific*.

Variable	Type	Default	Description
`EXTERNAL_ID_LABEL`	string	`"Mechanical Turk Worker ID"`	Label used to describe the external ID field.
`EXTERNAL_ID_PROMPT`	string	`"Please enter your MTurk Worker ID. You can find this on your MTurk dashboard."`	Instructions to fully override the external ID prompt.

5.1.7 Completion Settings

These settings control the `/end` page behavior. See *Recruiting via MTurk or Prolific*.

Variable	Type	Default	Description
`GENERATE_COMPLETION_CODE`	boolean	`true`	Generate a random completion code for each participant.
`STATIC_COMPLETION_COD`	string	*(unset)*	Use the same completion code for all participants.
`COMPLETION_CODE_MESSAGE`	string	`"Please copy and paste this code into the MTurk form:"`	Message displayed alongside the completion code.
`OUTGOING_URL`	string	*(unset)*	Redirect participants to this URL at study end instead of showing a completion code. Useful for Prolific redirects. Applies only to the default `end` page; for per-exit redirects, set `outgoing_url` on individual end entries instead (see *Multiple end pages*).

5.1.8 Experimental Conditions

Define experimental conditions for random assignment. See *Conditions and Branching*.

```
CONDITIONS = [
    {label="Control", enabled=true},
    {label="Treatment A", enabled=true},
    {label="Treatment B", enabled=false}
]
```

Property	Description
`label`	Human-readable name for the condition (shown in admin panel).
`enabled`	Whether to assign new participants to this condition. Set to `false` to stop assignment without removing the condition.

Participants are assigned to the condition with the fewest participants. Condition numbers start at 1. Participants without an assigned condition have condition 0.

Longitudinal Condition Lookup

For longitudinal studies where condition assignment must match a prior session, BOFS can look up conditions from an external source. See *Longitudinal Experiments*.

Variable	Type	Default	Description
`CONDITIONS_FROM_CSV`	string	*(unset)*	Path to a CSV file (relative to the project working directory) mapping participant IDs to condition numbers. The file must have at least two columns: the participant ID and the condition. Mutually exclusive with `CONDITIONS_FROM_DB`.
`CONDITIONS_FROM_DB`	string	*(unset)*	SQLAlchemy connection URI for an external database from which condition assignments are looked up by participant ID. Mutually exclusive with `CONDITIONS_FROM_CSV`.

5.1.9 PAGE_LIST Configuration

The `PAGE_LIST` defines the sequence of pages participants encounter. See *Setting Up Your Page Flow*.

Basic Structure

```
PAGE_LIST = [
    {name="Consent", path="consent"},
    {name="Survey", path="questionnaire/demographics"},
    {name="End", path="end"}
]
```

Page Entry Properties

Property	Description
`name`	Human-readable name shown in admin panel and progress tracking.
`path`	URL route that determines what content to display.
`show_if`	Optional. An expression; the page is shown only when it evaluates true. See *Expressions: Calculations and Conditional Display*.
`outgoing_url`	Optional, valid only on `end` / `end/<reason>` entries. Redirect the participant to this URL instead of rendering the end page. Rendered through Jinja with the `participant` in scope; takes precedence over the project-wide `OUTGOING_URL`. See *Multiple end pages*.

Required Pages

- **First page** must be one of: `consent`, `consent_nc`, `create_participant`, or `create_participant_nc`
- **Last page** must be `end`

Available Page Types

Path Format	Description
`consent`	Built-in consent form with condition assignment. See *Consent Forms*.
`consent_nc`	Consent form without condition assignment. See *Consent Forms*.
`create_participant`	Create participant with condition assignment (no consent form).
`create_participant_nc`	Create participant without condition assignment.
`external_id`	Collect external ID (MTurk Worker ID, Prolific ID, etc.).
`questionnaire/name`	Display questionnaire from `questionnaires/name.json`.
`instructions/name`	Show page from `templates/instructions/name.html` with a Continue button.
`simple/name`	Show page from `templates/simple/name.html` with manual navigation.
`end`	Built-in completion page with code or redirect.
`end/name`	An alternate completion page tagged with `name` (stamped on the participant as `end_reason`). See *Multiple end pages*.

Conditional Routing

Show different pages based on participant condition. See *Conditions and Branching*.

```
PAGE_LIST = [
    {name="Consent", path="consent"},
    {conditional_routing=[
        {condition=1, page_list=[
            {name="Control Task", path="instructions/control"}
        ]},
        {condition=2, page_list=[
            {name="Treatment Task", path="instructions/treatment"}
        ]}
    ]},
    {name="End", path="end"}
]
```

5.1.10 Database Configuration

SQLite

```
SQLALCHEMY_DATABASE_URI = "sqlite:///study.db"
```

PostgreSQL

```
SQLALCHEMY_DATABASE_URI = "postgresql://username:password@localhost:5432/database"
```

MySQL

```
SQLALCHEMY_DATABASE_URI = "mysql+pymysql://username:password@localhost/database"
```

5.1.11 Example Configurations

Minimal Configuration

```
SQLALCHEMY_DATABASE_URI = "sqlite:///study.db"
ADMIN_PASSWORD = "admin"
```

(continues on next page)

(continued from previous page)

```
PAGE_LIST = [
    {name="Consent", path="consent"},
    {name="Survey", path="questionnaire/survey"},
    {name="End", path="end"}
]
```

MTurk/Prolific Study

```
SQLALCHEMY_DATABASE_URI = "sqlite:///study.db"
TITLE = "Research Study"
ADMIN_PASSWORD = "secure_password"

EXTERNAL_ID_LABEL = "MTurk Worker ID"
EXTERNAL_ID_PROMPT = "Please enter your MTurk Worker ID."
RETRIEVE_SESSIONS = true
ALLOW_RETAKES = false
GENERATE_COMPLETION_CODE = true
COMPLETION_CODE_MESSAGE = "Please copy this code into MTurk:"

PAGE_LIST = [
    {name="External ID", path="external_id"},
    {name="Consent", path="consent"},
    {name="Demographics", path="questionnaire/demographics"},
    {name="Task", path="questionnaire/task"},
    {name="End", path="end"}
]
```

A/B Testing Study

```
SQLALCHEMY_DATABASE_URI = "sqlite:///study.db"
TITLE = "Decision Making Study"
ADMIN_PASSWORD = "secure_password"

CONDITIONS = [
    {label="Control", enabled=true},
    {label="Treatment", enabled=true}
]

PAGE_LIST = [
    {name="Consent", path="consent"},
    {name="Demographics", path="questionnaire/demographics"},
    {conditional_routing=[
        {condition=1, page_list=[
            {name="Control Instructions", path="instructions/control"}
        ]},
        {condition=2, page_list=[
            {name="Treatment Instructions", path="instructions/treatment"}
        ]}
    ]},
    {name="Main Task", path="questionnaire/task"},
    {name="End", path="end"}
]
```

5.2 Built-in Question Types

For a guided introduction to building questionnaires, see *Adding Survey Questions*.

On this page

The following attributes are common to every type of question.

- `id`: string - Your field's unique id.
  - **This must be completely unique within each questionnaire.**
  - This can be omitted for question types which contain `id` fields for each item in the question (e.g., radiogrid and checklist)
- `questiontype`: string - Defines the type of question/input field this is
- `instructions`: string - Appears directly above the field to indicate what the user should enter inside the field.
- `title`: string - Add text above the question, outside the question's box.

Note: Many of the attributes that accept strings support HTML, such as `instructions` and `title`. However, JSON does not support line breaks, so any HTML needs to appear on one line.

Currently, the following types of input are supported:

Choice questions:

- `radiolist` - Select one option out of a list

- `checklist` - Select multiple options out of a list
- `drop_down` - Select one option from a drop-down list
- `radiogrid` - Display a collection of items in a grid. One row per item, with responses in a likert scale where the headers are shown above.
- `image_select` - Select one option from a set of images

Text and numeric entry:

- `field` - Simple single-line text entry
- `num_field` - Input a single number
- `multi_field` - Multi-line text entry
- `slider` - Drag the slider to a numeric value, with optional labels on the left and right.

Media and display:

- `textview` - Display plain text (HTML syntax is supported)
- `video` - Embed an HTML5 video, optionally requiring the participant to watch it before continuing
- `audio` - Embed an HTML5 audio clip, optionally requiring the participant to listen to it before continuing
- `image_click` - Click on an image to record one or more (x, y) positions, measured in the image's original (unscaled) pixels

Layout:

- `group` - Render a header followed by several sub-questions of any other type, optionally laid out side-by-side

5.2.1 Choice questions

radiolist

```
questiontype == 'radiolist'
```

Properties

- `id`: Unique id for checklist (required, string)
- `instructions`: Text needed to describe what slider input represents (optional, string)
- `required`: whether or not this input is required to submit form (optional, boolean: `true` or `false`, default is `false`)
- `required_selection`: If specified, force the user to select the specified value before the form can be submitted (optional, string).
- `shuffle`: Whether or not the possible response labels should be shuffled (optional, boolean: `true` or `false`, default is `false`)
- `horizontal`: Should the options be listed vertically (default) or horizontally? (optional, boolean: `true` or `false`, default is `true`)

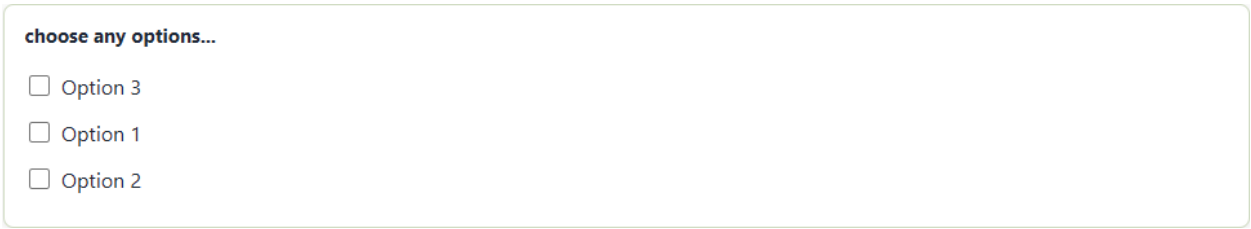
- `labels`: A list. One entry per each radio button. (required, list of strings)
- `other_enabled`: Show an "other" option as one of the options in the list. (optional, boolean: `true` or `false`, default is `false`)
- `other_text_prompt`: Specify the text to indicate what the "other" option means (optional, string).
- `other_input_width`: How wide the input field for the "other" option should be (optional, integer).
- `other_input_hides`: Should the input field for the "other" hide if not selected (optional, boolean, default `false`)?
- `disable_paste`: block paste and drag-drop into the "other" text input so it has to be typed (optional, boolean, default `false`; only applies when `other_enabled` is `true`).

Example

```
{
    "questiontype":"radiolist",
    "instructions":"Do you eat meat?",
    "id":"radiolist_1",
    "horizontal": false,
    "required": true,
    "labels":[
        "Always",
        "Sometimes",
        "Never"
    ]
}
```

checklist

`questiontype == 'checklist'`



Properties

- `instructions`: text needed to describe what slider input represents (optional, string)
- `shuffle`: should the order of the responses be shuffled? (optional, boolean: `true` or `false`, default is `false`)
- `horizontal`: should be options be listed vertically? (optional, boolean: `true` or `false`, default is `true`)
- `disable_paste`: block paste and drag-drop into every free-text (`text_entry`) input in this checklist so answers have to be typed (optional, boolean, default `false`)
- `questions`: one for each checkbox, a list of dictionaries, each with the following keys.
  - `id`: Must be unique within the questionnaire (required, integer).
  - `text`: The label for the option (required, string).
  - `text_entry`: Are users allowed to enter custom text to be associated with this checkbox (optional, boolean, default `false`)?

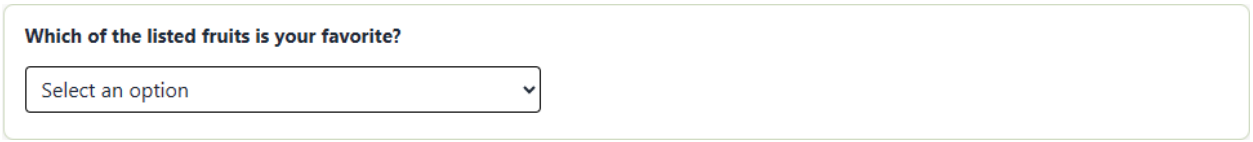
  - `text_entry_hides`: Does the text input area hide if the option is not selected (optional, boolean, default `false`)?
  - `text_entry_width`: How wide the input field for the text entry should be (optional, integer).

Example

```
{
    "questiontype":"checklist",
    "instructions":"choose any options...",
    "shuffle":true,
    "horizontal": false,
    "questions":[
        {
            "id":"cl_1",
            "text":"Option 1"
        },
        {
            "id":"cl_2",
            "text":"Option 2"
        },
        {
            "id":"cl_3",
            "text":"Option 3"
        }
    ]
}
```

drop_down

`questiontype == 'drop_down'`



Properties

- `id`: unique id for drop down menu (required, string)
- `instructions`: text to describe what the selection is for (optional, string)
- `required`: whether or not this input is required to submit form (optional, boolean: `true` or `false`, default is `false`)
- `items`: list of strings to describe possible selections in drop down menu (list of strings)
- `width`: width of the drop down (optional, integer, default `400`)

Example

```
{
    "questiontype": "drop_down",
    "id": "favorite_fruit",
    "instructions": "Which of the listed fruits is your favorite?",
    "items": [
        "apples", "oranges", "watermelon"
```

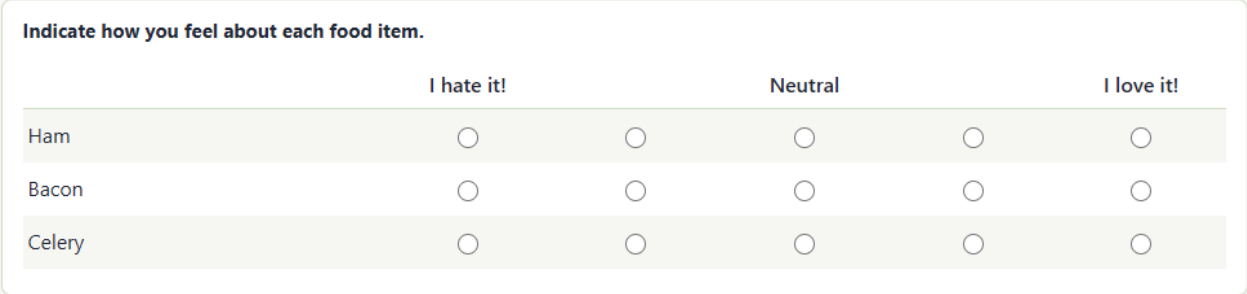
(continues on next page)

(continued from previous page)

```
    ]
}
```

radiogrid

`questiontype == 'radiogrid'`



- contains one or more horizontal rows of radio buttons.
- This input supports n-columns, and allows the researcher to provide a column header for each column.

Properties

- `instructions`: any text that is needed directly above the radiogrid (optional, string)
- `required`: whether or not responses to this radio grid are required to submit form (optional, boolean: `true` or `false`, default is `false`)
- `shuffle`: should the question order be shuffled? Randomising row order controls for item-order effects. (optional, boolean: `true` or `false`, default `false`)
- `labels`: list of strings that represent column headers (required, list of strings)
- `questions`: list of dictionaries that describe each individual question (required)
  - `id`: unique id of the row of radio buttons (string)
  - `text`: question text (string)
- `na_column`: add an extra column at the right of the grid for an "N/A" option. Selecting it satisfies `required` but stores the row's value as `NULL`. (optional, boolean, default `false`)
- `na_label`: header text for the N/A column. (optional, string, default `"N/A"`)
- `store_labels`: store the chosen column's label string (e.g. `"Strongly agree"`) instead of its 1-based index. When set, `participant_calculations` entries cannot reference this grid's rows — label text cannot be used in arithmetic, so BOFS reports an error at startup. (optional, boolean, default `false`)

Note

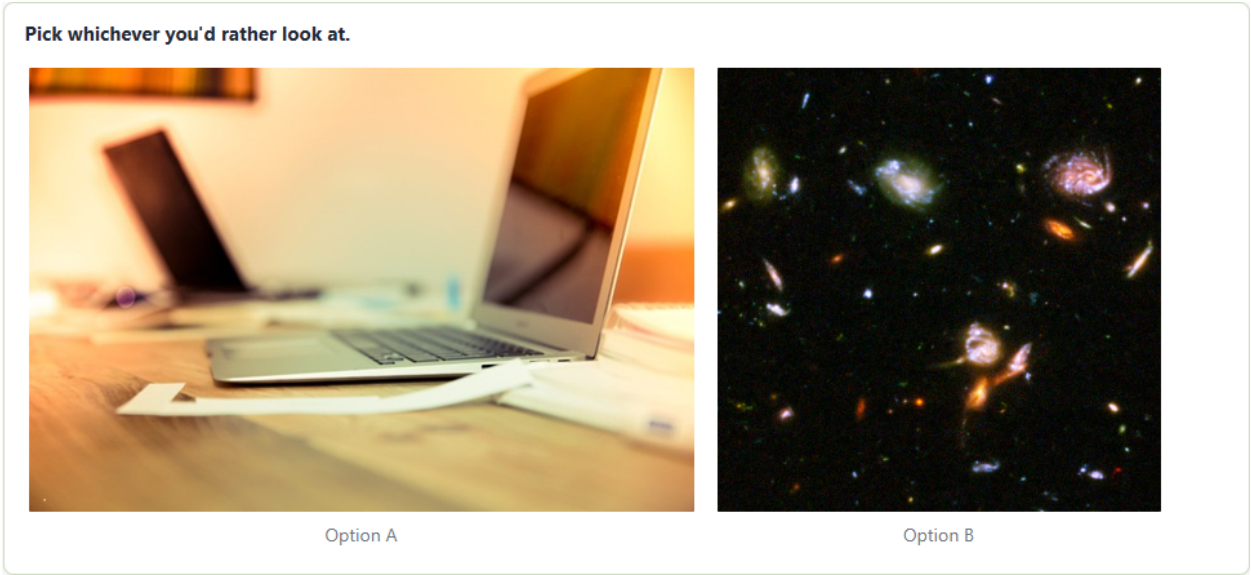
Radiogrid row columns are nullable so the `na_column` option can record `NULL`. Tables created before this option existed have non-null columns; to use `na_column` on a previously deployed questionnaire, drop or alter the existing table so it can be recreated with the new schema.

Example

```
{
    "questiontype": "radiogrid",
    "instructions": "Indicate how you feel about each food item.",
    "shuffle": true,
    "labels": [
        "I hate it!",
        "",
        "Neutral",
        "",
        "I love it!"
    ],
    "questions": [
        {
            "id": "q_1",
            "text": "Ham"
        },
        {
            "id": "q_2",
            "text": "Bacon"
        },
        {
            "id": "q_3",
            "text": "Celery"
        }
    ]
}
```

image_select

`questiontype == 'image_select'`



Select one option out of a set of images. Each image becomes a clickable thumbnail; the participant's selection is stored as the chosen image's `value`. Only one image can be selected at a time (radio behaviour).

Properties

- `id`: unique id for this question (required, string)
- `instructions`: text shown above the image grid (optional, string)
- `required`: whether a selection is required to submit the form (optional, boolean: `true` or `false`, default is `false`)
- `shuffle`: whether to shuffle the image order on each render (optional, boolean: `true` or `false`, default is `false`)
- `horizontal`: when `true` (the default), images wrap in a centered row; when `false`, they stack in a single centered column (optional, boolean: `true` or `false`, default is `true`)
- `width`: explicit width (in pixels) applied to every image (optional, integer). Ignored when `auto_resize` is `true`.
- `auto_resize`: resize the images in the browser so they all display at matching sizes, based on the smallest image in the group. With `horizontal: false` all widths are matched to the smallest image's width; otherwise all heights are matched to the smallest image's height (optional, boolean: `true` or `false`, default is `false`).
- `images`: list of image entries (required). Each entry is an object:
  - `src`: URL of the image (required, string), e.g. `/static/option_a.png`. At startup, BOFS checks that each locally-served `src` resolves to an existing file and reports a setup warning when one is missing. Remote URLs and data URIs are not checked.
  - `value`: value stored in the database when this image is selected (required, string / integer / float). When every entry's `value` is an integer (or float), the database column is created with that numeric type automatically.
  - `label`: short caption shown beneath the image, also used as `alt` text fallback (optional, string)
  - `alt`: explicit `alt` text for the `<img>` tag. Use this when the screen-reader description should differ from the visible caption, or set it to `""` for a purely decorative image (optional, string)

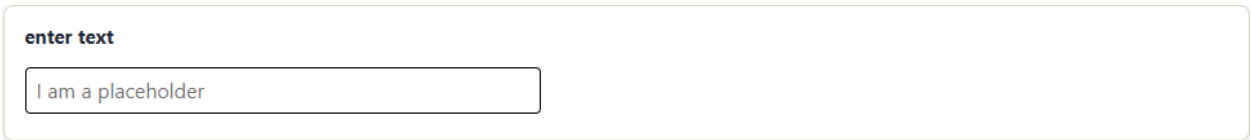
Example

```
{
    "questiontype": "image_select",
    "id": "favorite_picture",
    "instructions": "Pick whichever you'd rather look at.",
    "auto_resize": true,
    "required": true,
    "images": [
        {"src": "/static/option_a.jpg", "value": 1, "label": "Option A"},
        {"src": "/static/option_b.jpg", "value": 2, "label": "Option B"}
    ]
}
```

5.2.2 Text and numeric entry

field

`questiontype == 'field'`



- Standard single-line text entry field.

Properties

- `id`: unique id for text field (required, string)

- `instructions`: text needed to describe what field input should be (optional, string)
- `required`: whether or not this input is required to submit form (optional, boolean: `true` or `false`, default is `false`)
- `placeholder`: example text to show in field by default (optional, string)
- `width`: width of the field (optional, integer, default `400`)
- `disable_paste`: block paste and drag-drop into this field so the answer has to be typed (optional, boolean, default `false`)

Example

```
{
    "questiontype": "field",
    "instructions": "enter text",
    "placeholder": "I am a placeholder",
    "id": "input_1"
}
```

num_field

`questiontype == 'num_field'`

- Numeric text entry field.

Properties

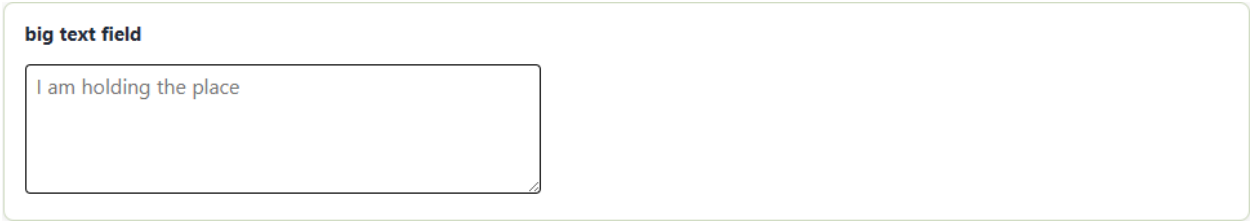
- `id`: unique id for number field (required, string)
- `instructions`: text needed to describe what field input should be (optional, string)
- `required`: whether or not this input is required to submit form (optional, boolean: `true` or `false`, default is `false`)
- `min`: minimum range for input (optional, integer)
- `max`: maximum range for input (optional, integer)
- `width`: width of the field (optional, integer, default `400`)
- `disable_paste`: block paste and drag-drop into this field so the answer has to be typed (optional, boolean, default `false`)

Example

```
{
    "questiontype": "num_field",
    "datatype": "integer",
    "instructions": "enter a number",
    "id": "input_1"
}
```

multi_field

`questiontype == 'multi_field'`



- Multi-line text field.

Properties

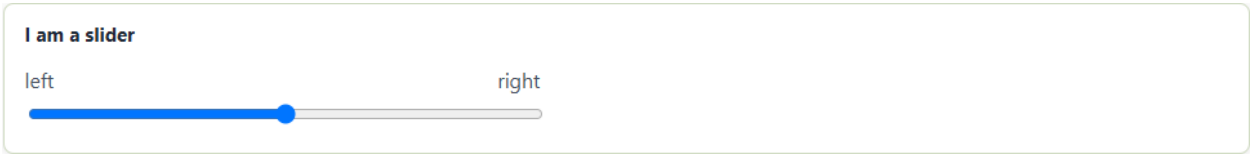
- `id`: unique id for number field (required, string)
- `instructions`: text needed to describe what field input should be (optional, string)
- `required`: whether or not this input is required to submit form (optional, boolean: `true` or `false`, default is `false`)
- `placeholder`: example text to show in field by default (optional, string)
- `height`: height of multifield (optional, integer, default `80`)
- `width`: width of the field (optional, integer, default `400`)
- `disable_paste`: block paste and drag-drop into this field so the answer has to be typed (optional, boolean, default `false`)

Example

```
{
    "questiontype": "multi_field",
    "id": "big",
    "placeholder": "I am holding the place",
    "instructions": "big text field",
    "height": 100
}
```

slider

`questiontype == 'slider'`



Properties

- `id`: unique id for slider (string)
- `instructions`: text needed to describe what slider input represents (optional, string)
- `left`: text for left label (optional, string)
- `right`: text for right label (optional, string)
- `tick_count`: number of ticks represented by the slider (required, integer)

- `width`: width of drop down (optional, integer, default `400`)

Example

```
{
    "questiontype": "slider",
    "instructions": "I am a slider",
    "id": "slider_1",
    "left": "left",
    "right": "right",
    "tick_count": 5
}
```

5.2.3 Media and display

textview

`questiontype == 'textview'`

Some header

These are some instructions which will appear wherever you place this question.

Properties

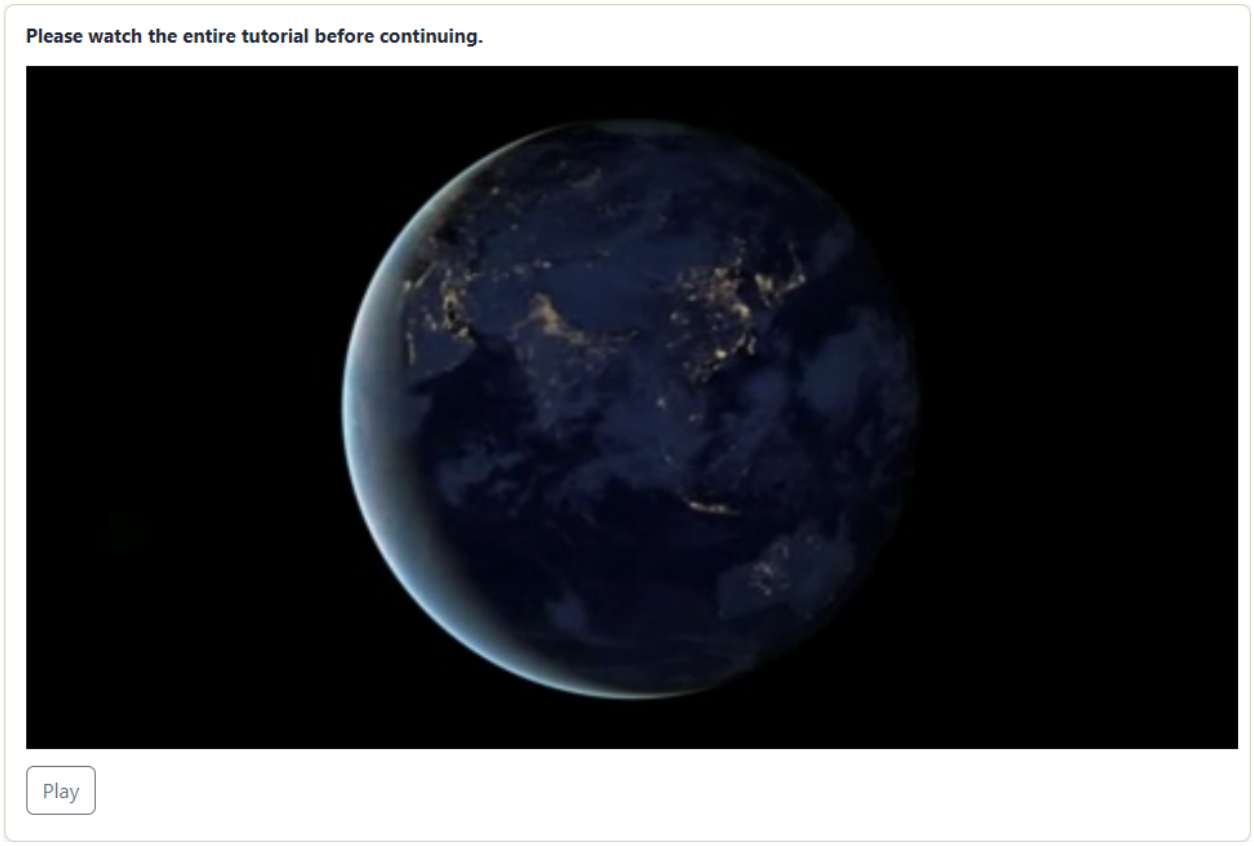
- `instructions`: title for block of text (optional, string)
- `text`: block of text to be displayed (optional, string)

Example

```
{
    "questiontype": "textview",
    "instructions": "Some header",
    "text": "These are some instructions which will appear wherever you place this␣
↪question."
}
```

video

`questiontype == 'video'`



Embeds an HTML5 `<video>` element from any URL. Can optionally require the participant to watch the video to completion before the *Continue* button is enabled, and (when `id` is set) records viewing telemetry to the database.

How "force watch" works

When `force_watch` is `true`, the *Continue* button is disabled and an explanatory notice appears beside it. A watched-time accumulator counts only forward playback while the tab is visible — forward seeks, backward seeks, playback rate changes, and background-tab playback do not count. The button unlocks once the accumulator reaches `completion_threshold` of the video's duration. If multiple videos on the same page set `force_watch`, the button is gated until *all* of them are satisfied.

When `minimal_controls` is `false` (the default), the native video controls are shown and a "snap-back" guard prevents the participant from scrubbing past the supposed playhead position. When `minimal_controls` is `true`, the native controls are hidden entirely and only a custom *Play/Pause* button is rendered, so seeking is impossible by construction.

Properties

- `src`: URL of the video file (required, string)
- `id`: when set, three columns are written to the questionnaire table (optional, string):
  - `{id}_started` — epoch seconds when the participant first pressed play, or `0` if they never started it
  - `{id}_ended` — epoch seconds at the last observed activity (play / timeupdate / pause / ended)
  - `{id}_watched` — accumulated forward play time, in seconds
- `width`, `height`: pixel dimensions for the video element (optional, integer)
- `autoplay`: start playing as soon as the page loads (optional, boolean, default `false`)
- `force_watch`: disable the *Continue* button until the participant has watched `completion_threshold` of the video (optional, boolean, default `false`)

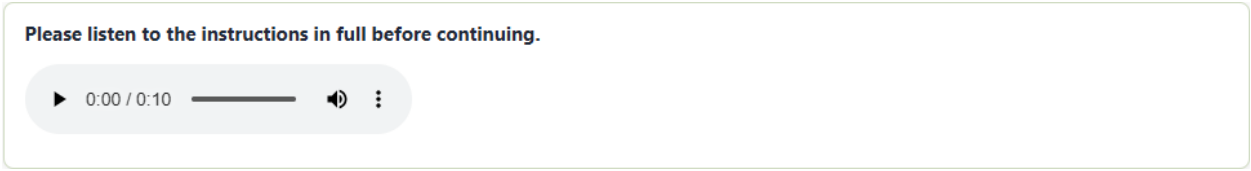
- `completion_threshold`: fraction of the video's duration that counts as "watched" (optional, float between 0 and 1, default `0.98`)
- `minimal_controls`: hide the native player controls and show only a custom Play/Pause button (optional, boolean, default `false`)

Example

```
{
    "questiontype": "video",
    "id": "tutorial_video",
    "instructions": "Please watch the entire tutorial before continuing.",
    "src": "https://example.com/static/tutorial.webm",
    "width": 940,
    "height": 530,
    "force_watch": true,
    "completion_threshold": 0.95,
    "minimal_controls": true
}
```

audio

`questiontype == 'audio'`



Embeds an HTML5 `<audio>` element from any URL with native player controls. Can optionally require the participant to listen to the clip in full before the *Continue* button is enabled, and (when `id` is set) records listening telemetry to the database.

When `force_listen` is on, a snap-back guard prevents the participant from scrubbing past the supposed playhead position, the playback rate is pinned to 1.0, and the clip pauses automatically when the tab loses focus. The *Continue* button is gated by the same shared notice used by `video` — if a page contains both pending audio and video, the notice reads "Please play all media before continuing."

Audio always shows native controls; there is no `minimal_controls` option, because the scrubber doubles as the participant's only progress indicator.

Properties

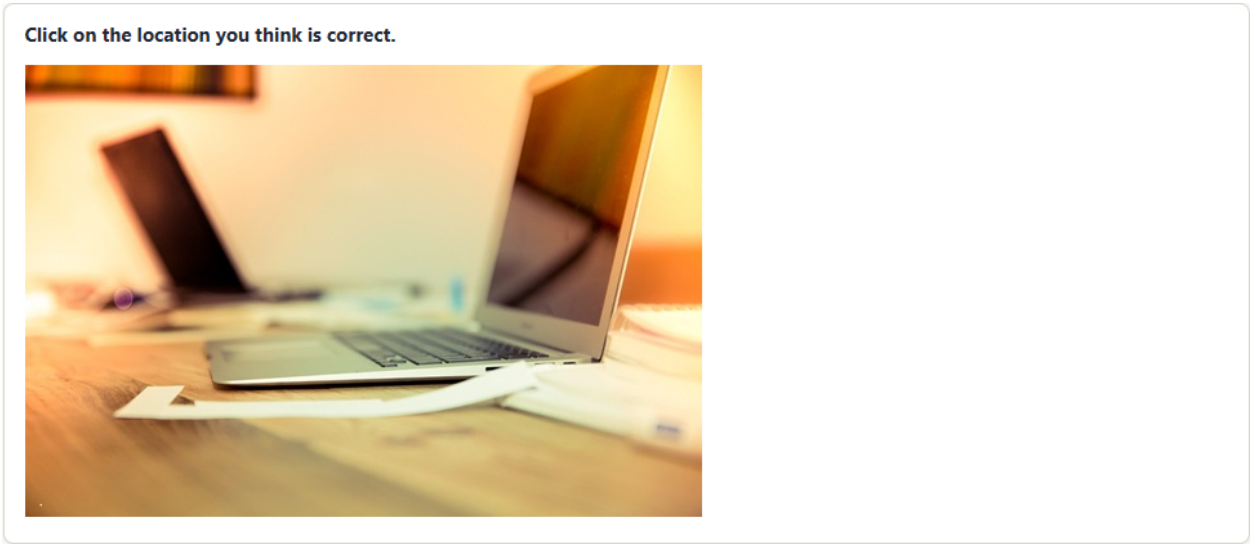
- `src`: URL of the audio file (required, string)
- `id`: when set, three columns are written to the questionnaire table (optional, string):
  - `{id}_started` — epoch seconds when the participant first pressed play, or `0` if they never started it
  - `{id}_ended` — epoch seconds at the last observed activity (play / timeupdate / pause / ended)
  - `{id}_listened` — accumulated forward play time, in seconds
- `autoplay`: start playing as soon as the page loads (optional, boolean, default `false`)
- `force_listen`: disable the *Continue* button until the participant has listened to `completion_threshold` of the clip (optional, boolean, default `false`)
- `completion_threshold`: fraction of the clip's duration that counts as "listened" (optional, float between 0 and 1, default `0.98`)

Example

```
{
    "questiontype": "audio",
    "id": "instructions_clip",
    "instructions": "Please listen to the instructions in full before continuing.",
    "src": "https://example.com/static/instructions.ogg",
    "force_listen": true,
    "completion_threshold": 0.95
}
```

image_click

`questiontype == 'image_click'`



Displays an image and stores the natural-image pixel coordinates of one or more clicks. Each click leaves a crosshair marker at the cursor position.

Coordinates are reported in the source image's *natural* pixel space (origin top-left, x to the right, y downward), independent of how the browser sized the image. A click on a 1000×800 image always lands within `(0, 0)-(1000, 800)` even when the image was scaled to fit the viewport.

Properties

- `id`: unique id for this question (required, string)
- `src`: URL of the image to display (required, string), e.g. `/static/map.png`
- `instructions`: text shown above the image (optional, string)
- `required`: when `true`, the *Continue* button is disabled until the participant has placed at least one click (optional, boolean, default `false`)
- `max_clicks`: maximum number of clicks accepted (optional, integer, default `1`).
  - `1` (default): each click moves the single marker to the new position.
  - any integer `> 1`: up to that many markers can be placed; once full, a new click drops the oldest marker.
  - `0`: unlimited clicks.

- `width`: maximum displayed width of the image, in pixels (optional, positive integer). The image is scaled down to fit while preserving aspect ratio. Coordinates remain in natural pixels regardless.
- `marker_color`: CSS colour of the crosshair (optional, string, default `"#ff0000"`)
- `marker_size`: pixel size of the crosshair (optional, integer, default `14`)

Storage

The database schema depends on `max_clicks`:

- When `max_clicks` is `1` (the default), two `FLOAT` columns are created — `{id}_x` and `{id}_y` — holding the natural-pixel x and y of the click.
- Otherwise, one `TEXT` column `{id}` is created, holding a JSON array of `{"x": ..., "y": ...}` objects in click order, e.g. `[{"x": 123.45, "y": 67.89}, {"x": 200.0, "y": 150.5}]`.

If the participant submits without clicking and `required` is `false`, the corresponding columns receive their default zero / empty value.

Example (single-click)

```
{
    "questiontype": "image_click",
    "id": "target_location",
    "instructions": "Click on the location you think is correct.",
    "src": "/static/map.png",
    "required": true
}
```

Example (multi-click)

```
{
    "questiontype": "image_click",
    "id": "all_errors",
    "instructions": "Click on every part of the diagram that looks wrong.",
    "src": "/static/diagram.png",
    "max_clicks": 0,
    "marker_color": "#0066ff"
}
```

5.2.4 Layout

group

```
questiontype == 'group'
```

Renders a header followed by a list of sub-questions of any other type.These questions are all visually grouped within the same card. The group itself does not create a database column — only its sub-questions do — so the group 's `id` is purely structural. Groups cannot contain other groups.

Sub-question `id`s share the same namespace as top-level question ids, so each one must be unique within the questionnaire as a whole, not just within the group.

The `show_sub_labels` property controls how the group reads:

- When `false` (the default), the per-sub-question `instructions` labels are hidden and the group reads as a single compound question — for example, one "About you" header above height and weight inputs.
- When `true`, each sub-question keeps its own `instructions` label and the group reads as a visually-grouped cluster of separately-labelled fields.

A group has two layered headings:

- `instructions`: the bold heading rendered at the top, like the `instructions` field on every other question type.
- `text`: an optional non-bold sub-heading rendered inside the group's fieldset, between the bold heading and the sub-questions.

Properties

- `id`: optional structural id for the group's HTML wrapper; not stored as a database column (optional, string). Sub-questions carry their own IDs, which is what populates the database.
- `instructions`: bold heading shown above the group (optional, string). Same as the common `instructions` attribute on other question types.
- `text`: optional non-bold sub-heading shown inside the group, between the bold `instructions` and the sub-questions (optional, string)
- `questions`: list of sub-question objects of any non-`group` type (required, list)
- `show_sub_labels`: whether each sub-question keeps its own `instructions` label (optional, boolean: `true` or `false`, default `false`)
- `horizontal`: lay the sub-questions out side-by-side instead of stacked vertically (optional, boolean: `true` or `false`, default `false`)
- `show_if`: expression that conditionally shows or hides the entire group (optional, string)

Example

```
{
    "questiontype": "group",
    "id": "demographics",
```

(continues on next page)

(continued from previous page)

```
    "instructions": "About you",
    "text": "These details help us interpret your responses.",
    "show_sub_labels": true,
    "questions": [
        {
            "questiontype": "field",
            "id": "first_name",
            "instructions": "First name"
        },
        {
            "questiontype": "num_field",
            "id": "age",
            "instructions": "Age (years)"
        },
        {
            "questiontype": "slider",
            "id": "experience",
            "instructions": "Experience level",
            "tick_count": 5
        }
    ]
}
```

5.3 Questionnaire Properties

A questionnaire is a JSON file in your project's `questionnaires/` directory. Each file produces one page of questions and one database table. For a guided introduction to writing questionnaires, see *Adding Survey Questions*.

5.3.1 Top-level properties

Property	Required	Description
`questions`	Yes	Array of question objects. Must be present; may not be empty.
`title`	No	Name shown in the admin panel. Not shown to participants.
`reference`	No	Citation text for a validated scale (admin panel only).
`doi`	No	DOI string for the instrument (admin panel only).
`instructions`	No	Text rendered above the questions. Accepts HTML. Also participates in `{{ }}` substitution (see below).
`code`	No	A string of JavaScript inserted into the page when it loads in the participant's browser, after the question inputs. Useful for task-specific logic. The value of `code` is never HTML-sanitised or `{{ }}`-substituted — literal `{{ }}` in third-party templates is not affected.
`participant_calculations`	No	Object mapping calculated column names to expression strings. See *participant_calculations* below.
`database`	No	Name of a `SQLALCHEMY_BINDS` entry from `config.toml`. Saves this questionnaire's responses to that database instead of the project's main one. See *The Database Layer* for the trade-offs.

BOFS ignores unrecognised top-level keys, so you can add metadata fields for your own reference.

5.3.2 `{{ }}` substitution

Any user-facing string in a questionnaire can contain `{{ expression }}` placeholders. When the questionnaire is rendered for a participant, each placeholder is replaced with the result of evaluating the inner text as an expression (the same syntax used by `show_if` and `participant_calculations` — see *Expressions: Calculations and Conditional Display*).

Fields that participate in substitution

Substitution applies to every string-valued field in the JSON that is not on the skip list below, including:

- Top-level: `title`, `instructions`
- Per-question: `title`, `instructions`, `left`, `right`, `prompt`, `text`
- List elements: items in `labels`, `items`, and the `text` field of `q_text` entries

Fields that are skipped (values pass through verbatim):

- `id`, `questiontype` — structural identifiers
- `show_if`, `participant_calculations`, `_show_if_ast` — already use the expression DSL directly
- `code` — JS slot; literal `{{ }}` is common in third-party templates
- `src` — asset URL

Substitution rules

- Each `{{ ... }}` is evaluated once against the current participant's stored data. The substituted value is HTML-escaped, so free-text answers or table values cannot inject markup.
- An expression that returns `None`, fails to parse, or references data the participant has not yet produced substitutes as an empty string. The surrounding text still renders.
- Substitution is single-pass. A substituted value that itself looks like `{{ x }}` is not re-scanned.
- The `/admin/preview_questionnaire/...` route renders the raw JSON without substitution so you can inspect the source text.

Because `{{ }}` is now reserved inside questionnaire JSON, a questionnaire that previously contained literal `{{` in participant-facing copy will render differently. To include a literal `{{`, write `{{ '{{' }}`.

Example

Listing 1: A debrief questionnaire that pulls scores from a JSONTable

```
{
    "title": "Your results",
    "instructions": "<p>You scored {{ tables.scores.total_score }} points across all␣
↪rounds (best: {{ tables.scores.high_score }}).</p>",
    "questions": [
        {
            "id": "thoughts",
            "questiontype": "multi_field",
            "title": "Reflection",
            "instructions": "Round 1: {{ tables.scores.round_score[1] }}. Round 2: {{␣
↪tables.scores.round_score[2] }}. What strategy did you use?"
        }
    ]
}
```

5.3.3 participant_calculations

The participant_calculations block computes derived values from a participant's responses. Each key in the object becomes an additional column in the questionnaire's database table and CSV export; the value is an expression string evaluated when data is exported or read back via the participant API.

Listing 2: Computing a scale score and a reverse-scored item

```
{
    "questions": [
        {"id": "ext_1", "questiontype": "slider", "instructions": "I am outgoing",
         "left": "Strongly disagree", "right": "Strongly agree", "tick_count": 7},
        {"id": "ext_2", "questiontype": "slider", "instructions": "I am reserved␣
↪(reverse scored)",
         "left": "Strongly disagree", "right": "Strongly agree", "tick_count": 7},
        {"id": "ext_3", "questiontype": "slider", "instructions": "I am full of␣
↪energy",
         "left": "Strongly disagree", "right": "Strongly agree", "tick_count": 7}
    ],
    "participant_calculations": {
        "extraversion": "mean([ext_1, 8 - ext_2, ext_3])"
    }
}
```

Naming rules for calculated fields:

- Must start with a letter or underscore, followed by letters, digits, or underscores (the same rule as question field IDs).
- Must not be a Python keyword.
- Must not collide with the reserved BOFS columns participantID, timeStarted, timeEnded, tag, or duration.
- Must not collide with the reserved expression names condition or tables.

For expression syntax, see *Expressions: Calculations and Conditional Display*.

5.3.4 Question-level show_if

A question can declare a show_if predicate that controls whether it is visible on the page.

```
{
    "questions": [
        {"id": "age", "questiontype": "num_field", "instructions": "How old are you?"}
↪,
        {"id": "guardian", "questiontype": "field", "show_if": "age < 18",
         "instructions": "Who is your parent or guardian?"}
    ]
}
```

The predicate is evaluated live in the browser as the participant fills out the form. When the predicate is false, the question is hidden and its inputs are excluded from submission — including required fields, which will not block the participant from continuing.

At submission time, a hidden question's database column receives its default value: an empty string for text fields, 0 for numeric fields. No submitted value is stored.

BOFS parses every `show_if` predicate at startup. A parse error or a reference to an unrecognised field ID is reported immediately with the questionnaire filename and question number.

For expression syntax, see *Expressions: Calculations and Conditional Display*.

5.3.5 Disabling paste

Text-entry question types (`field`, `num_field`, `multi_field`, and the free-text inputs of `checklist` and `radiolist`) accept a `disable_paste` property. When `true`, paste and drag-drop into that question's input are blocked so the answer has to be typed. To disable paste for every text input in the study at once, set `DISABLE_PASTE = true` in your config instead. See *Built-in Question Types* for the per-type property and *Data Quality* for how this fits with interaction logging.

5.3.6 Custom question types

If none of the built-in types fit your needs, you can define a custom type by creating a Jinja2 HTML template. BOFS discovers any `.html` file in a `templates/questions/` directory and registers the filename (without `.html`) as a valid question type. Template lookup follows the same precedence order as all BOFS templates; see *Templates and Jinja2* for details.

Creating a custom type

Place a file at `templates/questions/<your_type>.html` in your project directory (or in a blueprint). In a questionnaire JSON, set `"questiontype"` to the filename without the extension:

```
{
    "id": "agreement",
    "questiontype": "custom_scale",
    "instructions": "How much do you agree with this statement?",
    "low_label": "Strongly Disagree",
    "high_label": "Strongly Agree"
}
```

Template variables

Every custom question template receives these variables:

- `question` — the full question dict from the JSON, including any custom properties you added (`low_label`, `high_label`, etc.).
- `participant` — the current `Participant` model instance. Provides access to previously submitted questionnaire responses via `participant.questionnaire('name').field`.
- `session` — the Flask session dict. Injected by Flask into every Jinja context. Contains `condition` (the participant's assigned condition number) and other session keys set by BOFS or your own blueprints.

The template must produce one or more `<input>` elements whose `name` attributes match the question's `id` (or, for multi-value types, the sub-question IDs) so that the form submission is captured and stored correctly.

Example template:

```
<div class="custom-scale">
    <p>{{ question.instructions }}</p>
    <div class="scale-container">
        {% for i in range(1, 8) %}
        <label>
            <input type="radio" name="{{ question.id }}" value="{{ i }}">
            {{ i }}
```

(continues on next page)

(continued from previous page)

```
            {% if i == 1 %}<small>{{ question.low_label }}</small>{% endif %}
            {% if i == 7 %}<small>{{ question.high_label }}</small>{% endif %}
        </label>
        {% endfor %}
    </div>
</div>
```

The multiple-IDs pattern

A custom type can save multiple values per question by using a nested `questions` list. Each entry in that list needs its own `id`; those IDs become the database column names.

```
{
    "questiontype": "contact_form",
    "instructions": "Contact Information",
    "questions": [
        {"id": "first_name"},
        {"id": "last_name"},
        {"id": "phone"}
    ]
}
```

In the template, iterate over `question.questions`:

```
{% for sub in question.questions %}
    <input type="text" name="{{ sub.id }}" placeholder="{{ sub.id }}">
{% endfor %}
```

Some built-in types also use this pattern (`radiogrid`, `checklist`). The `EXPANDED_TYPES` mechanism in `BOFS/validation.py` covers the special case of a single top-level `id` that expands to multiple suffixed columns — `video` and `audio` work this way, writing `<id>_started`, `<id>_ended`, and `<id>_watched` / `<id>_listened` to the database.

5.3.7 Question ID naming rules

Question IDs become column names in the questionnaire's database table — and therefore the column headers in your exported CSV — and are read back as Python attributes (e.g. `participant.questionnaire('demographics').age`). The following rules apply to both question field IDs and `participant_calculations` keys:

- Must match the pattern `[A-Za-z_][A-Za-z0-9_]*` — a letter or underscore, then any mix of letters, digits, and underscores.
- Must not be a Python keyword (`class`, `return`, `for`, etc.). Keywords are syntax errors when used as attribute names in templates and custom code.
- Must not be one of the reserved BOFS column names: `participantID`, `participant`, `tag`, `timeStarted`, `timeEnded`, `duration`.
- Must not be one of the reserved expression names: `condition`, `tables`. These have special meaning inside `show_if` and `participant_calculations` expressions.
- Must be unique within a questionnaire. Duplicate IDs in the same file are a validation error.

Column type inference

BOFS infers the SQLAlchemy column type from the question type:

Question type	Column type	Notes
`slider`, `num_field`, `checklist`	`INTEGER`	
`image_select`	`INTEGER` or `FLOAT`	Inferred from the `value` fields in the `images` list; falls back to `TEXT` when values are mixed or non-numeric.
Everything else	`TEXT`	

To override the inferred type, add `"datatype"` to the question definition with one of: `"integer"`, `"float"`, `"string"`, `"datetime"`, or `"boolean"`.

5.3.8 Modifying questionnaires with existing data

The database schema for a questionnaire is derived from its question IDs. Changing the schema after participants have submitted responses needs care.

During development, delete the database file (e.g. `your_study.db`) and restart BOFS. The schema is recreated from scratch on the next run.

With live participant data, three paths are available:

1. **Admin panel preview** — visit `/admin` and use "Preview Questionnaire". BOFS will offer to add columns for new question IDs without touching existing data.
2. **Drop the questionnaire table** — deletes responses for that questionnaire while leaving other tables (demographics, custom data) intact.
3. **Manual schema migration** — alter the database directly using SQL. This preserves all data but requires knowledge of the underlying schema.

Changing or removing an existing question ID breaks the link to previously collected responses. Back up the database before making schema changes.

Questionnaire JSON files are loaded at startup. Restart BOFS after editing a questionnaire file.

Orphaned columns (columns that exist in the database but are no longer defined in the JSON) are preserved with `NULL` for new submissions and flagged as validation warnings at startup.

5.4 Expressions: Calculations and Conditional Display

Three places in BOFS take a small expression written over a participant's responses: the `participant_calculations` block on a questionnaire and per-question `show_if` expressions (both described in *Questionnaire Properties*), and per-page `show_if` expressions inside `PAGE_LIST` (described in *Conditions and Branching*). They all accept the same syntax, so this page is the canonical reference for the syntax itself.

The same syntax is also available from templates and custom blueprint code via `participant.evaluate(expression)` — see *Using Participant Data* for that surface.

5.4.1 Where expressions are used

Where you write it	When it runs	A field name on its own refers to
`participant_calculations` (questionnaire)	When data is exported	A field on the same questionnaire row
Question-level `show_if` (questionnaire)	Live, in the browser	Another field on the same page
Page-level `show_if` (`PAGE_LIST`)	At each page navigation	A field on any prior questionnaire

Each expression is checked once when BOFS launches. A typo or an unsupported piece of syntax raises an error at startup, not silently at the moment a participant lands on the page.

5.4.2 Syntax

Expressions look like Python expressions, restricted to a small safe subset. Whether your expression runs on the server (calculations, page skipping) or in the browser (live question hiding), the result is the same — the syntax and the rules are identical.

You can use:

- Arithmetic: `+`, `-`, `*`, `/`, integer division `//`, remainder `%`
- Comparisons: `<`, `<=`, `>`, `>=`, `==`, `!=` (and the chained form, e.g. `0 < x < 10`)
- Logic: `and`, `or`, `not`
- Membership: `in`, `not in` (against a list, e.g. `country in ['US', 'CA']`, or a string)
- If/else as a value: `a if condition else b`

Values you can write directly:

- Numbers (`42`, `3.14`)
- Strings, with single or double quotes (`'High'`, `"Other"`)
- `True`, `False`, `None`
- Lists: `[1, 2, 3]`, or lists of expressions like `[q1, 8 - q2, q3]`

Functions you can call:

- Aggregates: `mean`, `median`, `stdev`, `std`, `var`, `variance`
- Common: `len`, `min`, `max`, `sum`, `abs`, `round`
- Type conversion: `int`, `float`, `str`, `bool`

What you cannot use: arbitrary function calls (only the names above are permitted), method calls on values (`x.lower()`), indexing into a list with brackets (`items[0]`), and Python-only constructs such as `lambda` or list comprehensions. If a participant's questionnaire response needs reshaping in those ways, do it in your analysis script after export rather than inside an expression.

5.4.3 Referring to participant data

A field name on its own — `age`, `q1`, `01_inv` (field IDs that start with a digit work too) — refers to a stored value. What it actually resolves to depends on where you wrote it:

- Inside `participant_calculations`, it is a field from the same questionnaire's row (the one whose calculation is being computed).
- Inside a question-level `show_if`, it is another field on the same page, read live from the browser as the participant types or clicks.

- Inside a page-level `show_if`, it is looked up across every questionnaire the participant has already submitted; the most recent matching submission wins.

Inside a page-level `show_if`, three bare names are reserved and resolve to columns on the `Participant` row rather than to a questionnaire field:

- `condition` — the participant's assigned condition number. `show_if = "condition == 1"` keeps the page only for participants in condition 1.
- `source` — the recruitment-channel tag captured from the `?source=` URL parameter (see *Recruiting via MTurk or Prolific*). `show_if = "source == 'reddit'"` keeps the page only for participants who arrived with `?source=reddit`.
- `end_reason` — the reason stamped when a participant reaches an `end` page (`"complete"` by default, or the `<reason>` from an `end/<reason>` entry; see *Setting Up Your Page Flow*).

Because these names are reserved, you cannot use any of them as a questionnaire field ID or as a key in `participant_calculations`.

5.4.4 Referring to table values

Page-level `show_if` can also reference per-participant aggregates exported by a *JSONTable*. The form is `tables.<table_name>.<column>`, where `<column>` is the export-column name declared in the table's `exports` block:

```
PAGE_LIST = [
    {name="Consent", path="consent"},
    {name="Practice", path="task/practice"},
    {name="Practice debrief", path="debrief/practice", show_if=
↪"tables.practice_trials.accuracy < 0.6"},
    {name="End", path="end"}
]
```

The aggregate is computed once per evaluation by running the same query the data export uses, restricted to the current participant. If the participant has no rows in the table, the value resolves to `None` and the page stays visible (the expression is treated as undecided).

Only the columns listed under a table's `exports` block are reachable this way — raw rows are not. `tables` is reserved at the top of an expression: a questionnaire or table file cannot be named `tables`, and questionnaire field IDs and `participant_calculations` keys cannot be named `tables` either.

Subscripting a `group_by` export

An export that uses `group_by` produces one value per level. The expression engine resolves the bare reference to a per-level dict, which can be indexed in either dotted or bracket form:

Form	Resolves to
`tables.<name>.<col>.<key>`	Dotted literal key. Digit-only segments are coerced to `int` so `round_score.1` matches an integer key.
`tables.<name>.<col>[<literal>]`	Bracket literal key (int or quoted string).
`tables.<name>.<col>[<expr>]`	Bracket expression key — the inner expression is evaluated against the same env, so `tables.scores.round_score[condition]` selects the row keyed by the participant's condition number.
`tables.<name>.<col>`	Bare reference resolves to the whole dict, useful for `len(tables.foo.bar)` or `mean([tables.foo.bar[1], tables.foo.bar[2]])`.

A missed key (no matching level) makes the expression undecided — page-level `show_if` keeps the page visible, and inline `{{ }}` substitution renders empty.

Multi-column `group_by` (a list of grouping columns, e.g. `"group_by": ["phase", "block"]`) produces a dict keyed by Python tuples (one value per grouping column). Tuple literals are not part of the expression syntax, so multi-column group_by exports cannot be subscripted from an expression. Either declare separate scalar exports per cell or read the dict from `participant.table('foo').col` in a Jinja template or custom blueprint.

Page-level `show_if` also accepts more specific reference forms when the same questionnaire appears in `PAGE_LIST` multiple times under different tags (for example, a wellbeing questionnaire filled in once before an intervention and once after):

Form	Resolves to
`qname.field`	Most recent submission of `qname` (any tag)
`qname.tag.field`	The row of `qname` submitted under the given tag
`qname..field`	The row of `qname` with the empty (default) tag

The `tag` segment matches the second part of paths like `questionnaire/qname/tag`. The qualified forms only apply to page-level `show_if` — the other two surfaces always look at a single questionnaire row, so there is nothing to disambiguate.

5.4.5 Calculations on a questionnaire

A `participant_calculations` block computes derived values from a participant's responses — scale scores, reverse-scored items, categorical bins. The result is stored alongside the raw responses and shows up as its own column in the CSV export. The block lives inside a questionnaire's JSON file (see *Questionnaire Properties*).

```
{
    "questions": [
        {"id": "ext_1", "questiontype": "slider", "instructions": "I am outgoing",
         "left": "Strongly disagree", "right": "Strongly agree", "tick_count": 7},
        {"id": "ext_2", "questiontype": "slider", "instructions": "I am reserved␣
↪(reverse scored)",
         "left": "Strongly disagree", "right": "Strongly agree", "tick_count": 7},
        {"id": "ext_3", "questiontype": "slider", "instructions": "I am full of␣
↪energy",
         "left": "Strongly disagree", "right": "Strongly agree", "tick_count": 7}
    ],
    "participant_calculations": {
        "extraversion": "mean([ext_1, 8 - ext_2, ext_3])",
        "high_extraversion": "mean([ext_1, 8 - ext_2, ext_3]) > 5"
    }
}
```

Each key under `participant_calculations` becomes an export column with the computed value. The keys must follow the same naming rules as field IDs (start with a letter or underscore, alphanumerics and underscores after) and must not collide with the built-in BOFS columns `participantID`, `timeStarted`, `timeEnded`, `tag`, or `duration`.

5.4.6 Hiding a question

A question inside a questionnaire (see *Questionnaire Properties*) can declare a `show_if` expression that branches on the participant's other answers on the same page. The expression is checked live in the browser as the participant fills out the page; when it is false, the question is hidden and the form treats its inputs as if they weren't there — including any required fields, which won't block submission.

```
{
    "questions": [
        {"id": "age", "questiontype": "num_field", "instructions": "How old are you?"}
↪,
        {"id": "guardian", "questiontype": "field", "show_if": "age < 18",
         "instructions": "Who is your parent or guardian?"},
        {"id": "lang", "questiontype": "drop_down",
         "items": ["English", "French", "Spanish", "Other"],
         "instructions": "Primary language"},
        {"id": "lang_other", "questiontype": "field", "show_if": "lang == 'Other'",
         "instructions": "Please specify"}
    ]
}
```

When a question is hidden at submission time, no value is sent and the database column receives its default — an empty string for text fields, `0` for numeric fields, and so on.

5.4.7 Skipping a page

Any entry in `PAGE_LIST` (including entries inside a `conditional_routing` block) can carry a `show_if` expression that branches on the participant's stored answers. When it evaluates false, the page is removed from that participant's flow — the next/back navigation skips past it and it does not appear in their breadcrumb. `PAGE_LIST` itself lives in your project's configuration file (see *Conditions and Branching*).

```
PAGE_LIST = [
    {name="Consent", path="consent"},
    {name="Demographics", path="questionnaire/demographics"},
    {name="Followup", path="questionnaire/followup", show_if="demographics.age < 18"},
    {name="End", path="end"}
]
```

For repeated-measures designs that fill in the same questionnaire more than once under different tags, the qualified reference forms compare specific submissions:

```
PAGE_LIST = [
    {name="Consent", path="consent"},
    {name="Pre-survey", path="questionnaire/wellbeing/before"},
    {name="Intervention", path="task/intervention"},
    {name="Post-survey", path="questionnaire/wellbeing/after"},
    {name="Improvement debrief", path="debrief/improved", show_if=
↪"wellbeing.before.score < wellbeing.after.score"},
    {name="End", path="end"}
]
```

When a page-level `show_if` references a questionnaire the participant has not yet submitted, the page stays visible — the expression is treated as undecided rather than removing a page on the basis of data that hasn't been collected.

5.4.8 Errors and validation

- A bad expression — unsupported syntax or a disallowed function — raises an error when BOFS starts up, naming the calculation or the question or the page that contains it.
- A reference to a field ID that doesn't exist on the relevant questionnaire is reported as a validation warning at startup, listing the questionnaire's known field IDs alongside the unknown one to help you catch typos.

5.5 Custom Tables

Custom tables let you record structured experiment data (trial responses, event logs, timing measurements) to the database outside the questionnaire system. A table is defined by a JSON file; BOFS creates the corresponding database table automatically at startup.

For a guided introduction to writing and reading data with custom tables, see *Storing Custom Data*.

5.5.1 Table Definition Files

Place `<name>.json` files in the `tables/` directory at your project root, or inside a blueprint at `<blueprint_name>/tables/`. The file name (without extension) becomes the table name used in routes and Python access.

See *Blueprints and Routes* for how blueprint-scoped tables are discovered.

5.5.2 Column Types

Each column in the `"columns"` object specifies a `"type"` and an optional `"default"`. Valid types are:

Type	SQLite storage	Notes
`integer`	`INTEGER`	Default is `0` unless overridden.
`float`	`NUMERIC`	Default is `0` unless overridden.
`boolean`	`BOOLEAN`	Default is `false` unless overridden. Accepts `true`/`false`, `1`/`0`, `"yes"`/`"no"`, `"on"`/`"off"` (case-insensitive) from form-encoded payloads.
`string`	`TEXT`	Default type when `"type"` is absent. Default value is `""` unless overridden.
`datetime`	`DATETIME`	No `"default"` is accepted; the column defaults to `datetime.min`. Send ISO 8601 strings from JavaScript (a standard date-time format, e.g. `2026-03-15T14:30:00`).
`json`	`TEXT`	No `"default"` is accepted; the column defaults to `NULL`. Stores arrays, objects, or any JSON-serialisable structure. The HTTP API serialises and deserialises transparently — JavaScript always receives a native object or array.

5.5.3 Example Table with All Column Types

```
{
  "columns": {
    "integer_column": {
      "type": "integer",
      "default": 0
    },
    "float_column": {
      "type": "float",
      "default": 0
    },
    "boolean_column": {
      "type": "boolean",
      "default": true
    },
    "string_column": {
      "default": "this is a test"
    },
```

(continues on next page)

(continued from previous page)

```
    "datetime_column": {
      "type": "datetime"
    },
    "json_column": {
      "type": "json"
    }
  }
}
```

5.5.4 Auto-added Columns

Every custom table automatically receives two additional columns that you do not declare and should not include in POST payloads:

- `participantID` — foreign key (a link to the participant's record), populated from the current session.
- `timeSubmitted` — `DATETIME`, set to the server's current UTC time on each insert.

When the table is routed to a non-default database via the `"database"` field (see below), `participantID` is a plain indexed integer column rather than a foreign key, since SQLAlchemy can't enforce a FK across separate engines.

5.5.5 Routing a Table to a Different Database

A custom table can write to a database other than the project's main one. Add a top-level `"database"` field naming an entry from `[SQLALCHEMY_BINDS]` in `config.toml`:

```
{
  "database": "pii",
  "columns": {
    "follow_up_email": {"type": "string"}
  }
}
```

The same option is available on questionnaire JSON files. Trade-offs and the per-bind admin export endpoint are covered in *The Database Layer*.

5.5.6 Naming Rules

Column names and export field names must start with a letter or underscore and contain only letters, digits, and underscores. Python keywords (`class`, `return`, `for`, etc.) are not allowed.

Within a single table, no export field name may duplicate a column name. The `TableAccessor` (returned by `participant.table('name')`) resolves attribute access to exports; a collision would shadow the raw column.

5.5.7 Calculated Export Fields

The `"exports"` block defines per-participant aggregations that appear as columns on the admin panel's Export page and as attributes on the `TableAccessor`. Each entry in the array is a single export definition.

Export Object Keys

Key	Required	Description
`fields`	Yes	A dict mapping output column names to SQL expressions. Values may be bare column names (`"my_column"`) or aggregate expressions (`"sum(my_column)"`). When there is more than one row per participant, include an aggregate (summary) function: `MIN`, `MAX`, `SUM`, `COUNT`, or `AVG`.
`filter`	No	A SQL `WHERE` expression restricting which rows are included (e.g. `"my_column > 1"` or `"levelName IN ('Intro1', 'Intro2')"`).
`group_by`	No	A column name (string) or list of column names to group by. Each unique value in the grouped column produces a separate result. In the data export each level appears as a suffixed column (`<field>_<level>`). In templates, a `group_by` export returns a dict keyed by group value rather than a scalar.
`order_by`	No	A SQL `ORDER BY` expression controlling the column order in the data export.
`having`	No	A SQL `HAVING` expression. Only valid when `group_by` is also present.

Simple Example

A table that stores numbers entered by participants and exports a count per participant:

```
{
  "columns": {
    "your_number": {"type": "integer"}
  },
  "exports": [
    {
      "fields": {"total_numbers": "count(your_number)"}
    }
  ]
}
```

Complex Example

A game-progress table with multiple export definitions, including a `group_by` over session name:

```
{
  "columns": {
    "finishedLevel": {"type": "integer"},
    "levelName": {},
    "deathCount": {"type": "integer"},
    "levelTime": {"type": "float"},
    "sessionName": {}
  },
  "exports": [
    {
      "group_by": "sessionName",
      "order_by": "sessionName",
      "fields": {
        "totalLevelsFinished": "sum(finishedLevel = 'True')",
        "totalDeathCount": "sum(deathCount)"
      }
```

(continues on next page)

(continued from previous page)

```
    },
    {
      "filter": "levelName IN ('Intro1', 'Intro2', 'Intro3')",
      "fields": {
        "tutorialLevelsTime": "sum(levelTime)",
        "tutorialLevelsCompleted": "sum(finishedLevel = 'True')"
      }
    }
  ]
}
```

5.5.8 Reading Export Values from Templates

Each scalar (non-`group_by`) export field is accessible as an attribute on the `TableAccessor` returned by `participant.table('name')`:

```
{% set trials = participant.table('cognitive_task') %}
<p>Total trials: {{ trials.totalLevelsFinished }}</p>
```

The aggregate runs scoped to the current participant. All computed values are memoised on the accessor instance.

`group_by` exports return a dict keyed by group value (or a tuple of values when `group_by` is a list). Access a specific level by subscript:

```
{% set trials = participant.table('game_progress') %}
<p>Deaths in session A: {{ trials.totalDeathCount['session_a'] }}</p>
```

To iterate all levels:

```
{% for session, deaths in trials.exports['totalDeathCount'].items() %}
  <p>{{ session }}: {{ deaths }}</p>
{% endfor %}
```

In the CSV data export, each level appears as a separate suffixed column (`totalDeathCount_session_a`, `totalDeathCount_session_b`, etc.).

For `show_if` predicates and `participant.evaluate()`, scalar export fields are reachable via the `tables.<name>.<column>` reference form. See *Expressions: Calculations and Conditional Display* for expression syntax.

For raw row access, iterate `participant.table('name')` directly or use `participant.table('name').rows`. The accessor proxies `__iter__`, `__len__`, and `__getitem__` to `rows`.

5.5.9 Writing Data from JavaScript

POST to `/table/<name>` to insert one or more rows. The server attaches `participantID` and `timeSubmitted` from the session — do not include those fields in the payload.

See *Built-in Routes* for the full route reference.

Single Row — JSON

```
fetch('/table/my_task', {
    method: 'POST',
    headers: {'Content-Type': 'application/json'},
```

(continues on next page)

(continued from previous page)

```
    body: JSON.stringify({score: 42, label: 'correct'})
});
```

On success the endpoint returns `204 No Content`. On a validation error it returns `400 Bad Request`.

Batch Insert — JSON Array

Send a JSON array to insert multiple rows in a single request. All rows are committed together; if any row is malformed the entire batch is rolled back and `400` is returned.

```
const trials = [
    {trial: 1, rt: 312, response: 'left'},
    {trial: 2, rt: 498, response: 'right'},
    {trial: 3, rt: 271, response: 'left'},
];

fetch('/table/my_task', {
    method: 'POST',
    headers: {'Content-Type': 'application/json'},
    body: JSON.stringify(trials)
});
```

Single Row — Form-encoded

Form-encoded POSTs are accepted for compatibility with HTML forms or older code. Batch inserts are not supported in this mode.

```
fetch('/table/my_task', {
    method: 'POST',
    body: new URLSearchParams({score: 42, label: 'correct'})
});
```

Storing Structured Data in a `json` Column

Pass a JavaScript object or array directly in the JSON payload — the server serialises it automatically:

```
const mouseTrajectory = [
    {t: 0,  x: 512, y: 300},
    {t: 16, x: 510, y: 298},
    {t: 32, x: 505, y: 290},
];

fetch('/table/my_task', {
    method: 'POST',
    headers: {'Content-Type': 'application/json'},
    body: JSON.stringify({trial: 1, trajectory: mouseTrajectory})
});
```

When the row is retrieved via GET, `trajectory` is already a JavaScript array — no client-side `JSON.parse` is needed.

5.5.10 Reading Data from JavaScript

A GET request to `/table/<name>` returns a JSON array of all rows belonging to the current participant.

```
fetch('/table/my_task')
    .then(r => r.json())
    .then(rows => {
        // rows is an array of objects, e.g. [{trial: 1, score: 42}, ...]
        console.log(rows);
    });
```

Query-string parameters are applied as exact-match filters. The comparison is string-cast, so this is best suited to integer or string columns with known discrete values:

```
// Retrieve only rows where score equals 42
fetch('/table/my_task?score=42')
    .then(r => r.json())
    .then(rows => console.log(rows));
```

Note

Query-string filters use string-cast equality (`CAST(column AS TEXT) = 'value'`). Range queries and other filtering must be done in Python or post-processed on the client.

5.5.11 Accessing Tables from Python

Import `db` from `BOFS.globals` to access SQLAlchemy sessions and dynamically created table classes.

```
from BOFS.globals import db
```

Custom table classes are registered on the `db` object by the table file name. For example, a table defined in `tables/answers.json` is accessible as `db.answers`.

Querying

Use `db.session.query` with standard SQLAlchemy ORM patterns:

```
# All rows for participants who completed the experiment
rows = db.session.query(db.answers).filter(
    db.answers.participantID == participant_id
).all()
```

See the SQLAlchemy ORM querying documentation for the full query API. For database internals and session management, see *The Database Layer*.

Inserting

Create an instance of the table class, set attributes, then add and commit:

```
from flask import session

entry = db.answers()
entry.participantID = session['participantID']
entry.answer = request.form['answer']
```

(continues on next page)

(continued from previous page)

```
db.session.add(entry)
db.session.commit()
```

`timeSubmitted` is populated automatically by its column default and does not need to be set.

5.6 Participant Data API

Reference for the `participant` object and all template variables BOFS injects into every rendered template. For guidance on when and why to use each surface, see *Using Participant Data*. For template variable injection mechanics, see *Templates and Jinja2*.

On this page

5.6.1 Template Variables

BOFS makes the following variables available inside every template it renders (via a Flask context processor).

Variable	Description
`participant`	The current *Participant object*. `None` when no participant is in session (admin previews, the consent page before submission, error pages). Guard with `{% if participant %}` in templates that may render in those contexts.
`session`	Data tied to the current participant's visit (Flask's session dictionary). Contains the *session fields* described below. Populated progressively as the participant moves through the experiment.
`debug`	`True` when BOFS is running with the `-d` flag.
`config`	The project's TOML configuration, accessible as `config['KEY']` or `config.KEY`. See *Config Access* below.
`flat_page_list`	The current participant's filtered page sequence as a list of path strings, with `show_if` predicates and condition routing applied. Empty or condition-filtered pages are excluded.

5.6.2 The Participant Object

Attributes

These are database columns on the `Participant` model, readable directly in templates and custom blueprint code.

Attribute	Type	Description
`participantID`	`int`	Auto-assigned primary key. Unique per participant within the project database.
`condition`	`int`	Assigned condition number, starting at 1. `0` means no condition has been assigned (single-condition projects, or before assignment runs).
`externalID`	`str`	External worker or participant identifier (MTurk Worker ID, Prolific ID, or the value passed via the `external_id` query parameter). Empty string when not set. Also accessible as `mTurkID`, an alias kept for backward compatibility with code written before the rename.
`source`	`str` or `None`	Recruitment source (e.g. `"prolific"`, `"reddit"`, `"email"`). Set from a `?source=` URL query parameter, or inferred when a `PROLIFIC_PID` parameter is present. `None` for participants who arrived without a source hint, and on rows created before this column existed.
`timeStarted`	`datetime`	UTC timestamp recorded after the participant submits the consent page (when their session is created). Naive datetime — no timezone info attached.
`timeEnded`	`datetime` or `None`	UTC timestamp recorded when the participant reaches the final page. `None` until the participant finishes.
`finished`	`bool`	`True` once the participant has completed the experiment.
`end_reason`	`str` or `None`	Why the participant's session ended. Stamped when the participant reaches `/end/<reason>` — one of the built-in reasons (`complete`, `bot`, `quota_full`, `duplicate`) or any researcher-defined string. `None` for participants who abandoned mid-experiment without reaching an end page.
`code`	`str`	Completion code generated at the end of the experiment. Empty string until generated.

Methods

`participant.`**`display_duration`**`()` → str

Returns a human-readable string representing elapsed time (`"HH:MM:SS"` or shorter form) when the participant has finished, or a status string (`"In Progress"` or `"Abandoned"`) when they have not.

When `timeEnded` is `None`, the return value is `"In Progress"` if the participant is currently active, or `"Abandoned"` if they have been inactive beyond the `ABANDONED_MINUTES` threshold.

`participant.`**`questionnaire`**`(`*name*, *tag=''*`)` → row

Returns the participant's most recent response row for the questionnaire named `name`, optionally filtered to the submission filed under `tag`.

When the participant has not yet submitted the questionnaire, returns a blank-default row whose field attributes hold each column's default value (empty string for text fields, `0` for numeric fields, `False` for boolean fields). Accessing any field on a blank-default row succeeds without raising — use *`has_questionnaire()`* to distinguish a real submission from a defaulted one.

When the same questionnaire has been submitted more than once under the same tag, the most recent submission (by `timeEnded`) is returned.

Field values and any `participant_calculations` columns are available as attributes on the returned row

(e.g., `row.age`, `row.extraversion_score`). For the rules governing field IDs and calculated columns, see *Questionnaire Properties*.

Parameters

- `name` (`str`) — The questionnaire filename without the `.json` extension.
- `tag` (`str`, default `""`) — The tag under which the questionnaire was submitted. Corresponds to the optional third segment of the URL path (`questionnaire/<name>/<tag>`). Pass the empty string (or omit) for untagged submissions.

Returns a SQLAlchemy model instance with all questionnaire fields as attributes.

`participant.`**`has_questionnaire`**(*name*, *tag=''*) → bool

Returns `True` when the participant has at least one stored submission of `name` with the given `tag`.

`questionnaire()` always returns a row (falling back to a blank default), so this method is the only way to tell whether a real submission exists without inspecting field values.

Parameters

- `name` (`str`) — The questionnaire filename without the `.json` extension.
- `tag` (`str`, default `""`) — Tag to match. Pass the empty string (or omit) for untagged submissions.

Returns `bool`.

`participant.`**`questionnaire_interactions`**(*name*, *tag=''*) → list

Returns a list of interaction event rows for the questionnaire named `name`, ordered by `timestamp` ascending. Requires `LOG_QUESTIONNAIRE_INTERACTIONS = true` in the project configuration; returns an empty list otherwise, or when the participant has not yet reached the questionnaire.

Each row in the list has the following attributes:

Attribute	Type	Description
`questionnaire`	`str`	Questionnaire name.
`tag`	`str`	Submission tag (`"0"` for untagged submissions internally; treat empty string and `"0"` as equivalent when filtering).
`questionID`	`str`	The `id` of the question field that generated the event.
`eventType`	`str`	One of `focus`, `blur`, `change`, `paste`, or `visibility`. Text inputs also log authenticity signals such as keystroke counts, backspace counts, paste character counts, focus duration, and time-to-first-keystroke as additional event records.
`timestamp`	`datetime`	UTC timestamp of the event.
`value`	`str` or `None`	The field value captured at event time. `None` for event types that do not carry a value.

Parameters

- `name` (`str`) — The questionnaire filename without the `.json` extension.
- `tag` (`str`, default `""`) — Tag to match.

Returns `list` of `QuestionnaireInteraction` model instances.

`participant.`**`table`**(*name*) → TableAccessor

Returns a *TableAccessor* for the JSONTable named `name`. Raises `KeyError` when no table by that name is loaded.

For details on defining tables and their `exports` block, see *Custom Tables*.

Parameters

- `name` (`str`) — The JSONTable filename without the `.json` extension.

Returns *TableAccessor*.

`participant.`**`evaluate`**(*expression*) → value or None

Evaluates a BOFS expression string against this participant's stored data and returns the result. Uses the same syntax as `show_if` predicates and `participant_calculations` in questionnaire JSON files.

Returns `None` when the expression cannot be parsed, contains unsupported syntax, or references a questionnaire the participant has not yet submitted. A template `{% if participant.evaluate(...) %}` therefore falls through to the `else` branch rather than raising.

For direct field access, `participant.questionnaire('name').field` is shorter and more idiomatic. Reach for `evaluate()` when you want to share an expression string with a `show_if` predicate in a configuration file, or when you need to build the expression dynamically.

Parameters

- `expression` (`str`) — A BOFS expression string. See *Expressions: Calculations and Conditional Display* for full syntax.

Returns the expression result (bool, int, float, str, or list), or `None` on failure.

5.6.3 TableAccessor

`TableAccessor` is returned by *`participant.table()`*. It exposes the participant's raw rows and any per-participant aggregates declared in the table's `exports` block.

Iteration and indexing

The accessor proxies `__iter__`, `__len__`, `__getitem__`, and `__bool__` to its `rows` list, so the following patterns all work without accessing `.rows` explicitly:

- `{% for row in participant.table('foo') %}`
- `participant.table('foo')|length`
- `participant.table('foo')[0]`
- `{% if participant.table('foo') %}` — `False` when the participant has no rows.

`TableAccessor.`**`rows`**

The participant's raw rows in the table as a list of model instances. Returns `[]` when the participant has no rows.

Export field attributes

Every field declared in the table's `exports` block is accessible as an attribute on the accessor. Each aggregate is computed once per accessor instance and cached; reading the same attribute twice in a template runs the query only once.

Scalar exports (no `group_by`) return a single value, or `None` when no rows match the export's `filter` and `having` clauses.

`group_by` exports return a dict keyed by the group value. When `group_by` is a list of columns, the key is a tuple of values in declaration order. An empty dict means the participant had no rows that satisfied the filter.

Page-level `show_if` expressions can consume scalar aggregates from a `TableAccessor`; `group_by` exports are not reachable from `show_if` (the reference resolves to undecided). Access `group_by` results from templates or custom blueprint code instead.

`TableAccessor.`**`exports`**

All export aggregates as a plain `dict`. Scalar exports map to their value; `group_by` exports map to a dict keyed by group value or tuple. Reading `.exports` computes and caches every aggregate at once, then returns the dict. Useful when you want to iterate: `{% for key, value in participant.table('foo').exports.items() %}`.

5.6.4 Session Fields

The `session` variable is the Flask session dictionary. It is populated progressively as the participant moves through the experiment, so some keys may be absent early in the flow. Use `session.get('key')` or `'key' in session` to guard against missing keys in templates that may render before a value is set.

Key	Type	Description
`participantID`	`int`	Set when the participant's database row is created (after consent is submitted). Absent before consent.
`condition`	`int`	The participant's assigned condition number. Set at the same time as `participantID`.
`currentUrl`	`str`	The path of the current page in the experiment sequence, as tracked by the page-flow system. Absent before the first page navigation.
`mTurkID`	`str`	External worker or participant identifier. Set from query parameters (`workerId`, `PROLIFIC_PID`, or `external_id`) or copied from the participant row. Absent when not provided.
`code`	`str`	Completion code. Set when the participant reaches the end page. Absent until then.

5.6.5 Config Access

The `config` variable provides access to all TOML configuration settings. Read values using dictionary syntax or attribute access:

```
{{ config['TITLE'] }}
{{ config.TITLE }}
```

All keys defined in the project's `.toml` file are available, including custom keys you define for your own project logic. Built-in keys are documented in *Configuration Reference*.

The `config` object is Flask's application config, a dict-like object. Accessing a missing key with `.get()` returns `None`, but bracket access raises `KeyError` (an error) for undefined keys.

5.7 Built-in Routes

For a guided introduction to the page flow these routes plug into, see *Setting Up Your Page Flow*.

5.7.1 Routes For Use in `PAGE_LIST`

The First Page

Your project should always be set up to use one of these pages first.

`BOFS.default.views.`**`route_consent`**`()`

`/consent`

This shows a consent form. Upon submission, the participant entry is created in the database, they are assigned a condition, and the session variables are set. This is typically the first page you will use in `PAGE_LIST`. If not, use one of

- /consent_nc
- /create_participant
- /create_participant_nc

BOFS.default.views.**route_consent_nc**()

/consent_nc

This acts just like /consent, except it does not assign the user a condition (defaults to 0).

BOFS.default.views.**route_create_participant**()

/create_participant

This creates the participant in the database and sets up the session variables. Use if you don't need to show a consent form.

BOFS.default.views.**route_create_participant_nc**()

/create_participant_nc

This creates the participant in the database and sets up the session variables. Use if you don't need to show a consent form.

This does not assign the participant a condition (defaults to 0) and so could be used in conjunction with /assign_condition.

Subsequent Pages

BOFS.default.views.**route_assign_condition**()

/assign_condition

Typically, conditions are assigned upon consent. Use this page if you want to assign a condition later on in the experiment. This might be used, for example, after several initial questionnaires, so that participants who fail to actually attempt the task don't end up getting assigned a condition.

If you used /consent_nc or /create_participant_nc, then you will need to use this to assign them to a condition other than 0.

BOFS.default.views.**route_external_id**()

/external_id or (for backwards compatibility) /startMTurk or /start_mturk

If we are using a platform where the user has a unique (and anonymous) ID associated with their account, then you can use this page to request that ID. This is set up to work with Mechanical Turk, but the template can be overwritten to request different types of ID. :return:

BOFS.default.views.**route_instructions**(*pageName*)

/instructions/<pageName>

Generic route to render instructions. The instructions are defined in HTML within a file and rendered in BOFS using the the templating system. A button to redirect to the next page in the study is shown after the instructions.

Instruction HTML files can be placed in the project root directory's templates folder in /templates/instructions/... or in one of your blueprint's templates folder in /<my_blueprint>/templates/instructions/....

The files must be in HTML format and use the .html extension. The pageName specified is the filename for the instructions, without the file extension.

Parameters

pageName – the name of the file to use to render the instructions (without the .html extension)

BOFS.default.views.**route_simple_html**(*pageName*)

`/simple/<pageName>`

Generic route to render simple Jinja 2 templates (or simple HTML pages) that do not need any additional Python code. The pages are defined in HTML/Jinja 2 within a file and rendered in BOFS using the templating system. Unlike the instruction pages, you are responsible for redirecting participants yourself (e.g., via a JavaScript redirect to `/redirect_next_page`. A generic POST request to this route (such as from a form submission) will also trigger a redirection to the next page.

Simple HTML files can be placed in the project root directory's templates folder in `/templates/simple/...` or in one of your blueprint's templates folder in `/<my_blueprint>/templates/simple/...`.

The files must be in HTML format and use the `.html` extension. The `pageName` specified is the filename for the html file, without the file extension.

Parameters

pageName – the name of the file to use to render the simple page (without the .html extension)

BOFS.default.views.**route_questionnaire**(*questionnaireName*, *tag=''*)

`/questionnaire/<questionnaireName>` or `/questionnaire/<questionnaireName>/<tag>`

Render a questionnaire with the specified name. The questionnaire should be defined as a JSON file (with a .json file extension) in the `/questionnaire` directory.

If the same questionnaire is going to be used twice, then use the URL that includes the <tag>, this allows you to define a name associated with that instance of the questionnaire. For example, "before" and "after" or "1" and "2".

Parameters

- **questionnaireName** – The name of the json file (without the .json extension).
- **tag** – If the same questionnaire is to be displayed more than once, provide it with a unique tag (e.g., "before" or "after" or "1")

BOFS.default.views.**route_end**()

`/end`

Ends the experiment, marks the participants as finished, and shows the user's completion code if they have been given one. Can also be configured to redirect to an external URL.

5.7.2 Developer Routes

Navigation

BOFS.default.views.**route_redirect_previous_page**()

`/redirect_previous_page`

Sends a user to the previous page. This is intended primarily for debugging purposes.

BOFS.default.views.**route_redirect_next_page**()

`/redirect_next_page`

This is the preferred way of sending a user to the next page.

BOFS.default.views.**route_redirect_from_page**(*page*)

`/redirect_from_page/<path:page>`

Redirect the user from a specific page.

Parameters

page – The page to start from, the user will be sent to next page in the list

BOFS.default.views.**route_redirect_to_page**(*page*)

/redirect_to_page/<path:page>

Redirect the user to a specific page path in PAGE_LIST

Parameters
page

BOFS.default.views.**route_current_url**()

/current_url

Returns
The current URL of the user. For a new user, it returns "/".

BOFS.default.views.**route_restart**()

/restart

Clears all participant/progress session state and redirects to the start of the experiment. Admin login state (loggedIn) is preserved so the admin does not need to re-authenticate after restarting a session.

Form submission

BOFS.default.views.**submit**()

/submit

Use this if you simply need to submit a form that redirects to the next page without doing anything with the form data.

BOFS.default.views.**route_questionnaire_question**(*question_type: str*)

/questionnaire_question/<questionType>

Render a specific question type for the questionnaire. Only accepts POST requests. Data posted to this route must be a JSON object of the question data.

Activity tracking

BOFS.default.views.**route_user_active**()

Custom tables

BOFS.default.views.**route_table**(*tableName*)

/table/<tableName>

Provides a simple API to get data from a table (via GET) or add data to a table (via POST).

Parameters
tableName – The name of the table, as it is in /app/tables (without the .json)

Returns
the return value is either JSONTable.handle_post() or JSONTable.handle_get() depending on the request type.

5.8 CLI Reference

BOFS provides two commands:

- BOFS init — create a new project interactively.
- BOFS run <config> (or simply BOFS <config>) — run an existing project.

5.8.1 init Command

`BOFS init` launches an interactive wizard that scaffolds a new project.

```
BOFS init
```

The wizard will prompt you for:

1. **Project name** - The directory name for your project (alphanumeric, hyphens, and underscores only)
2. **Project title** - The title displayed to participants in the browser
3. **Admin password** - Password for accessing the admin panel at `/admin`
4. **Features** - Optional features to include in your project:
    - External ID page (MTurk/Prolific participant tracking)
    - Instructions page
    - Simple custom page
    - Multiple conditions (experimental groups)
    - Pre/post questionnaires
    - Custom blueprint (Python routes)
    - Example questionnaires
    - Example JSON tables

After creating the project, the wizard will ask if you want to start it immediately. If you choose yes, BOFS will start the server in debug mode and open your default web browser to the project URL.

Example session:

```
$ BOFS init

BOFS Project Initialization Wizard
========================================

? Project name (directory name): my_experiment
? Project title (displayed to participants): My Experiment
? Admin password (default: admin): ********
? Select features to include: (Space to select, Enter to confirm)
  ❯ ◉ Example questionnaires - Include demo questionnaires showing different question␣
↪types
    ○ External ID page (MTurk/Prolific) - Collect participant IDs from recruitment␣
↪platforms
    ...

Creating project...

Created: my_experiment/
    |-- config.toml
    |-- consent.html
    `-- questionnaires/
        `-- survey.json

? Would you like to start your project now? Yes
```

(continues on next page)

(continued from previous page)

```
Starting project in debug mode...
Opening http://localhost:5000 in your browser...
```

5.8.2 run Command

The `run` command starts a BOFS project server. For backward compatibility, you can omit the `run` keyword.

```
BOFS run config_file.toml [OPTIONS]
BOFS config_file.toml [OPTIONS]  # Equivalent (backward compatible)
```

Where `config_file.toml` is the path to your TOML configuration file that defines your experiment setup.

5.8.3 run Command Options

config (Required)

Positional argument

The name or path of the configuration file to load. This should be a TOML file containing your experiment configuration.

```
BOFS run minimal.toml
BOFS run /path/to/my/experiment.toml
```

–debug, -d

Optional flag

Toggles debug mode, which provides enhanced development features:

- Enables a debugging toolbar in the web interface
- Provides detailed error messages and stack traces
- Enables Flask's built-in debugger
- Shows more verbose logging output

```
BOFS run config.toml --debug
BOFS run config.toml -d
```

> **Warning**
>
> Debug mode should **never** be used in production environments as it can expose sensitive information and security vulnerabilities.

–path, -p

Optional parameter

Specifies the working directory for your BOFS project. This is useful when your configuration file is in a different directory than your project files, or when you want to run BOFS from a different location.

```
BOFS run config.toml --path /path/to/project
BOFS run config.toml -p ./my_experiment
```

If not specified, BOFS will use the directory containing the configuration file as the working directory.

Stopping the Server

To stop a running BOFS server, press **Ctrl+C** in the terminal. The server will shut down gracefully.

–reloader-off, -r

Optional flag

When running in debug mode, disables the automatic reloader feature. The reloader normally restarts the server automatically when code changes are detected.

```
BOFS run config.toml --debug --reloader-off
BOFS run config.toml -d -r
```

This option is only meaningful when used in combination with `--debug` mode.

5.8.4 Common Usage Patterns

Basic Development

For standard development work:

```
BOFS run experiment.toml -d
```

This enables debug mode with auto-reloading for rapid development.

Production Testing

For testing in a production-like environment:

```
BOFS run experiment.toml
```

This runs without debug mode, similar to how it would run in production.

Custom Project Directory

When your project files are in a specific directory:

```
BOFS run config.toml -p /home/researcher/experiments/study1
```

Debug Without Auto-reload

For debugging without automatic restarts (useful when testing session persistence):

```
BOFS run experiment.toml -d -r
```

Running as Python Module

You can also run BOFS as a Python module:

```
python -m BOFS run config.toml
python -m BOFS run config.toml -d
```

This is particularly useful when your terminal does not recognize the `BOFS` command.

5.8.5 Integration with Development Workflow

Virtual Environments

When using virtual environments, activate your environment first:

```
# Linux/Mac
source bofs_venv/bin/activate
BOFS run experiment.toml -d

# Windows (Command Prompt)
.\bofs_venv\Scripts\activate.bat
BOFS run experiment.toml -d

# Windows (PowerShell)
.\bofs_venv\Scripts\Activate.ps1
BOFS run experiment.toml -d
```

IDE Integration

PyCharm Configuration:

1. Go to Run → Edit Configurations
2. Add a new Python configuration
3. Set Module name to: `BOFS`
4. Set Parameters to: `run your_config.toml -d`
5. Set Working directory to your project path

VS Code Configuration:

Add to your `.vscode/launch.json`:

```
{
    "version": "0.2.0",
    "configurations": [
        {
            "name": "BOFS Debug",
            "type": "python",
            "request": "launch",
            "module": "BOFS",
            "args": ["run", "config.toml", "-d"],
            "cwd": "${workspaceFolder}",
            "console": "integratedTerminal"
        }
    ]
}
```

5.8.6 Port Configuration

By default, BOFS runs on port 5000. You can specify a different port in your configuration file:

```
PORT = 8080
```

The server will now be accessible at `http://localhost:8080` (or your specified port).

5.8.7 Troubleshooting

Common Issues

Command not found:

If you get a "command not found" error, ensure BOFS is properly installed:

```
pip install bride-of-frankensystem
```

Permission denied:

On some systems, you may need to use `python -m BOFS` instead of the `BOFS` command directly.

Config file not found:

Check that the file exists in your current directory (`dir` on Windows, `ls` on Mac/Linux) and that you are running the command from the folder containing it:

```
BOFS run ./config.toml  # Use relative path
```

Port already in use:

If port 5000 is already in use, either:

- Stop the other service using port 5000
- Change the `PORT` setting in your configuration file
- Kill any existing BOFS processes: `pkill -f BOFS`

Debugging Tips

Enable verbose output:

Use debug mode to see detailed error messages:

```
BOFS run config.toml -d
```

Check configuration:

Verify your TOML file is valid and contains required settings. Common required settings include:

- `SQLALCHEMY_DATABASE_URI`
- `PAGE_LIST`

Database issues:

If you encounter database errors, check:

- Database file permissions (for SQLite)
- Database connection settings
- Whether the database file exists and is writable

5.8.8 Exit Codes

BOFS uses standard exit codes:

- `0`: Successful execution
- `1`: General error (configuration issues, startup failures)
- `2`: Command line argument errors

5.8.9 Environment Variables

BOFS respects standard Flask environment variables:

- `FLASK_ENV`: Set to `development` for enhanced debugging
- `FLASK_DEBUG`: Set to `1` to enable debug mode (alternative to `-d` flag)

Note that command-line flags take precedence over environment variables.

5.8.10 See Also

- *Install BOFS* - Install BOFS
- *Initialize Your First Project* - Create your first project
- *Configuration Reference* - Configuration reference
- *Monitoring and Exporting Data* - Admin panel documentation

5.9 Helper Functions

Functions from the `BOFS.util` package, available to custom blueprint code. For how blueprints fit into a project, see *Blueprints and Routes*.

`BOFS.util.`**`verify_correct_page`**(*f*)

A decorator to be used on routes/views, which checks the the user is on the correct page. Checks `session['currentUrl']`. If the user is on the wrong page, they will be redirected to the correct page.

> **Note**
>
> - Should be used just on routes after the user's session is created (usually after the consent form).

`BOFS.util.`**`verify_session_valid`**(*f*)

A decorator to be used on routes/views, which checks for the existence of the 'currentUrl' key in `session`.

> **Note**
>
> - Should be used just on routes after the session is created (for example, after the initial questionnaire).
> - The `/submit` path is where the value of `session['currentUrl']` is set.

`BOFS.util.`**`redirect_and_set_next_path`**(*current_path=None*)

Uses the next_path_in_list() method to redirect the user

Parameters
`current_path` (*str*) – The user's current path

Returns
str – the next path in PAGE_LIST which the user should be sent to.

`BOFS.util.`**`redirect_next_page`**()

Like redirect_and_set_next_path but uses the Flask request variable instead. Needs a valid app context to work. Redirects user to the next page based on their present URL (from the request).

BOFS.util.**create_breadcrumbs**()

An optional function, the result of which can be passed to templates which extend the base `template.html` file. Pages with the same name will be represented as Page Name (3) or **Page Name (2 of 3)** when the user is on that particular page.

Returns
A list of "breadcrumbs", each of which are a dictionary with a human-readable name for the path, and whether or not that page is the active page, meaning it should be made bold.

BOFS.util.**fetch_attr**(*obj*, *attribute*, **args*) → Any

Returns attribute value, or calls a method. Can handle attributes nested with dots (.)

This is similar to Python's built in [https://docs.python.org/2/library/functions.html#getattr getattr()], but with support for an arbitrary depth.

BOFS.util.**fetch_current_condition**() → int | None

Returns
The participant's current condition, as an integer (or None)

BOFS.util.**fetch_condition_count**() → int

BOFS.util.**fetch_condition_count_db**()

This is useful only after data has been collected. :return:

BOFS.util.**provide_consent**(*assignCondition=True*, *logDisplaySize=False*)

This needs to be used inside a route, otherwise session and request won't work.

Parameters

- **assignCondition**
- **logDisplaySize**

Returns
A Participant object

BOFS.util.**float_or_0**(*value*) → float

Cast the value to a float if it can, otherwise return 0.0.

Parameters
value – value to be cast

Returns
the value as a float (or 0.0)

BOFS.util.**int_or_0**(*value*) → int

Cast the value to an int if it can, otherwise return 0.

Parameters
value – value to be cast

Returns
the value as a int (or 0)

CHAPTER

SIX

CHANGELOG

Notable changes to BOFS by release. Backward compatibility is a design constraint — existing configs, questionnaires, and database schemas continue to work across upgrades; see the migration notes when they don't.

6.1 Changes from 2.0 to 2.1 (unreleased)

Getting Started

- New `BOFS init` CLI wizard for creating a new BOFS project; can launch the project with the browser opened automatically.
- `SECRET_KEY` is no longer required in the config; one is auto-generated and stored when the database is initialized.
- Major overhaul of the documentation, with separation sections aimed at different audiences ("Building Your Experiment", for researchers who want to use the built-in features, and "Understanding the Framework" for those who want to go further and extend the functionality of BOFS).
- New `CHECK_FOR_UPDATES` config option (default `true`): BOFS checks PyPI at startup and periodically while the admin panel is open, surfacing an admin notice when a newer `bride-of-frankensystem` (or `-dev`) release is available. Set to `false` for air-gapped servers.

New Question Types and Questionnaire Features

- Added a "video" question type for embedding video stimuli.
- Added an "image_select" question type for choosing one option from a set of images. Supports horizontal (default) or vertical centered layouts, a per-question `width`, and an `auto_resize` flag that normalizes images to the smallest natural dimension in the group. The stored column's data type is auto-detected (integer/float/string) from the image values, and the `images` list is validated at startup.
- Added an "image_click" question type for selecting coordinates on an image.
- Added a "group" question type for visually grouping a set of questions together; respects the `horizontal` layout flag.
- Checklist now supports options with associated text fields.
- Radiolist now supports an "other" option with a free-text field, and can require a specific option to be chosen to continue.
- Radiogrid can now be made optional, supports `questions` as an alias for `q_text`, supports an optional "not applicable" column, can store the chosen column's label string instead of its index via `store_labels: true`, and gives clearer validation feedback.
- Any text input can block paste and drag-drop via a per-question `disable_paste` property, and pasting/dropping can be disabled study-wide with the `DISABLE_PASTE` config option (individual questions can still opt in when it is `false`). Blocked pastes are recorded as a distinct event in the interaction log.

- Questions can define a `title` attribute that displays above the question box.
- Questions can be configured to show only for specific conditions.
- Questionnaire JSON accepts `question_type` as an alias for `questiontype`.
- The questionnaire parser now coerces submitted values to each column's declared type. (Previously this worked implicitly only for types whose value was rendered back through an HTML template.)
- Questionnaire questions are now loaded server-side instead of via htmx.
- Questionnaire validation issues are reported on the console, and the user is directed to a friendlier error page.
- If a participant returns to a prior questionnaire (supported internally, not exposed in the UI), their previously-entered values are reloaded into the form. (Will support an eventual/possible "back" button.)
- New interaction event log (stored in the `bofs_interaction_log` table) captures focus/blur/change/paste/drop/visibility events for every input on every questionnaire (replacing the radiogrid-only click log). Text inputs also record per-field authenticity signals — keystrokes, backspaces, pastes, pasted character count, drops, dropped character count, final length, total focus duration, and time-to-first-keystroke. Blocked pastes (see `DISABLE_PASTE`) are logged as their own event.

Page Flow and Ending the Study

- Added an end-reason system: `/end/<reason>` records *why* a participant's session ended. BOFS auto-logs its own built-in reasons (`bot`, `quota_full`, `session_loaded`, and others), and end-reason counts appear on the admin progress page (shown only when more than one reason is present).
- A `PAGE_LIST` entry for the `end` path can set an `outgoing_url`, so participants can be redirected to different destinations depending on which ending they reach.
- Reason-specific end screens can be added as `templates/end/<reason>.html`; a matching template is shown when no `outgoing_url` is set for that reason.
- The same path may now appear multiple times in `PAGE_LIST`. Occurrences are tracked individually (so admin progress and routing no longer break on repeats), and questionnaires re-used across the page list are auto-tagged. (`redirect_to_page` / `redirect_from_page` still target a path reliably only when it appears once.)
- BOFS internal routes are now rejected if placed in a project's `PAGE_LIST`, with a setup-diagnostic message explaining the conflict.

Conditions

- Can optionally load conditions from a `.csv` file or an existing `.db` file, making longitudinal studies easier to set up.
- Individual conditions can be "paused" to stop new participants from being assigned to them.
- If all conditions are paused, new participants are blocked from entering the study.

External IDs, Sources, and Recruitment

- The `mTurkID` database column was renamed to `externalID` (the old name is kept as a SQLAlchemy synonym, so existing code and queries keep working). The in-place column-migration helper now supports SQLite, PostgreSQL, MySQL, and MariaDB.
- External IDs captured from URL parameters (`PROLIFIC_PID`, `external_id`) are now saved to the participant record right after consent instead of only living in the session, making `/external_id` an optional manual-entry fallback. The canonical external-ID session key is now `externalID`, with `session['mTurkID']` kept in sync for backward compatibility.
- New `Participant.source` column captures the recruitment channel from `?source=<value>` (inferred as `prolific` when `PROLIFIC_PID` is present without an explicit source). Because it is a column on `Participant`,

it is automatically usable in expressions (e.g. `show_if = "source == 'prolific'"`). It appears in the CSV export and on the admin progress page (the column is hidden when every participant shares the same value).

Databases

- A study can now define multiple databases via `SQLALCHEMY_BINDS`, and each questionnaire or table can specify which database it uses. Exports, the admin database viewer, database download/delete, and in-place schema migration all operate per-bind, and a startup warning flags orphaned rows left in a bind's tables.

Setup Diagnostics and Validation

- All startup errors, warnings, and informational notices are now collected in one place. Fatal configuration errors block the study behind a dedicated error page; non-fatal warnings and notices are surfaced as pills in the admin navbar, and any pending warnings are shown as an interstitial first page of the study (visible only to the researcher, in debug mode or via localhost).
- Expanded startup validation: cross-questionnaire `show_if` references are checked, image assets referenced by image-based questions are verified to exist on disk, and empty questionnaires/tables now produce warnings.

Admin Panel

- New procedure diagram visualizing the page flow.
- Can preview an individual participant's responses.
- Admin panel redirects users back to the log-in page when their session is no longer valid.
- Refined admin control panel shown at the bottom of pages when logged in as admin or running in debug mode.
- Export button on the database table view moved to the top of the page.
- Large database tables now render with a paginated/scrollable display instead of dumping the whole table.
- The results preview now shows a histogram alongside the boxplot for numeric fields.
- Constrained admin navbar dropdown heights so long menus no longer run off-screen.

Debug Mode and Developer Experience

- In debug mode, a condition picker lets developers choose which condition to test.
- In debug mode (or via localhost), users can restart the experiment from `/end`; the restart route keeps the admin login session intact.
- The debug toolbar is now shown on pages before the participant record exists, but with reduced functionality.
- A console warning is shown when `@verify_correct_page` is missing on a custom route.

Custom Routes and Templates

- Activity polling is now injected via `after_request`, so participants on long-running tasks with fully custom HTML are no longer falsely marked as abandoned. Pages are also marked "submitted" when the session's current URL is updated.
- Participant progress tracking moved out of `@verify_correct_page` into a `before_request_` handler, so custom routes that omit the decorator still track progress properly.
- Template's header colour can be set from the config file, and is now also subtly applied to questionnaire form elements.
- Alternative `style.css` files can be loaded via configuration.
- `APPLICATION_ROOT` can now be omitted from the config.
- Removed the Bootstrap CSS dependency from the main template; all built-in CSS classes were renamed to no longer collide with Bootstrap. Questionnaire-related styling is now isolated in `style_questionnaire.css`.

- Reworked checklist, radiolist, and radiogrid markup so the clickable area is more consistent across them, and moved handling of the `horizontal` layout flag into the questionnaire macro so it applies to the new group question type as well.

Expression Engine and Show-If

- Added an expression engine for `participant_calculations`, replacing the use of `eval()`.
- Added `show_if` predicates on questionnaire questions (live, browser-side, against same-page form values) and on entries in `PAGE_LIST` (server-side, against stored answers). See the expressions page in the documentation for the syntax.
- `conditional_routing` now accepts an optional `show_if` predicate alongside (or instead of) `condition`.
- All user-facing strings in questionnaire JSON (top-level `title`/`instructions`, per-question `title`/`instructions`/`text`/`placeholder`/`left`/`right`/`labels`/`q_text[].text`, etc.) are scanned for `{{ expression }}` placeholders at render time. Each placeholder is evaluated as a BOFS expression against the current participant and substituted (HTML-escaped) into the rendered HTML.

Participant Data Access in Templates and Custom Routes

- `participant` is now available as a template variable in every BOFS template; resolves to `None` when no participant is in session.
- New `participant.evaluate(expression)` runs a BOFS expression against the participant's stored data. The same syntax used in `show_if` and `participant_calculations` is now usable from templates and custom blueprint code.
- New `participant.has_questionnaire(name, tag='')` distinguishes a real submission from the blank-default row that `participant.questionnaire(name)` returns when nothing has been submitted yet.
- `participant.table(name)` now returns a `TableAccessor` exposing the participant's raw rows (`accessor.rows`), every export aggregate as an attribute, and all aggregates as a dict (`accessor.exports`). Scalar exports resolve to a single value; `group_by` exports resolve to a dict keyed by the group value (or by a tuple when `group_by` is a list of columns).

Security and Hardening

- Added IP-based brute-force protection on the admin login.
- Admin sessions are bound to the IP they were issued for. Participant sessions are bound by default and can be opted out via `SESSION_BIND_TO_IP_PARTICIPANT = false`.
- New `BEHIND_REVERSE_PROXY` config (default `false`). When `true`, BOFS reads the real client IP from `X-Forwarded-For` via Werkzeug's `ProxyFix`; otherwise `X-Real-IP` and `X-Forwarded-For` are ignored. Enable this when running behind Caddy or nginx, or IP-based protection sees every request as coming from the proxy.
- Added a timing trap and tightened the consent honeypot to better filter out bots at consent.
- Reloading a session via external ID no longer restores a prior admin-login flag, preventing accidental admin access after session recovery.
- Researcher-authored HTML in questionnaire JSON (`instructions`, `title`, `text`, `left`/`right`, sub-question text for grids and checklists) is run through a `bleach`-backed allowlist sanitizer at JSON load. Permitted tags cover text formatting, lists, tables, links, images, and `<video>`/`<audio>`/`<source>`/`<track>`; protocols are restricted to `http`, `https`, and `mailto`.
- CSRF protection is enforced across the admin blueprint via a `before_request` hook backed by `flask_wtf`.
- `csv.writer`-based serialisation.
- Admin password comparison uses `hmac.compare_digest` at both the login and the database-delete confirmation.

- `route_database_download` resolves the SQLite URI relative to `current_app.root_path` and refuses paths that escape it.

Internal Refactoring

- Introduced a service layer consolidating previously scattered logic: `ParticipantService`, `SessionRecoveryService`, `ParticipantRoutingService`, `ParticipantQuestionnaireService`, and `AdminStatsService`.
- `Results` class moved to `services/data_export.py`; fixed an export-flags issue discovered during the move.
- Each question type now has a machine-readable schema generated from a parsable Google-style docstring (`BOFS/question_schemas.py`), giving a single source of truth for the properties each question type accepts.
- Added unit tests covering JSON tables, built-in model classes, results calculations, session management, page routing, participant management, and questionnaire handling.
- Moved startup-related functions into a dedicated `startup.py` and added a `setup_diagnostics.py` module.
- Performance work: cached page lists and replaced a per-row loop with a JOIN when exporting interaction logs; reduced memory usage.
- Added a GitHub Actions CI workflow that runs the test suite, and a CONTRIBUTING.md.

Bug Fixes

- Fixed `redirect_next_page` edge cases.
- Fixed condition being lost when external ID was loaded via URL arguments.
- Fixed debug condition picker showing on projects with no conditions.
- Fixed slider default size not applying to the right label.
- Fixed page-list composition issue in the admin progress route.
- Fixed the export page ignoring the "show unfinished" and "show excluded" checkboxes.
- Fallback source for standard deviation/variance now uses the population method.
- Fixed `EXTERNAL_ID_PROMPT` being ignored except as failure feedback — the configured prompt text is now used on `/external_id`.
- Fixed the Unity WebGL loading spinner never disappearing — the spinner is now hidden when the Unity instance finishes loading (Unity's progress callback stops at 0.9), and a load failure now shows an error message instead of hanging on the spinner.
- Fixed an exception when previewing results in the admin panel.
- Fixed admin-panel buttons that had become invisible.
- Fixed a condition being assigned twice when `assign_condition` is the first entry in `PAGE_LIST`.
- Fixed interaction logging breaking on free-text inputs, and redirect to the start page when a content page is reached with no participant in session.

6.2 Changes from 1.2 to 2.0 (2024-05-10)

See the migration guide for upgrading projects from 1.x.

Core Changes

- Removed support for Python 2.7.x
- Changes to support the latest Flask version and latest SQLAlchemy.

- New documentation.
- Updated all JS libraries.
- Consent text is now read in from a `consent.html` file next to the configuration file instead of from the configuration file.
- Added "BOFS" command to start projects instead of needing a separate `run.py` file.
- No more need for the "app" directory to be a part of the project.
- Use `flask-compress` to serve brotli and gz files (e.g., for Unity WebGL builds).
- Improved auto-reload feature that now works with templates.
- Export code was rewritten to minimize the chance of throwing exceptions.
- Simple error messages will now show up if in production mode, rather than a generic 500 error.
- If not in debug mode, save any 500 errors to a log file.
- When a project starts, check for missing DB columns from tables and questionnaires and add them (only works with SQLite).
- Detect web scrapers and if found exclude them from the condition counts and progress page.
- Many bug fixes.

Project Configuration Options

- Can now specify a specific completion code, or redirect users to an external URL upon reaching `/end`.
- `/start_mturk` route is now `/external_id`.
- Can now configure what the prompt on `/external_id` says.
- If `PROLIFIC_PID` is passed in as a URL parameter, the session variable is set and the input field on `/external_id` is automatically populated with the value passed.
- Conditions are now defined with labels and can be disabled selectively; labels show on progress page and are used in the export instead of the numbers.
- Can now configure `ABANDONED_MINUTES` and `COUNTS_INCLUDE_ABANDONED`, relating to ignoring participants who abandoned the task when considering participant counts within conditions.

Administration

- Can now download SQLite database from the admin panel as well as clear it out.
- Admin panel styling changes.
- Admin login now redirects users to where they were trying to go instead of always redirecting to the progress page.
- Added the ability to selectively indicate individual participants to exclude from the participant counts.
- Added a results page to the admin panel to view summary statistics of any numeric data.

Tables, Questionnaires, and Blueprints

- Can define custom exports within the JSON-defined tables.
- Can now use 'HAVING' clauses in custom exports.
- Can now configure the default value of sliders.
- Some slight improvements to how the questionnaires are rendered on mobile devices.
- `/table` POST route now understands JSON data.
- Blueprints no longer need a `views.py` file to be included.

- Can now have "simple" HTML pages, that support templates but can be shown without any additional Python code.
- It is now possible to define custom question types to use within questionnaires.
- Can put questionnaires and tables inside of a blueprint's folders.

6.3 Changes from 1.1 to 1.2 (2022-12-27)

This is the last release of the 1.x version. Later releases do not support Python 2.7.x.

Changes and Enhancements

- Flag some parts of the questionnaire JSON as safe (so HTML can be added to those fields).
- Custom export feature now works without grouping or ordering variables.
- Can now group by multiple columns on custom exports.
- Added a simple honeypot to the consent page.
- Added a progress summary to the top of the admin panel's progress page.
- When assigning conditions to participants, omit "abandoned" attempts from the calculation of participant counts in each condition.
- Added in the ability to load in participant progress based on their MTurk ID (via the `RETRIEVE_SESSIONS` config option, set it to `True`).
- Removed the IP address check on `verify_session_valid`.
- Made changes to work with the 2021.1.x version of Unity (which uses a different type of compression).
- Added a method of defining database tables via a JSON file instead of in the `models.py` file.

Fixes

- Fixed variance fallback method.
- Fixed bug where the last condition's questionnaires weren't found.
- Possible fix for participants getting redirected back to the `start_mturk` page.

6.4 Changes from 1.0 to 1.1 (2020-04-03)

Features Added

- Session information from previous attempts can now be loaded in after entering the MTurkID.
- Questionnaire calculation strings that can be embedded into the .json files and are exported by `/admin/export` (e.g., for subscales).

General Improvements

- `condition` field is now nullable.
- Added config option to allow/restrict re-takes (multiple attempts).
- Added `crumbs` variable to be added to templates by default.
- Pages with blank names no longer show in the breadcrumb list.
- Admin panel: if `logGridClicks` is `False`, don't show the dropdown.
- Server-side sessions are now provided by a custom interface, rather than flask-sessionstore.
- Removed dependency on Flask-SocketIO (while allowing eventlet to stay).

- Slight change to how blueprints are imported.
- Embedded JS libraries into the project rather than linking to external URLs.
- Additional admin pages can now be defined directly in blueprints, rather than just in the config file.
- Logging of grid click data is now working again.
- Added check on `/current_url` route to avoid exceptions for people who have no session data.
- Fixed previewing questionnaires not loaded in the config.
- Now reports the URL when running with `self.debug = False` (on launch).
- Fixed a bug with condition assignment.
- Added basic `/submit` route that redirects the user on form submission.
- Blank logo (instead of the USask logo).
- Updated README.md and setup.py.

6.5 1.0.0 — Initial release (2019-11-17)

First tagged release.

CHAPTER

SEVEN

WHERE TO START

- Never used BOFS? — *What is BOFS?* then *Install BOFS*.
- Already installed? — *Initialize Your First Project* walks through `BOFS init` and what it generates.
- Want a JavaScript task? — *Quickstart: Integrating a Custom Task*.

CHAPTER

EIGHT

COMMON GOALS

The pages above follow a linear path. If you already know what your study needs, this table maps common study designs and needs to the page that covers them.

I want to...	Covered in
Compare two or more versions of something (a between-subjects A/B design)	*Conditions and Branching*
Have participants return for a second session (multi-day, pre/post, diary)	*Longitudinal Experiments*
Give the same questionnaire twice in one session (pre-test and post-test)	*Including the same questionnaire multiple times* in *Setting Up Your Page Flow*
Screen out participants who don't qualify	*Conditions and Branching* (question-based routing) and *Multiple end pages* in *Setting Up Your Page Flow*
Embed a game or custom task (Unity, jsPsych, p5.js, custom JavaScript)	*Quickstart: Integrating a Custom Task*, then *Adding Your Own Pages*
Record trial-by-trial data from a task	*Storing Custom Data*
Show or hide a question based on an earlier answer	`show_if` in *Adding Survey Questions*
Compute scores from responses (scale totals, subscales)	`participant_calculations` in *Questionnaire Properties*
Watch participants' progress and export the data	*Monitoring and Exporting Data*
Detect bots and low-quality responses	*Data Quality*
Test everything before recruiting real participants	*Testing Your Study Before Launch*
Change the colors and styling participants see	*Customizing the Appearance*
Recruit through Prolific or MTurk	*Recruiting via MTurk or Prolific*

Note

Source code and issue tracker: github.com/colbyj/bride-of-frankensystem.

Example projects: github.com/colbyj/bride-of-frankensystem-examples.

If you need additional help, please join us on Discord.

CHAPTER

NINE

CITATION

If you use BOFS in your research, please cite it via the following identifiers:

- **DOI:** 10.5281/zenodo.11176739
- **RRID:** SCR_028428

Colby Johanson. (2024). colbyj/bride-of-frankensystem: 2.0 (2.0). Zenodo. https://doi.org/10.5281/zenodo.11176739

BibTeX:

```
@software{bride-of-frankensystem,
  author       = {Colby Johanson},
  title        = {colbyj/bride-of-frankensystem},
  month        = may,
  year         = 2024,
  publisher    = {Zenodo},
  version      = {2.0},
  doi          = {10.5281/zenodo.11176739},
  url          = {https://doi.org/10.5281/zenodo.11176739}
}
```