Bride of Frankensystem

Colby Johanson

May 10, 2024

INTRODUCTION:

CHAPTER

ONE

OVERVIEW

1.1 Intended Uses

Bride of Frankensystem (BOF or BOFS) is intended to be used by developers to deploy custom-developed experiments online. It provides a variety of solutions to common problems associated with online experiments while being easily extended in a way that minimizes code duplication and encourages code reuse. Its design focuses on flexibility rather than providing concrete solutions to every scenario, and so is targeted primarily towards software developers while still being accessible for non-developers for many simple use-cases.

BOF is built using Flask and is intended to be used as a Python library. Because it is built with Flask, all Flask extensions and features are supported, and it is relatively straightforward to extend the project with your own custom web pages and tasks.

If you use this for your research, please cite it!

> Colby Johanson. (2022). colbyj/bride-of-frankensystem: 1.2 (1.2). Zenodo. https://doi.org/10.5281/zenodo.7487295

1.2 Features

BOF includes a number of features relating to deploying online experiments:

- **Automatic *routing* between pages in a pre-defined order.**
  - Built-in consent page in which the text can be easily configured.
  - Automatic random assignment of participants to various conditions, so different participants can be shown different pages.
- Define *questionnaires* using a custom JSON structure.
- Show your own *simple web pages* defined via HTML (e.g., for instructions).
- Show more complex *custom web pages* (using Python and Flask) to embed custom tasks.
- Create *database tables* to record information about what your participants are doing.
- An admin panel to track participants' progress and export data.

1.3 Dependencies

BOFS requires Python 3.9+, along with the following Python packages.

- `flask` - The web framework that BOF is based off of.
- `sqlalchemy` - An object-relational manager that is used for database table definitions and query access.
- `flask-sqlalchemy` - A bridge between Flask and SQLAlchemy.
- `eventlet` - This is used as the production (live) web server, as an alternative to Flask's built in web server or the Apache web server.
- `toml` - The configuration files use the toml format.

CHAPTER

TWO

INSTALLATION INSTRUCTIONS

These instructions cover two different ways in which you can install BOFS.

2.1 Installing Within a Virtual Environment

It is highly recommended that you install BOFS in a virtual environment (venv). What this does is ensure that your project always has access to the version of BOF that it was developed with.

The steps for doing this type of install are:

1. Ensure that Python 3.9 (or newer) is installed on your machine and that `pip` is accessible via the command line.
2. Using the command line, create the venv with: `python -m venv bofs_venv`
3. Activate the venv.
    - In Windows this is done via `.\bofs_venv\Scripts\activate.bat` if using `cmd` or `.\bofs_venv\Scripts\Activate.ps1` if using Powershell (the default command line in Windows 11).
    - In MacOS or Linux this is done via `source bofs_venv/bin/activate`
4. Install BOFS via pip:
    - `pip install bride-of-frankensystem` for the latest release (recommended).
    - Or, for the latest development version, download the project source code as a zip and install it via pip: `pip install bride-of-frankensystem-master.zip`
5. Ensure that you can execute the `BOFS` command. Try it without any arguments and you should see a help message.

2.2 System-Wide Installation

An alternative approach to installing BOFS is to install it onto your system directly, so that you do not need a Python venv for your project. This is more convenient, but it could cause trouble in the future for projects that depend on the specific release of BOFS that existed at the time when the project was created (in the case of updates to BOFS).

The steps for doing this type of install are:

1. Ensure that Python 3.9 (or newer) is installed on your machine and that `pip` is accessible via the command line.
2. Using the command line, install BOFS: `pip install bride-of-frankensystem`
3. Ensure that you can execute the `BOFS` command. Try it without any arguments and you should see a help message.

That's it!

2.3 Running BOFS

Once installed via `pip`, you can run your project by executing the `BOFS <your_config_file>` command in your project directory. You can use the `-d` flag to enable debugging mode. If you installed BOFS via venv as suggested, then you will need to activate the virtual environment via step 2 of the installation instructions.

Note: If you are using PyCharm, then you can run it by adding a custom run configuration with *BOFS* as the module name and your config file as the parameter.

CHAPTER

THREE

QUICKSTART GUIDE (THE MINIMAL EXAMPLE)

To get started using Bride of Frankensystem (BOFS), the first thing you need to do is ensure that it is installed on your computer. Please follow the instructions on *Installation Instructions* and then come back to this page to continue the quickstart guide.

In this guide, we will go over a simple example project to demonstrate how to get started with a BOFS project and to demonstrate some of the features of BOFS.

The minimal example only demonstrates some of the features of BOFS, but it is a good starting point for new projects. To follow the quickstart, download the example code from here. You can also preview the code on on GitHub.

3.1 The Files

The minimal example consists of only 4 files:

File	Description
/minimal_example/questionnaires/example	An example questionnaire file demonstrating all of the question types available.
/minimal_example/consent.html	A HTML file that defines how the consent form should look.
/minimal_example/minimal.toml	A configuration file that defines many important aspects of the minimal example.

Let's go through each file to understand what each is doing.

Listing 1: example.json

```
{
    "title":"Example Questionnaire",
    "instructions":"Please rate each video game experience....",
    "code":"",
    "questions":[
        {
            "questiontype":"textview",
            "title":"I am a title",
            "text":"This allows you to display descriptive or informative text."
        },
        {
            "questiontype":"radiogrid",
            "instructions":"choose option...",
            "id":"radio_1",
```

(continues on next page)

(continued from previous page)

```
            "shuffle":"true",
            "labels":[
                "I hate it!",
                "",
                "Neutral",
                "",
                "I love it!"
            ],
            "questions":[
                {
                    "id":"q_1",
                    "text":"dfdjs;aljd"
                },
                {
                    "id":"q_2",
                    "text":"hahahahaha"
                },
                {
                    "id":"q_3",
                    "text":"muahahahaha"
                }
            ]
        },
        {
            "questiontype":"radiolist",
            "instructions":"choose one option...",
            "id":"radiolist_1",
            "labels":[
                "Yes",
                "Maybe",
                "No"
            ]
        },
        {
            "questiontype":"checklist",
            "instructions":"choose any options...",
            "id":"checklist_1",
            "shuffle":true,
            "questions":[
                {
                    "id":"cl_1",
                    "text":"Option 1"
                },
                {
                    "id":"cl_2",
                    "text":"Option 2"
                },
                {
                    "id":"cl_3",
                    "text":"Option 3"
                }
            ]
```

(continues on next page)

(continued from previous page)

```
        },
        {
            "questiontype":"slider",
            "instructions":"I am a slider",
            "id":"slider_1",
            "left":"left",
            "right":"right",
            "tick_count":5,
            "width":500
        },
        {
            "questiontype":"field",
            "instructions":"enter text",
            "placeholder":"I am a placeholder",
            "id":"input_1"
        },
        {
            "questiontype": "num_field",
            "instructions": "enter a number",
            "id": "number_field"
        },
        {
            "id": "dropdownexample",
            "questiontype":"drop_down",
            "instructions":"choose one",
            "items":[
                "apples",
                "oranges",
                "watermelon"
            ]
        },
        {
            "questiontype":"multi_field",
            "id":"big",
            "placeholder":"I am holding the place",
            "instructions":"big text field",
            "height":"100"
        }
    ]
}
```

This file describes a questionnaire that will be used within the study. This is the same example questionnaire as is shown on *Questionnaires*. It includes 9 different question types:

- A text view
- A radio grid
- A radio list
- A check list
- A slider
- A text entry field

- A number entry field
- A drop down menu
- A multi-line text entry field

Listing 2: consent.html

```
Your consent html can go here.
```

This file demonstrates how consent text is defined within a project.

Listing 3: minimal.toml

```
# Database settings
SQLALCHEMY_DATABASE_URI = 'sqlite:///minimal.db'

# The secret key MUST be changed to something unique.
# You should at the very least mash your keyboard a bit to generate a random string.
SECRET_KEY = 'You Must Change This to Something Unique'

# --------------------
# Application Settings
# --------------------
APPLICATION_ROOT = ''       # Rarely adjusted, used to set the project to be accessible␣
↪at a different URL rather than /.
TITLE = 'Example Project'   # What users see at the top of the page
ADMIN_PASSWORD = 'example'  # Used to log in to the admin pages at /admin
USE_BREADCRUMBS = false     # Show breadcrumbs-style progress bar
PORT = 5001                 # Configure what port the project will be accessible at
RETRIEVE_SESSIONS = true    # If ID entered at /external_id was already used, then␣
↪attempt to load a participant's progress from the database and redirect them to where␣
↪they last were.
ALLOW_RETAKES = true        # With the external_id page in use, setting this to true␣
↪will prevent the same ID from being used twice.
LOG_GRID_CLICKS = false     # Used for more fine-grained logging of participant's␣
↪progress through questionnaires. Log the time the each radio button in a radio grid is␣
↪clicked.
CONDITIONS = []             # Leave blank for only 1 condition. Format for multiple␣
↪conditions is shown below
#CONDITIONS = [{label='condition 1', enabled=true}, {label='condition 2', enabled=true}]

# External ID Settings, adjust these to adjust the phrasing on /external_id for Prolific,
↪ MTurk, etc.
EXTERNAL_ID_LABEL = "Mechanical Turk Worker ID"
EXTERNAL_ID_PROMPT = "Please enter your MTurk Worker ID. You can find this on your MTurk␣
↪dashboard."

# ----------------------------
# Completion Codes and end route
# ----------------------------
#STATIC_COMPLETION_CODE = "        # Set this if you want all participants to be given␣
↪the same completion code at the end of the survey.
GENERATE_COMPLETION_CODE = true     # Generate a random completion code for the user.
COMPLETION_CODE_MESSAGE = 'Please copy and paste this code into the MTurk form:'
```

(continues on next page)

(continued from previous page)

```
# OUTGOING_URL = "                # On the /end route, participants can be optionally␣
↪redirected to an external page instead of being given a code

# ---------
# Page List
# ---------
# Defines the pages that the user will see and their order.
# Each entry must contain a user-friendly name that is displayed to the user and a␣
↪unique path.
# For questionnaires, you can append a /<tag> to the end if you need to include the same␣
↪questionanire twice.
# If USE_BREADCRUMBS is true, then the pages with identical names will have (#) beside␣
↪them.
PAGE_LIST = [
    {name='Consent', path='consent'},
    {name='External ID', path='external_id'},
    {name='Questionnaire', path='questionnaire/example/first'},
    {name='Questionnaire', path='questionnaire/example/second'},
    {name='End', path='end'}
]

#
```

This file is the configuration of the study. It is where all of the important aspects of the study are defined. This quickstart only covers some of the options – for a complete description of every option, see *Project Configuration*.

Let's go over some of the variables in the configuration:

- `SQLALCHEMY_DATABASE_URI` defines the filename of the database. The database gets created when the project is run.
- `SECRET_KEY` is a string that should be unique to each project. It is used to secure the session cookies.
- `TITLE` defines what shows up on the web browser's tab as well as the text at the top of each page in the survey.
- `ADMIN_PASSWORD` defines the password to access the admin panel at `/admin`
- `PORT` defines the port that the survey will be accessible at once it is running (e.g., `http://<your ip address>:5001`) if the port is `5001`.
- `PAGE_LIST` defines the pages that will be included within the survey as well as the order in which they appear.

The page list is the most important variable for determining the behavior of your survey. It is a list of pages where each page is defined as a dictionary with two keys (`name` and `path`). For example, with `{name='Consent', path='consent'}` as the first page in the list, the first page that a user will see is the consent page (at `/consent`), where they are shown the consent form (as defined in `consent.html`). After this page, the next page they see is the `/external_id` page, which in this case asks the user for their Amazon Mechanical Turk ID (if you were looking through the configuration file, you'll have seen that this is something you can configure).

In general, pages are given a `name`, which is what shows up on the title of the page (i.e., the name shown in the tab on the web browser), and a `path`, which is the url that the user will be taken to (without the preceding `/`).

There are many paths built into BOFS, for display things like consent forms, instruction pages, and questionnaires. See *Built-in Routes* for a list of valid, built-in routes. For a more general overview of routing, see *Routing Participants*.

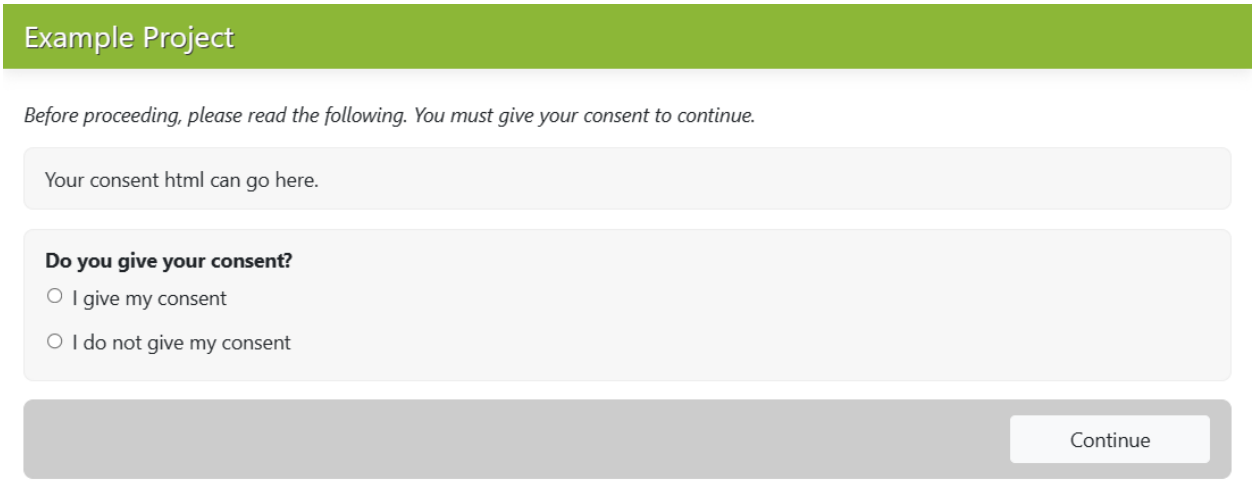
3.2 Running the Example

Let's now run the example by opening up the command line/terminal and executing `BOFS minimal.toml`. You should see several messages show up.

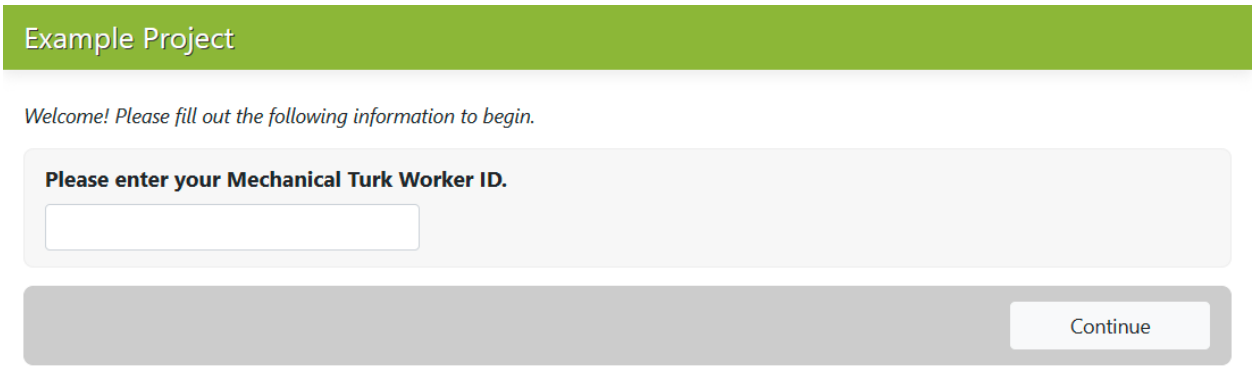
```
Loading blueprint: BOFS.admin
BOFS.admin: `models.py` not found. Add a `models.py` file to your blueprint folder use␣
↪this feature.
Loading blueprint: BOFS.default
BOFS.default: Loaded <class 'BOFS.default.models.create.<locals>.Participant'>
BOFS.default: Loaded <class 'BOFS.default.models.create.<locals>.Progress'>
BOFS.default: Loaded <class 'BOFS.default.models.create.<locals>.RadioGridLog'>
BOFS.default: Loaded <class 'BOFS.default.models.create.<locals>.Display'>
BOFS.default: Loaded <class 'BOFS.default.models.create.<locals>.SessionStore'>
BOFS.default: `models.py` loaded!
example
Listening on http://0.0.0.0:5001
Preview locally at http://127.0.0.1:5001
```

The last line of output shows you what URL your survey is available at for previewing it on your local machine. Open that URL in your web browser and let's go through each page.

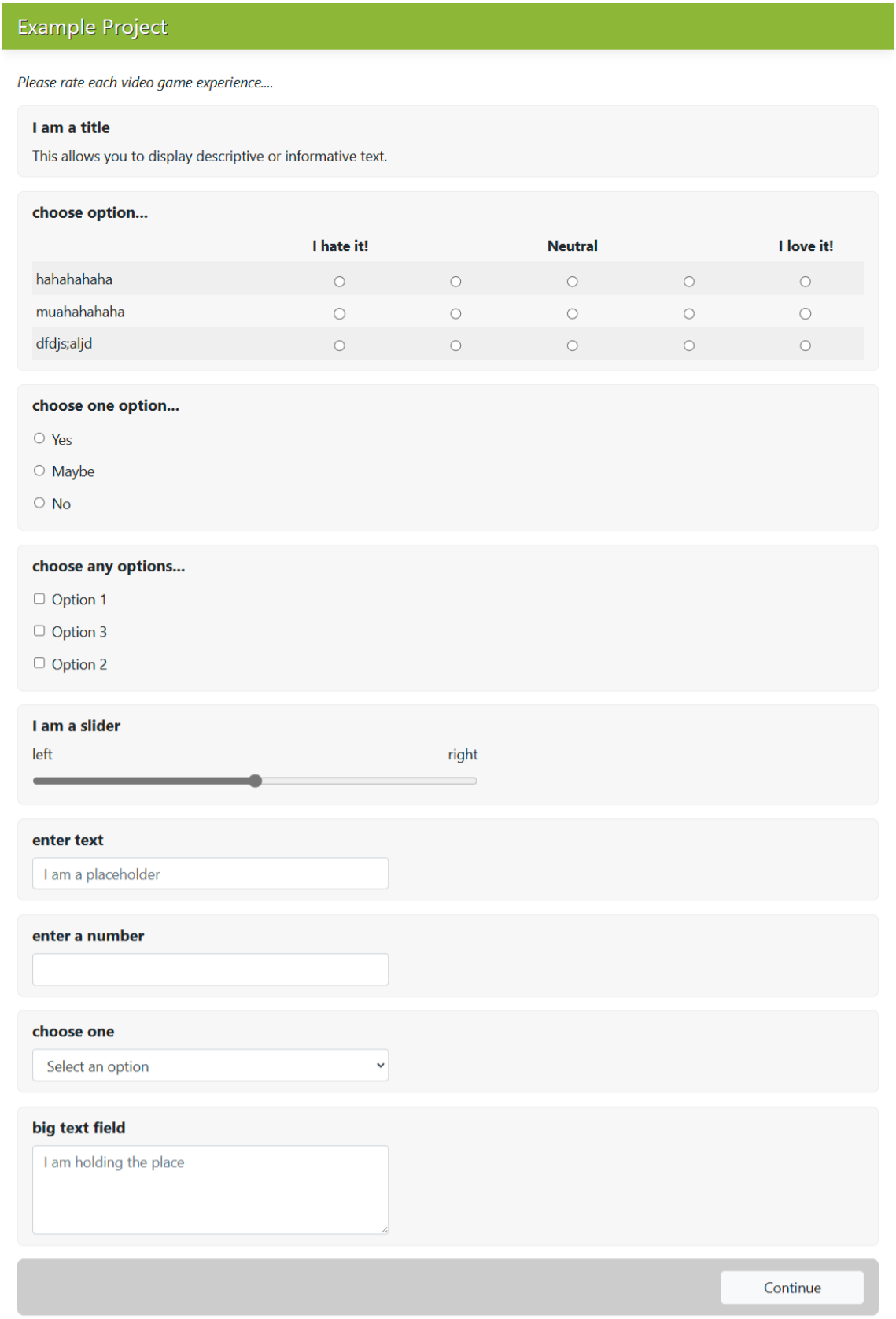
Tip: If you add the `-d` flag to your BOFS command, you will enable debugging mode. This provides more feedback if something goes wrong and also shows navigation and debugging information at the bottom of pages. For example, `BOFS minimal.toml -d`.



The first page that shows up is the consent page (`/consent`). Click "Continue" to go to the next page.

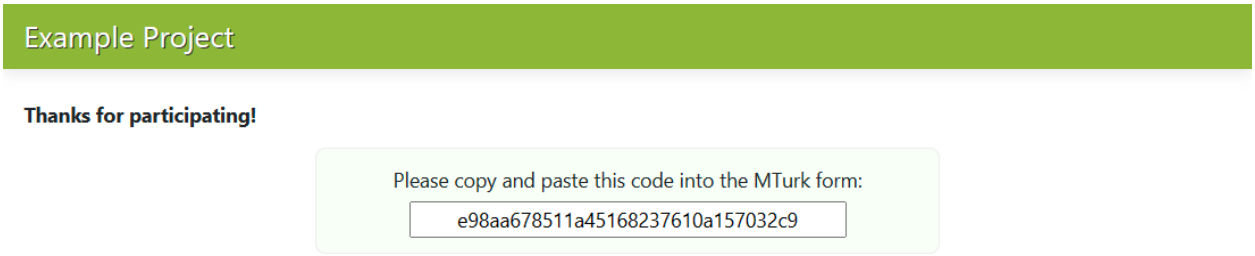


The second page is the external ID entry page (`/external_id`). The user gets prompted to enter an external ID and then needs to press "Continue" to move on to the next page.


Example Project
Please rate each video game experience....
I am a title
This allows you to display descriptive or informative text.
choose option...
I hate it!
Neutral
I love it!
hahahahaha
muahahahaha
dfdjs;aljd
choose one option...
Yes
Maybe
No
choose any options...
Option 1
Option 3
Option 2
I am a slider
left
right
enter text
I am a placeholder
enter a number
choose one
Select an option
big text field
I am holding the place
Continue

The third page is the questionnaire whose JSON code was shown earlier. You can see what each question looks like. To continue on to the next page, the user needs to provide responses to the questions and then press "Continue".

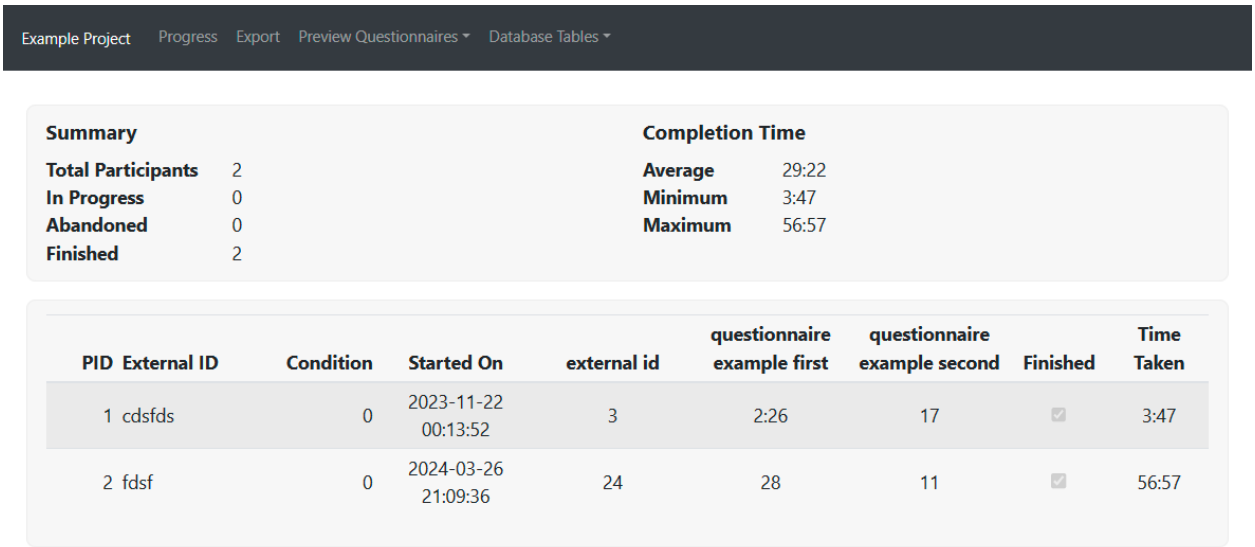
The fourth page is the same questionnaire being shown a second time.



The fifth page is the end page, where the user is shown a completion code.

3.3 The Administration Panel

Every BOFS project has a built in administration panel, accessible via the `/admin` route. So, for the minimal example, this will be accessible at `http://127.0.0.1:5001/admin`. Going to this URL should prompt the user for a password. This password is defined inside of the configuration file (`minimal.toml` for this project). For the example, the password is `example`.



Entering the password successfully will bring you to the administration panel, which shows participants' progress through the study, summary statistics relating to completion, and various other pages listed at the top. These include:

- **Progress**, the page you are looking at.
- **Export**, which allows you to export the data collected from the study in a CSV format.
- **Preview Questionnaires**, which gives you the ability to preview questionnaires used in the study, as well as any questionnaires on the file system.
- **Database Tables**, which lets you explore the underlying database and the data you've collected.

3.4 Extending the Minimal Example

Please continue reading to learn about the different ways in which this example project can be extended.

- To learn how to assign participants to different conditions, see *Assigning and Making Use of Conditions*.
- To learn more about questionnaires, see *Questionnaires*.
- To learn more about how to show static content to participants, see *Instruction Pages*.
- To learn about how to add your own custom pages, see *Adding Custom Routes (Blueprints)*.
- To learn how to store your own data to the database, see *Custom Database Tables (Models)*.

Tip: When developing BOFS projects, you may find it helpful to run the project in a new private/incognito window each time. Alternatively, you can restart your session (and go back to the start of your study) via the `/restart` route.

CHAPTER

FOUR

INTEGRATING A JAVASCRIPT TASK

The next step up in terms of project complexity from the project demonstrated in the *quickstart guide* is to integrate a simple JavaScipt task within your study. This requires the use of a *blueprint* (which further involve Jinja 2 templating and static files) and a custom database *table*.

This guide will walk you through the steps of doing this. First, let's create a new project directory called *p5_example* and populate that directory with several subdirectories.

The source code is available on GitHub.

4.1 Getting Ready

Table 1: The directories required by the project.

Directory	Description
/p5_example/my_task/	The root directory for the blueprint.
/p5_example/my_task/static/	The static folder for the blueprint. Where we will place any static project files (such as `.js` files).
/p5_example/my_task/tables/	The directory where we will put the definition for the custom database table (the `.json` file).
/p5_example/my_task/templates/	The directory where we will put any template files (the `.html` files).
/p5_example/my_task/templates/instructions/	An optional directory where we will place an instruction page (as a `.html` file).
/p5_example/my_task/templates/simple/	The directory where we will put the HTML code for our task (as a `.html` file).

In the screenshot below, you can see the complete listing of files your project will have at the end of the guide.

4.2 Python Code (`views.py`)

For more complex integrations you will need to write Python code inside of a `views.py` file, see *Adding Custom Routes (Blueprints)*. For this example you do not! We will use the features built into BOFS to handle the required logic.

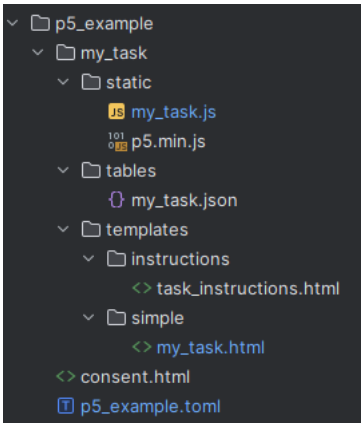


Fig. 1: The files you will have in those directories at the end of the guide.

4.3 The Task (`my_task.js`)

The following JavaScript task is written using the P5 JavaScript library. A copy of the minified P5 library itself is in the `/p5_example/my_task/static/p5.min.js` file (which you can find on the internet yourself).

Listing 1: /p5_example/my_task/static/my_task.js

```
let score = 0;

function setup() {
    createCanvas(720, 400);
}

function draw() {
    background(230);
    text('score is ' + score, 100, 100);
}

function mousePressed() {
    score += 1;
}

// Send our score after 5000 seconds
setTimeout(function () {
    let dataToSend = {
        score: score
    };

    $.post("/table/my_task", dataToSend, function () {
        window.location.href = "/redirect_next_page"
    });
}, 5000);
```

This guide will not explain how this code was written, but only point out two things. First, there is a `score` variable that increases when the canvas is clicked. Second, there is a `setTimeout()` call that performs a POST request of the `score` variable a specific BOFS route after 5 seconds, and then redirects the user.

The redirection URL (`/redirect_next_page`) is a special one that is build into BOFS. It figures out which page the user is currently on and then redirects them to the next page defined in `PAGE_LIST` (which is defined in your project's configuration `.toml`).

The URL specified for the POST request (`/table/my_task`) is for a built in route, but it requires that a table be defined as a `.json` file and placed in `/p5_example/my_task/tables/`.

4.4 The Table (`my_task.json`)

The database table is defined by following a specific `.JSON` specification, as described in *Custom Database Tables (Models)*.

Listing 2: /p5_example/my_task/tables/my_task.json

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

This file defines one column, "score", as an integer with a default value of `0`. It also defines two exports: `average_score`, which calculates the average of a user's scores via the SQL `avg` function, and `high_score`, which calculates the maximum value of the user's scores via the SQL `max` function. (As it'll turn out, a user will only ever submit one score, so the average and max yield boring results.)

Including this file in your project will add a new table called `my_task` to your database.

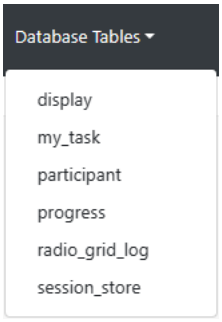


Fig. 2: A listing of the database tables now included in the project.

4.5 The View (my_task.html)

The last piece of the task is the HTML file to display the P5 JavaScript app.

Listing 3: /p5_example/my_task/templates/simple/my_task.html

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

The JavaScript code does most of the work here, and all that is needed is the <main> tag, so that P5 knows where to put the canvas. Note that this HTML will get utilized within the main project template so that the usual survey header in green and all of the styling get applied as if it's any other page.

Placing a html file under the templates/simple/ directory allows it to be accessible via the /simple/<fileName> route, which we will make use of in the configuration later on. In this case, it will be available at the /simple/my_task URL.

4.6 The Instruction Page (task_instructions.html)

There are two more small pieces to the project to add before testing. The first is an instruction page.

Listing 4: /p5_example/my_task/templates/instructions/task_instructions.html

```
<b>Click</b> as many times as you can before time runs out!
```

This will be available via the /instructions/task_instructions URL.

4.7 The Consent Text (consent.html)

And the last thing is the consent page.

Listing 5: /p5_example/consent.html

```
Your consent html can go here.
```

4.8 The Configuration File (p5_example.toml)

Let's now hook up all of these pieces together and get the example working. The important part here is to add the correct URLs to PAGE_LIST.

Listing 6: /p5_example/p5_example.toml

```
# Database settings
SQLALCHEMY_DATABASE_URI = 'sqlite:///p5_example.db'

# The secret key MUST be changed to something unique.
# You should at the very least mash your keyboard a bit to generate a random string.
SECRET_KEY = 'You Must Change This to Something Unique'
```

(continues on next page)

(continued from previous page)

```
# --------------------
# Application Settings
# --------------------
APPLICATION_ROOT = ''           # Rarely adjusted, used to set the project to be␣
↪accessible at a different URL rather than /.
TITLE = 'P5 Example Project'    # What users see at the top of the page
ADMIN_PASSWORD = 'example'      # Used to log in to the admin pages at /admin
USE_BREADCRUMBS = true         # Show breadcrumbs-style progress bar
PORT = 5002                     # Configure what port the project will be accessible at
RETRIEVE_SESSIONS = true        # If ID entered at /external_id was already used, then␣
↪attempt to load a participant's progress from the database and redirect them to where␣
↪they last were.
ALLOW_RETAKES = true            # With the external_id page in use, setting this to true␣
↪will prevent the same ID from being used twice.
LOG_GRID_CLICKS = false         # Used for more fine-grained logging of participant's␣
↪progress through questionnaires. Log the time the each radio button in a radio grid is␣
↪clicked.
CONDITIONS = []                 # Leave blank for only 1 condition. Format for multiple␣
↪conditions is shown below
#CONDITIONS = [{label='condition 1', enabled=true}, {label='condition 2', enabled=true}]

# External ID Settings, adjust these to adjust the phrasing on /external_id for Prolific,
↪ MTurk, etc.
EXTERNAL_ID_LABEL = "Mechanical Turk Worker ID"
EXTERNAL_ID_PROMPT = "Please enter your MTurk Worker ID. You can find this on your MTurk␣
↪dashboard."

# -----------------------------
# Completion Codes and end route
# -----------------------------
#STATIC_COMPLETION_CODE = "        # Set this if you want all participants to be given␣
↪the same completion code at the end of the survey.
GENERATE_COMPLETION_CODE = true     # Generate a random completion code for the user.
COMPLETION_CODE_MESSAGE = 'Please copy and paste this code into the MTurk form:'
# OUTGOING_URL = "                  # On the /end route, participants can be optionally␣
↪redirected to an external page instead of being given a code

# ---------
# Page List
# ---------
# Defines the pages that the user will see and their order.
# Each entry must contain a user-friendly name that is displayed to the user and a␣
↪unique path.
# For questionnaires, you can append a /<tag> to the end if you need to include the same␣
↪questionanire twice.
# If USE_BREADCRUMBS is true, then the pages with identical names will have (#) beside␣
↪them.
PAGE_LIST = [
    {name='Consent', path='consent'},
    {name='Instructions', path='instructions/task_instructions'},
    {name='Task', path='simple/my_task'},
```

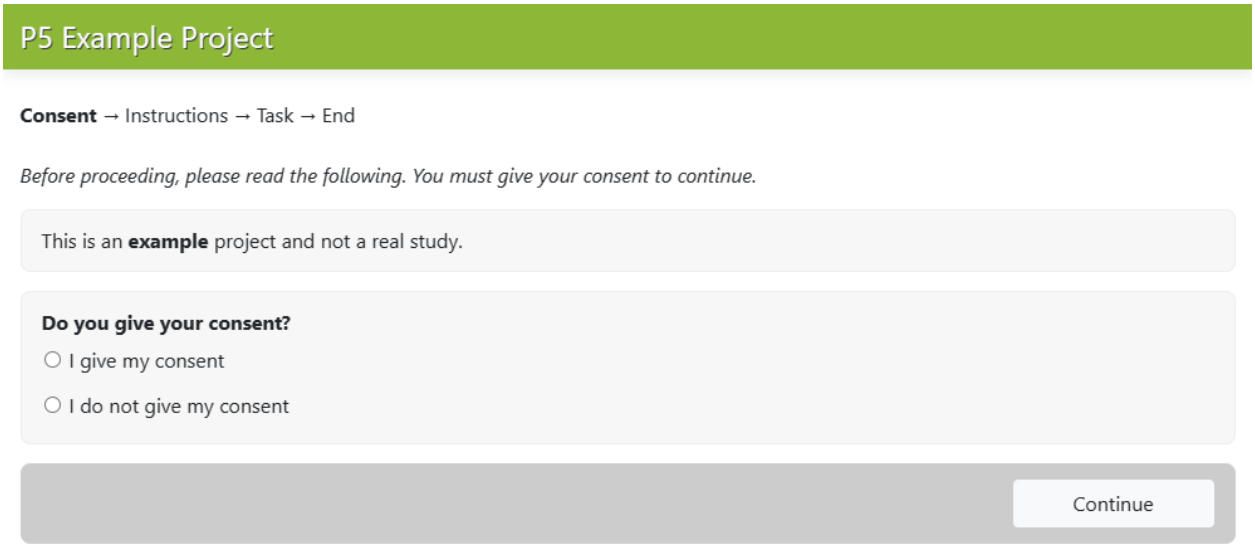
(continues on next page)

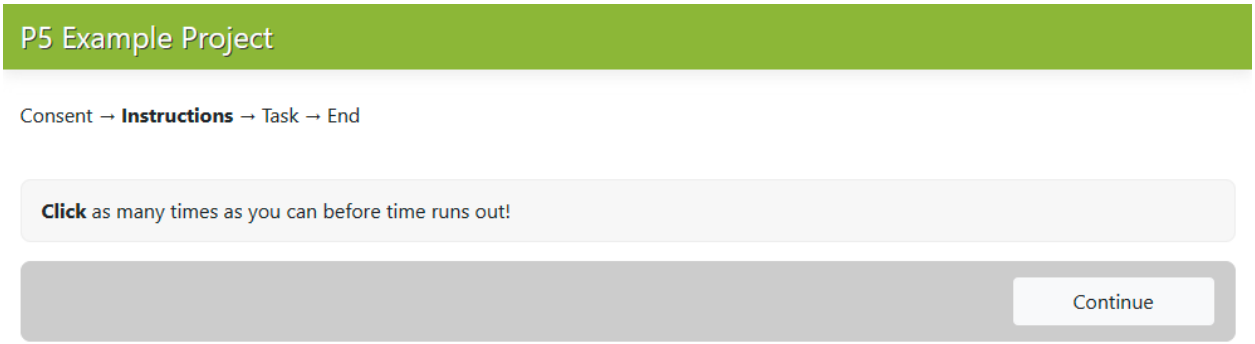
(continued from previous page)

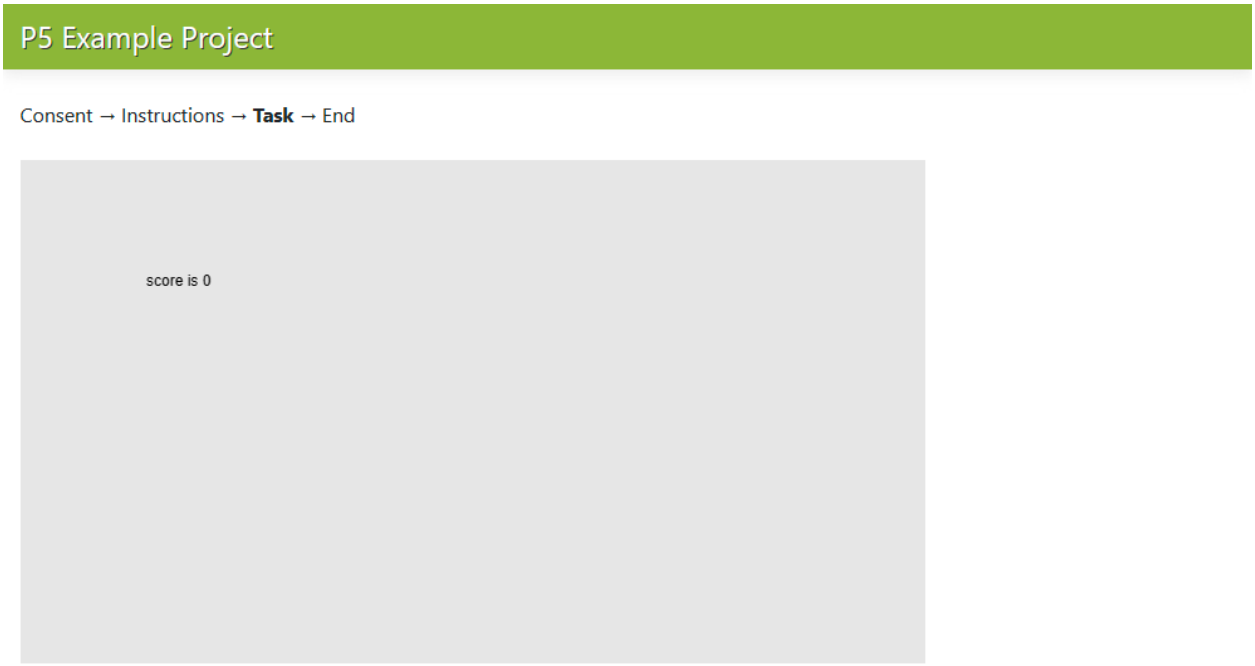
```
    {name='End', path='end'}
]
```

4.9 Walkthrough Screenshots

That's it! Let's look at what we just created.






P5 Example Project
Consent → Instructions → **Task** → End
score is 0


P5 Example Project
Consent → Instructions → Task → **End**
Thanks for participating!
Please copy and paste this code into the MTurk form:
e06b04b20ffa421e83b1a331b683d71b

CHAPTER

FIVE

PROJECT CONFIGURATION

This page includes a description of each of the options within a project's configuration file (its `.toml` file).

Table 1: Configuration variables

Variable	Data Type	Description
SQLALCHEMY_DATABASE_URI	string	This is the URI of the SQLite database.
SECRET_KEY	string	You *must* set this to a unique string. At least mash your keyboard's keys a bit.
APPLICATION_ROOT	string	Rarely adjusted, used to set the project to be accessible at a different URL rather than /.
TITLE	string	What users see at the top of the page.
ADMIN_PASSWORD	string	Used to log in to the admin pages at `/admin`.
USE_BREADCRUMBS	boolean	Show breadcrumbs-style progress bar.
PORT	integer	Configure what port the project will be accessible at.
RETRIEVE_SESSIONS	boolean	If ID entered at `/external_id` was already used, then attempt to load a participant's progress from the database and redirect them to where they last were.
ALLOW_RETAKES	boolean	With the external_id page in use, setting this to true will prevent the same ID from being used twice.
LOG_GRID_CLICKS	boolean	Used for more fine-grained logging of participant's progress through questionnaires. Log the time the each radio button in a radio grid is clicked.
CONDITIONS	list of dictionaries	See *Assigning and Making Use of Conditions*.
ABANDONED_MINUTES	integer	The minutes that a participant needs to have been away from the study before they are considered to have abandoned it.
COUNTS_INCLUDE_ABANDONED	boolean	If true, then when assigning participants to a condition, the abandoned participants will be considered as a member of the condition group.
EXTERNAL_ID_LABEL	string	For example, "Mechanical Turk Worker ID".
EXTERNAL_ID_PROMPT	string	For example, "Please enter your MTurk Worker ID. You can find this on your MTurk dashboard."
STATIC_COMPLETION_CODE	string	Set this if you want all participants to be given the same completion code at the end of the survey.
GENERATE_COMPLETION_CODE	boolean	Generate a random completion code for the user.
COMPLETION_CODE_MESSAGE	string	For example, "Please copy and paste this code into the MTurk form:".
OUTGOING_URL	string	On the `/end` route, participants can be optionally redirected to an external page instead of being given a code. Specify the URL here.
PAGE_LIST	list of dictionaries	See *Routing Participants*.

CHAPTER

SIX

ROUTING PARTICIPANTS

As explained within *Quickstart Guide (The Minimal Example)* and *Project Configuration*, the configuration file contains a `PAGE_LIST` variable that is used to configure how participants are routed through the study. There is no limit on the number of pages you can use in your experiment, but there are two pages you need to include in your project at a minimum.

The first page must be one of the following pages:

Table 1: Routes you can use for your first page

Route	Description
`/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.
`/consent_nc`	This is the same as `/consent`, except that the participant is not assigned a condition (defaults to 0).
`/create_participant`	This creates the participant entry in the database, assigns a condition to the participant, and their session variables are set. Does not show a consent form and instead automatically redirects them to the next page.
`/create_participant_nc`	This is the same as `/create_participant`, except that the participant is not assigned a condition (defaults to 0).

And the last page must be `/end`. So the simplest study would just show the participant a consent page and then the final page, which would show a message or a message and a completion code.

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

Further pages can be added to the `PAGE_LIST` as needed. For an overview of the available pages, see *Built-in Routes*.

For example, we could add a questionnaire to the PAGE_LIST, so that the participant must complete a questionnaire before the study ends.

```
PAGE_LIST = [
    {name='Consent', path='consent'},
    {name='Questionnaire', path='questionnaire/example'},
    {name='End', path='end'}
]
```

You could also define your own pages (see *Adding Custom Routes (Blueprints)*) and use these within `PAGE_LIST`.

```
PAGE_LIST = [
    {name='Consent', path='consent'},
    {name='My Custom Page', path='my_page'},
    {name='End', path='end'}
]
```

6.1 Routing by Condition

If using conditions (see *Project Configuration*), then one simple way of making use of these is within the PAGE_LIST. Here, you can define specific page sequences for users based on their assigned condition number. These are added in place of a single page. For example:

```
PAGE_LIST = [
    {name='Consent', path='consent'},
    {name='Questionnaire', path='questionnaire/example'},
    {conditional_routing=[
        {condition=1, page_list=[
            {name='Task Instructions', path='instructions/1'}
        ]},
        {condition=2, page_list=[
            {name='Task Instructions', path='instructions/2'}
        ]}
    ]},
    {name='End', path='end'}
]
```

CHAPTER

SEVEN

BUILT-IN ROUTES

7.1 Routes For Use in PAGE_LIST

7.1.1 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.

7.1.2 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.

7.2 Developer Routes

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_restart**()

/restart

Use this if you ever need to the user to start the experiment over for any reason. This tries to clear out all the cookies.

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_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.

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**(*questionType: 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.

CHAPTER

EIGHT

ASSIGNING AND MAKING USE OF CONDITIONS

Conditions are set up in your project's configuration `.toml` file, with the `CONDITIONS` variable.

Leave the value an empty list (`[]`) if you do not need multiple conditions. The format for multiple conditions is `[{label='condition 1', enabled=true}, {label='condition 2', enabled=true}]`. The first entry in the list is condition 1, the second is condition 2, etc. A participant without an explicitly assigned condition will have a condition of `0`.

8.1 Assigning Conditions

By default, a condition will be assigned to the participant if they visit the `/consent`, `/create_participant`, or `/assign_condition` route. They will be assigned to whichever condition has the fewest number of participants, with the lowest number condition being chosen in the case of a tie. Participants who have abandoned the study without completing it will not be considered within the count of participants in the condition group.

8.2 Using Conditions

These condition group numbers can be used within your project configurations `PAGE_LIST` variable. See *Routing Participants* for more details.

When developing custom pages, you can access the participant's assigned condition via the `condition` session variable: `session['condition']`.

CHAPTER

NINE

QUESTIONNAIRES

Bride of Frankensystem includes a simple questionnaire system for displaying static questionnaires. These questionnaires are defined in the JSON language so that they can be easily re-used across multiple projects and shared.

For an example of the JSON markup for a complete questionnaire, see *Example Questionnaire*.

9.1 Creating Questionnaires

Questionnaires are defined as JSON files and placed within your project under the `/questionnaires` directory.

9.1.1 Syntax of a Questionnaire

Questionnaires are created by defining the structure of the questionnaire within a `.json` file. This stucture is made up of key-value pairs. For an introduction to the syntax and structure of JSON and see this tutorial.

The overall structure of the `.json` files that define questionnaires looks like this:

```
{
    "title": "",
    "reference": "",
    "doi": "",
    "instructions": "",
    "code":"",
    "questions": [ ],
    "participant_calculations": {}
}
```

Table 1: Questionnaire JSON keys

Key	Data Type	Description of value
`title`	string	Optional field used to label the questionnaire for reference only. This is never shown to participants and only shows up on the preview in the Administration section.
`reference`	string	Optional field to store the citation information for the questionnaire. This is never shown to participants and only shows up on the preview in the Administration section.
`doi`	string	Optional field to store the doi string relating to the citation. This is never shown to participants and only shows up as a link on the preview in the Administration section.
`instructions`	string	Instructions to appear at the top of the form, before any questions are asked. Supports HTML. Optional.
`code`	string	For advanced users. JavaScript code to be executed at run time. Optional.
`questions`	list	A list of questions. Each question is defined as a dictionary of key-value pairs. Question types are shown below.
`participant_calculations`	dictionary	A dictionary of named calculated fields. The keys are the name of the calculated field and the value is the calculation. The calculation is Python-compatible code that gets executed for the calculation. Question IDs can be used as variables, and the code is not sand-boxed in any way, so some caution is required.

Note: You can safely add new keys here as desired and they will be ignored by BOFS. This can be used to add relevant metadata for the questionnaire, including the citation, questionnaire name, or instructions to the researcher.

9.1.2 Adding Questions

On line 7 of the example of the JSON structure for questionnaires, you can see that there is a blank list (*[]*). It is within this list that new questions can be added to the questionnaire. For examples of how to define questions of different types, see the example below. For a detailed reference, see *Built-in Question Types*.

9.1.3 Example Questionnaire

An example questionnaire demonstrating every question type.

Listing 1: example.json

```
{
    "title":"Example Questionnaire",
    "instructions":"Please rate each video game experience....",
    "code":"",
    "questions":[
        {
            "questiontype":"textview",
            "title":"I am a title",
            "text":"This allows you to display descriptive or informative text."
        },
```

(continues on next page)

(continued from previous page)

```
        {
            "questiontype":"radiogrid",
            "instructions":"choose option...",
            "id":"radio_1",
            "shuffle":"true",
            "labels":[
                "I hate it!",
                "",
                "Neutral",
                "",
                "I love it!"
            ],
            "questions":[
                {
                    "id":"q_1",
                    "text":"dfdjs;aljd"
                },
                {
                    "id":"q_2",
                    "text":"hahahahaha"
                },
                {
                    "id":"q_3",
                    "text":"muahahahaha"
                }
            ]
        },
        {
            "questiontype":"radiolist",
            "instructions":"choose one option...",
            "id":"radiolist_1",
            "labels":[
                "Yes",
                "Maybe",
                "No"
            ]
        },
        {
            "questiontype":"checklist",
            "instructions":"choose any options...",
            "id":"checklist_1",
            "shuffle":true,
            "questions":[
                {
                    "id":"cl_1",
                    "text":"Option 1"
                },
                {
                    "id":"cl_2",
                    "text":"Option 2"
                },
                {
```

(continues on next page)

(continued from previous page)

```
                    "id":"cl_3",
                    "text":"Option 3"
                }
            ]
        },
        {
            "questiontype":"slider",
            "instructions":"I am a slider",
            "id":"slider_1",
            "left":"left",
            "right":"right",
            "tick_count":5,
            "width":500
        },
        {
            "questiontype":"field",
            "instructions":"enter text",
            "placeholder":"I am a placeholder",
            "id":"input_1"
        },
        {
            "questiontype": "num_field",
            "instructions": "enter a number",
            "id": "number_field"
        },
        {
            "id": "dropdownexample",
            "questiontype":"drop_down",
            "instructions":"choose one",
            "items":[
                "apples",
                "oranges",
                "watermelon"
            ]
        },
        {
            "questiontype":"multi_field",
            "id":"big",
            "placeholder":"I am holding the place",
            "instructions":"big text field",
            "height":"100"
        }
    ]
}
```

9.1.4 Participant Calculations

You can perform calculations with the data captured by this questionnaire. These calculations are done on a per-participant basis. So you can, for example, calculate a participant's score on a set of questions, or work out the value associated with a particular scale.

The calculations must be placed within the `participant_calculations` dictionary. The key is the variable name (this will become the column header in the export) and the value is Python code that represents the calculation. The Python code can reference any of the question ``id``s within the questionnaire.

Example

```
{
    /* title, questions, etc. hidden for brevity. */
    "participant_calculations":
    {
        "MyCalculatedVariable": "mean([id01, id02, id03, id04, id05])"
    }
}
```

The following functions are supported: `mean`, `variance`, `std`, and `median`.

Below is a complete questionnaire featuring calculations:

Listing 2: grid.json

```
{
    "title":"Example Questionnaire 2",
    "instructions":"",
    "code":"",
    "questions":[
        {
            "questiontype":"radiogrid",
            "instructions":"Rate your agreement",
            "id":"radio_1",
            "shuffle":"true",
            "labels":[
                "Strongly disagree",
                "",
                "Neutral",
                "",
                "Strongly agree"
            ],
            "q_text":[
                {
                    "id":"q1",
                    "text":"This example is fantastic"
                },
                {
                    "id":"q2",
                    "text":"This example could be improved"
                },
                {
                    "id":"q3",
                    "text":"This example is the highlight of my day"
```

(continues on next page)

(continued from previous page)

```
                }
            ]
        }
    ],
  "participant_calculations": {
    "ExampleQualityMean": "mean([q1, 6-q2 , q3])",
    "ExampleQualitySum": "q1 + 6-q2 + q3"
  }
}
```

9.2 Previewing Questionnaires

Every questionnaire inside of the `/questionnaires` directory can be previewed from within BOFS via the admin panel. This is accessed at `http://<host>:<port>/admin`. `host` is the IP address of the system on which BOFS is running and if running locally will be `127.0.0.1`. `port` is defined within your project's configuration file and is shown when the project is run via the `BOFS` command.

The preview will inform you of any and all syntax errors, and offer to add columns to the database if this questionnaire is one listed in the page sequence and it detects that there are new columns to add. For this reason, it is recommended that you use this preview feature when developing new questionnaires.

9.3 Adding Questionnaires to Your Study

Questionnaires must be added to the `PAGE_LIST` variable within your project's configuration file to be displayed to the end user. After adding the questionnaires `.json` file to */questionnaires*, you then add a entry to `PAGE_LIST`.

For example, the `my_questionnaire.json` file will have the `PAGE_LIST` entry:

```
{'name': 'Questionnaire', 'path': 'questionnaire/my_questionnaire'}
```

Doing this signals BOFS that a database table should be created for the questionnaire. Once the study is run, this table will get created.

If changes are made to the questionnaire, the table will *not* get updated to match. This is a problem any time a question ID changes or whenever a new question gets added. The easiest way to resolve this is to delete the database file and restart the project. This does mean you lose your data, so this option is only viable during development.

For a live database with participant data, you have three options:

1. **`DROP` the table.**
    - You lose data, but this time it's just the one table.
2. **Add the missing columns.**
    - This can get ugly as it has the potential to leave in old columns when renaming.
    - This can be done automatically within the system when previewing questionnaires.
    - This doesn't fix issues with mismatched data types if the type was changed
3. **Alter the table directly.**
    - This is tidiest solution if you need to modify a database with existing data. With SQLite, you would have to rename the table, create a new table with the correct structure and name, then copy over the

data with an `INSERT INTO ... SELECT ...` statement. Other DBMS allow you to alter the table more directly.

If you drop the table or delete the database, you will need to restart the app in order for the table to be generated.

CHAPTER

TEN

BUILT-IN QUESTION TYPES

The following attributes are common to every type of question (except radiogrid).

- `id`: string - Your field's unique id. **This must be completely unique within each questionnaire.**
- `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.

Currently, the following types of input are supported:

- `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.
- `radiolist` - Select one option out of a list
- `checklist` - Select multiple options out of a list
- `slider` - Drag the slider to a numeric value, with optional labels on the left and right.
- `field` - Simple single-line text entry
- `num_field` - Input a single number
- `multi_field` - Multi-line text entry
- `drop_down` - Select one option from a drop-down list
- `textview` - Display plain text (HTML syntax is supported)

10.1 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.
- A selection by the user is always required

Properties

- `instructions`: any text that is needed directly above the radiogrid (optional, string)
- `shuffle`: should the question order be shuffled? (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)

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"
        }
    ]
}
```

10.2 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`)
- `shuffle`: whether or not the possible response labels should be shuffled (optional, boolean: `true` or `false`, default is `false`)
- `horizontal`: should be 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)

Example

```
{
    "questiontype":"radiolist",
```

(continues on next page)

(continued from previous page)

```
    "instructions":"Do you eat meat?",
    "id":"radiolist_1",
    "horizontal": false,
    "required": true,
    "labels":[
        "Always",
        "Sometimes",
        "Never"
    ]
}
```

10.3 checklist

`questiontype == 'checklist'`

Properties

- `id`: unique id for checklist (required, string)
- `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`)
- `questions`: one for each checkbox. Each needs text and a unique ID. (required)

Example

```
{
    "questiontype":"checklist",
    "instructions":"choose any options...",
    "id":"checklist_1",
    "shuffle":true,
    "horizontal": false,
    "questions":[
        {
            "id":"cl_1",
            "text":"Option 1"
        },
        {
            "id":"cl_2",
            "text":"Option 2"
        },
        {
            "id":"cl_3",
            "text":"Option 3"
        }
    ]
}
```

10.4 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
}
```

10.5 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`)

Example

```
{
    "questiontype": "field",
    "instructions": "enter text",
    "placeholder": "I am a placeholder",
    "id": "input_1"
}
```

10.6 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`)

Example

```
{
    "questiontype": "num_field",
    "datatype": "integer",
    "instructions": "enter a number",
    "id": "input_1"
}
```

10.7 multi_field

`questiontype == 'multi_field'`

- Multi-line text field.

Properites

- `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`)

Example

```
{
    "questiontype": "multi_field",
    "id": "big",
    "placeholder": "I am holding the place",
    "instructions": "big text field",
    "height": 100
}
```

10.8 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",
    "instructions": "Which of the listed fruits is your favorite?",
    "items": [
        "apples", "oranges", "watermelon"
    ]
}
```

10.9 textview

`questiontype == 'textview'`

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."
}
```

CHAPTER

ELEVEN

CREATING CUSTOM QUESTIONS

It is possible to create your own custom question types to use within questionnaires. These are defined inside of your templates directory (for your blueprint, or your project), within the questions folder, so `/templates/questions/`.

Custom question types are defined in an HTML file that leverages Jinja 2 templating. To make use of it when creating a questionnaire, you specify a new question with a `questiontype` that matches the filename for your question (less the `.html` extension).

So for a question type of "custom", you need a .html file, `/templates/questions/custom.html`. You would use that custom question within your questionnaire's JSON via:

```
{
  "questiontype": "custom",
  "instructions": "I am special",
  "id": "special"
}
```

The keys shown here are the minimum required. The `instructions` will show above the questions like any other BOFS question, and the `id` gets used to generate the column within the database table (note that the `id` isn't technically required, but without it, nothing can be saved to the database).

In the question template, you can use any HTML, JavaScript, Jinja 2 control structures, etc. that you want. If an input with a `name` matches the `id` specified in the JSON question definition is in your question template, then that input field will get written to the database when the form is submitted.

An important aspect about how the question template works is that all of the JSON data that you define about the question in your questionnaire's JSON file gets passed to the template file as the `question` Jinja 2 variable. Consider the example question template:

Listing 1: custom.html

```
<p>Your condition is: {{ session['condition'] }}</p>
<p>You chose "{{ participant.questionnaire("example").radiolist_1 }}" for radiolist_1 on␣
↪the example questionnaire.</p>
<p><b>Do you still agree with this?</b></p>

<div class="form-check">
    <input class="form-check-input" type="radio" name="{{ question.id }}" id="{{␣
↪question.id }}_no" value="No">
    <label class="form-check-label" for="{{ question.id }}_no">No</label>
</div>

<div class="form-check">
    <input class="form-check-input" type="radio" name="{{ question.id }}" id="{{␣
```

(continues on next page)

(continued from previous page)

```
↪question.id }}_yes" value="Yes">
    <label class="form-check-label" for="{{ question.id }}_yes">Yes</label>
</div>
```

This template does some things that standard questions cannot do. First, it uses Jinja 2 to access the BOFS `session` variable and print out the participant's assigned condition. Consider what you could use this for. For example, you might use a Jinja 2 if statement to show something different based on assigned condition.

Another thing it does is access the participant's questionnaire data. When rendering this template, the current instance of the Participant class (i.e., the current participant's data) is passed to the template. This gives access to, for example, the `questionnaire()` method, where you can access the data submitted to a questionnaire. This means you can show participants what their past responses were, or do something different based on a past response.

Finally, notice that there is a `questionnaire` variable that was used, and the name of the radio input was set based on this.

11.1 Multiple IDs

One final note about IDs – it is possible to have multiple IDs instead of just the one. This would let you have many inputs associated with the one question type.

This is done just like the checklist and radiogrid question types:

```
{
  "questiontype": "custom2",
  "instructions": "I am a custom question with multiple inputs",
  "questions": [
    {
      "id": "question_1",
    },
    {
      "id": "question_2",
    },
    {
      "id": "question_3",
    }
  ]
}
```

And then within your template you can use a Jinja 2 for loop:

Listing 2: custom2.html

```
<div>
{% for sub_question in question.questions %}
Enter something: <input type="text" name="{{ sub_question.id }}" id="{{ sub_question.id }
↪}"> <br>
{% endfor %}
</div>
```

CHAPTER

TWELVE

INSTRUCTION PAGES

It is possible to include static HTML pages that can be shown to your participants as part of the study. The intention is the easy inclusion of instruction pages, but the pages can really be any static content that you like. The pages make use of the theme utilized by the rest of your study and always show a "continue" button.

To make use of this, place a `.html` within your project into the `/templates/instructions` directory (of your project or of your blueprint). For example: `/templates/instructions/my_page.html`.

This page can then be added to your project's `PAGE_LIST`, for example:

```
PAGE_LIST = [
    {name='Consent', path='consent'},
    {name='My Instructions', path='instructions/example_instructions'},
    {name='End', path='end'}
]
```

These html files are also Jinja2 templates, and therefore have access to the same variables.

See also *Adding Custom Routes (Blueprints)* (Templates (HTML) and Static Files) and *Simple HTML Pages*.

CHAPTER

THIRTEEN

SIMPLE HTML PAGES

You can show simple HTML pages as part of your study. These are similar to the instruction pages, but *no* "continue" button is shown, so you have to implement your own redirection (for example, to `/redirect_next_page`).

To make use of this, place a `.html` within your project into the `/templates/simple` directory (of your project or of your blueprint). For example: `/templates/simple/my_page.html`.

This page can then be added to your project's `PAGE_LIST`, for example:

```
PAGE_LIST = [
    {name='Consent', path='consent'},
    {name='My Instructions', path='simple/my_page'},
    {name='End', path='end'}
]
```

These html files are also Jinja2 templates, and therefore have access to the same variables.

See also *Adding Custom Routes (Blueprints)* (Templates (HTML) and Static Files) and *Instruction Pages*.

CHAPTER

FOURTEEN

ADDING CUSTOM ROUTES (BLUEPRINTS)

Custom web pages and functionality can be added to your Bride of Frankensystem projects by making use of Flask Blueprints. The Flask documentation describes blueprints as a way to organize groups of related views and other code ("view" refers to the web page seen by end users). You may find it helpful to look through Flask's documentation to see code for some example pages.

The code you write for BOFS Blueprints is like any other Flask blueprint, with the exception that your files must be structured in a specific way. BOFS blueprints must be in their own directory inside of your project's root directory.

For example, consider a blueprint called "my_blueprint":

```
/my_blueprint/__init__.py
/my_blueprint/views.py
/my_blueprint/templates
/my_blueprint/static
```

- `__init__.py` is a mandatory file to indicate that this is a Python package. It will typically be empty.
- `views.py` is a mandatory file that contains your code for your custom Flask views (web pages).

14.1 The Views File

Views are made up of three different components.

- **HTML templates**, which are defined in HTML and the Jinja2 templating system and are placed within `/my_blueprint/templates`.
- **Static files**, such as images and Javascript and are placed within `/my_blueprint/static`.
- **Python code** that controls what BOFS should do and display, within `/my_blueprint/views.py`.

Your `views.py` will initially look something like this (feel free to use this as a starting point in your own blueprint):

Listing 1: views.py

```
from flask import Blueprint, render_template
from BOFS.util import *
from BOFS.globals import db

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

The arguments passed to the `Blueprint()` constructor generally do not need to be adjusted (aside from the name, "my_blueprint"). If you want to learn more about how this works then refer to the Flask documentation.

14.2 Creating Routes

Routes define the actual pages that users can visit. This documentation will not go into much detail as to how routes work, so if you have further questions, do visit the Flask documentation and see if your questions are answered there.

14.2.1 Python Code

In BOFS, your routes will look something like this (this route is directly from the advanced example):

Listing 2: views.py

```
# preceding code in views.py omitted for brevity.

@my_blueprint.route("/task", methods=['POST', 'GET'])
@verify_correct_page
@verify_session_valid
def task():
    incorrect = None

    if request.method == 'POST':
        log = db.answers()  # This database table was defined in /advanced_example/
↪tables/answers.json
        log.participantID = session['participantID']
        log.answer = request.form['answer']

        db.session.add(log)
        db.session.commit()

        if log.answer.lower() == "linux":
            return redirect("/redirect_next_page")
        incorrect = True

    return render_template("task.html", example="This is example text.",␣
↪incorrect=incorrect)
```

This `task()` function has three decorators on it. The first one (`@my_blueprint.route()`) registers the function as a route associated with the `/task` URL, and indicates that the URL will accept POST requests (e.g., form submissions) and GET requests (in which the user asks to see what is at that URL). Note that within the function, `request.method` is checked and if it is a POST request, then something is added to the database.

The second decorator (`@verify_correct_page`) ensures that the user does not access this page except for when accessed by following the order defined within `PAGE_LIST`.

The third decorator (`@verify_session_valid`) checks that the user has the correct session values set and if not, redirects them to the first page listed in `PAGE_LIST`.

This function has two return values. At the bottom, the return value of the function in this example renders a template that will show the user the task. Two variables are sent to the template that configure aspects of how the template should render to the participant (`example` and `incorrect`). If a POST request was made, then an alternative return value is

to do a redirection to `/redirect_next_page`, which takes the user to the next page in `PAGE_LIST` after the `/task` page.

Tip: To better understand this example, you may want to run the provided advanced example project and see what `/task` looks like for yourself.

14.2.2 Database Tables

This example makes use of a database table. For more information on how to use database tables in your custom routes, see *Custom Database Tables (Models)*.

14.2.3 Templates (HTML) and Static Files

The presentation of the page is defined in the `task.html` template.

Listing 3: task.html

```
{% extends "template.html" %}
{% block head %}
{% endblock %}

{% block content %}
    {% if incorrect %}
        <h1>You were wrong! Try again.</h1>
    {% else %}
        <div>
            <h3>Some Information</h3>
            <ul>
                <li>Your participant ID is {{ session['participantID'] }}.</li>
                <li>You were assigned to condition {{ session['condition'] }}.</li>
                <li>This is the value of <tt>example</tt>: {{ example }}</li>
            </ul>
        </div>
    {% endif %}

    <hr>

    <form id="form" action="#" method="post">
        <p><img src="{{ url_for('my_blueprint.static', filename='tux.png') }}"></p>

        <p>
            <label for="answer">This is the mascot for which operating system?</label>
            <input type="text" id="answer" name="answer" required>
        </p>

        <input type="submit" name="submit" value="Submit Answer">
    </form>
{% endblock %}
```

This template extends `template.html`, which means that it will have the look and feel of other pages in BOFS. The `template.html` has two blocks, “head” and “content”. By defining them in your own template (as in `task.html`),

you can add your own content to the head of the page (useful for CSS, etc.) as well the body of the page.

This template demonstrates how to use **variables** and **static** content. In particular, `incorrect` and `example` were variables passed to the template from `render_template()` and are now being used, as well as `session`, which is always available to be used within the template. Static content is being demonstrated via displaying an image located at `/my_blueprint/static/tux.png`.

In addition to `session`, you will always have access to the following variables within your templates:

- `session['participantID']` - Accessible on routes that are decorated with `@verify_session_valid`.
- `session['condition']` - Accessible on routes that `@verify_session_valid`.
- `debug` - A boolean indicating whether the project is being run in debug mode.
- `config[...]` - Flask/BOFS configuration settings.

For more details on `url_for()`, see the Flask documentation on url_for().

For more details on how Jinja2 templates work, see the Flask documentation on templates.

CHAPTER

FIFTEEN

CUSTOM DATABASE TABLES (MODELS)

If you need to record your own data to the database, then you can do so by defining custom database tables. Tables can be placed within the `/tables` directory of your project or within a blueprint at `/<blueprint_name>/tables`.

15.1 Column Types

Tables are made up of columns. Each column can have its own data type. Possible data types are `integer`, `float`, `boolean`, or `string`.

Like questionnaires, tables are defined in JSON format. Below is an example of a table with every different possible type of data. Notice that each column has a "default" value; this entry can be omitted if not required. Additionally, notice that if a "type" is not specified, then it defaults to being a string.

```
{
  "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"
    }
  }
}
```

15.2 Calculated Export Fields

By default, you can view and export data collected in your database tables on the Administration section of BOF, by viewing a table and clicking "Export as CSV". However, this approach requires additional data processing for all but the simplest use cases, and relevant information is often missing from this table (e.g., assigned condition) and must be added in after the export. A more automated approach that allows you to include all relevant information and transform your data in different ways involves defining rules for exporting data. Once defined, your desired data will be exported alongside all of the questionnaire data on the "Export" page in a "wide" format.

These exports use the terminology and features of SQL, so if further clarification is needed, you can refer to a SQL tutorial (e.g., https://www.sqlitetutorial.net/).

Exports can be defined within the same file as the table by including an "exports" entry in the file.

```
{
  "columns": {},
  "exports": []
}
```

The most simple exports will simply be use a `MIN`, `MAX`, `SUM`, `COUNT`, or `AVG` aggregate function to calculate the minimum, maximum, sum, count, or average of the entries. For example, the on table defined below, numbers entered by the user can be stored in a table and the calculated field reports a count of the numbers entered for each user.

```
{
  "columns": {
    "your_number": {"type": "integer"}
  },
  "exports": [
    {
      "fields" : {"total_numbers": "count(your_number)"}
    }
  ]
}
```

Each export supports the following keys:

Table 1: JSON keys for tables

Key	Description
fields (required)	A dictionary of fields to export. This dictionary's keys are the names you want for your column, and the values are the data you want to export. This data can be the database table's column names (e.g., my_column) or column expressions (e.g., sum(my_column)). Note: you will want to include an aggregate function in your field's definition (MIN, MAX, SUM, COUNT, or AVG) unless there is only one row in your table per each participant.
filter (optional)	This is a SQL WHERE expression. This can be used to omit rows from the table that are not of interest (e.g., my_column > 1 or my_column != 'text').
group_by (optional)	This a SQL GROUP BY expression. If the table you are exporting from has groups of repeated measures that you want to analyze separately then you will need to make use of this. Each unique entry in the grouped column will have a corresponding column in the export. For example, if you had participants complete a task over multiple days, you could group by day and you will end up with a column for each day (e.g., monday_my_column, tuesday_my_column, etc.). It is also possible to group by multiple columns by specifying a list of column names (each a string).
order_by (optional)	This is a SQL ORDER BY expression. It determines the order of the columns in the export.
having (optional)	This a SQL HAVING expression. It can only be used if group_by is used.

Let's consider a more complicated example. In this example, there are 5 columns, two integers, one float, and two strings. What is being measured is progress within a game, with one entry in the table being one level. Multiple sessions of the game were played, and each had a unique name. The data being exported is the total levels finished over each play session, the total deaths for each play session, the time taken to complete three intro levels, and the count of of three intro levels completed.

```
{
  "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)"
      }
    },
    {
      "filter": "levelName IN ('Intro1', 'Intro2', 'Intro3')",
      "fields": {
        "tutorialLevelsTime": "sum(levelTime)",
        "tutorialLevelsCompleted": "sum(finishedLevel = 'True')"
      }
    }
```

(continues on next page)

(continued from previous page)

```
  ]
}
```

15.3 Accessing Tables from Python

From your python code, import `db` from `BOFS.globals`.

```
from BOFS.globals import db
```

The `db` object provides access to all database-related functionality.

15.3.1 Reading Data

Queries can be completed by using `db.session`. Refer to the SQLAlchemy documentation on using the session.

Example: Getting a list of all participants who have finished the experiment.

```
finished_participants = db.session.query(db.Participant).filter(db.Participant.finished
↪== True).all()
```

See the SQLAlchemy documentation on querying with the ORM. for more details.

15.3.2 Inserting Data

Using SQLAlchemy you create new database rows by creating new instances of your model classes. You then set your attributes, indicate to the session that you want to add a new row, and commit your changes.

For example:

```
log = db.answers()  # This database table was defined in /advanced_example/tables/
↪answers.json
log.participantID = session['participantID']
log.answer = request.form['answer']

db.session.add(log)
db.session.commit()
```

See the SQLAlchemy documentation on adding and updating objects for more details.

INDEX

R

S